Ergodicity in Software Systems

by

Zahra Nikdel
B.Sc., Iran University of Science and Technology, 2004
M.Sc., Sharif University of Technology, 2007

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering,
University of Victoria,
Victoria, BC, Canada

Ergodicity in Software Systems

by

Zahra Nikdel
B.Sc., Iran University of Science and Technology, 2004
M.Sc., Sharif University of Technology, 2007

Supervisory Committee

Dr. Stephen W. Neville, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Michael L. McGuire, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Sudhakar Gantie, Outside Member
(Department of Computer Science)

ABSTRACT

This dissertation applies dynamical systems theory (DST) to formally investigate the statistical run-time performance predictability of arbitrary scale software-centric systems, ranging from small-scale embedded systems to modern large-scale cloud-deployed container and virtual machine based distributed systems. The research focuses on verifying Birkhoff's Ergodic Theorem (BET) compliance for queuing network (QN) models of deployed software systems against BET-compliant Poisson and bursty incoming workloads. The approach applies a previously developed extension of Maurer's Turing-reducible computer model, termed the Extended Maurer Model (EMM), as the requisite bridge between classical QN software system models and the DST-based BET tenets underlying classical Markovian QN analysis approaches. Moreover, it is shown that as the EMM describes a σ-finite measure space, a known ergodicity equivalency theorem can be used to develop a formal DST analysis approach to prove when BET left-hand and right-hand side conditions can be met for run-time software systems performance measures. More specifically, formally proving recurrence holds within large-scale systems, as required by classical Markovian analyses, has remained an open problem. By comparison, this research shows that this issue can be addressed by instead assessing when and why: i) wandering sets of non-zero measure arise and ii) event space variations and non-invariant DST measures arise, given (i) and (ii) are mathematically known to be equivalent to assessing recurrence within σ-finite measure spaces.

This theoretical DST analysis of QNs defined over the EMM representation of run-time software system behaviors then leads to the development of four pragmatically easily measurable and implementable software engineering design rules that can be used to assess when and why a given deployed software system will (or will not) exhibit statistically predictable run-time behaviors. These design rules are then applied to develop a sufficiently rich cloud-deployed software system simulation framework, which includes incoming statistical workloads, cloud networking fabric, physical server, virtual machine, and container deployment regimes, fair and real-time OS scheduling, and background physical server workloads. This simulation framework is then used to validate the DST theory BET-compliance analysis insights through a detailed set of software system run-time deployment scenarios, both for an industry-held exemplar system and for emerging industry deployment trends.

To our knowledge, this is the first set of research to formally assess run-time software system BET-compliance for systems of arbitrary scale and complexities. Moreover, it is the first work to show, through theory and simulation-based validation, that modern software systems exist as highly complex dynamical systems that can concurrently admit BET-compliant and BET non-compliant performance measures, while also admitting measures that can dynamically transition back and forth between BET-compliance and non-compliance as the system runs. As engineering can be summarily defined in terms of the need to build systems and solutions that behave predictably in their real-world operation under all likely deployment scenarios and environments, the developed novel insight into the core DST complexity of software systems helps to partially explain why the software engineering of modern industry-scale software systems has remained a challenging and largely open problem, outside of specific quite tailored regimes, i.e., the regimes that can be seen to follow this dissertation's developed software engineering design rules.

The issue of BET-compliance has wide applicability across many areas, spanning statistical run-time performance predictability, control theory, machine learning (ML) and artificial intelligence (AI), quantum computing, etc. The insights and theoretical framework developed in this research have wide potential applicability beyond just the core software engineering need to develop and deploy software systems that behave predictably, in the sense of remaining within a defined set of statistical behavioral bounds. More particularly, emerging areas of potentially applicable research include: the use of formal control theory approaches within cloud elastic services; the safety and operationally critical aspects of Smart Cities, Smart Grids, autonomous vehicles and vehicle networks; when, why, and how AI/ML approaches can be applied to accurately predict software system run-time behaviors; BET-compliance within cyber-security focused regimes and solutions; etc. As such, we hope this work will be seen as a useful and insightful contribution to advancing the formal engineering of software-centric systems and solutions.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS AND DEDICATION

The knowledge of anything, since all things have causes, is not acquired or complete unless it is known by its causes.
Ibn Sina (Avicenna)

PhD is a rewarding journey, which would not be possible without the support of many people. As my journey is near to its end, I would like to take this opportunity to thank these amazing people who inspired me during the ups and downs of this pleasant experience.

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Stephen W. Neville for giving me the opportunity to pursue my studies under his continuing guidance and support. He encouraged me in all aspects of my research and writing of this dissertation. From him, I have gained invaluable knowledge, both in my academic endeavors and in life, as he opened new perspectives and pathways for me.

I would also like to thank Mr. Kevin Jones for his continuous help and support, as well as his incredible creativity in setting up the required framework for the simulations.

I express my profound gratitude to my parents and husband, who have always supported me at every stage of life, including my undergraduate and postgraduate studies. I also thank my sisters for their love and encouragement during times of trouble and doubt. I want to extend my heartfelt thanks to my sons, Mustafa, Taha, and Ebrahim, who have made my life so incredible and prosperous each and every day. These few words cannot express my deepest appreciation for my family's selfless patience, unconditional love, and endless support during these past years.

Lastly, and most importantly, I dedicate this thesis to my beloved mother. Her unwavering encouragement and invaluable support, particularly in helping raise my youngest child during my studies, has been invaluable.

Zahra Nikdel
Victoria, BC, Canada
July 1^{st} (Canada Day), 2024

Chapter 1

Introduction

1.1 Software System Ergodicity and Run-time Performance Predictability

Ergodicity is a fundamental concept in dynamical systems and statistical mechanics that implies the equivalence of time averages and ensemble averages for a system's properties. In simpler terms, if a system is ergodic, its behavior averaged over time will be the same as its behavior averaged over different initial states. This concept which has been formulated as Birkhoff Ergodic Theorem (BET) is crucial for predicting long-term behavior based on shorter-term and past observations. Therefore, ergodicity and consequently being BET-compliance is also a very important property in the world of software systems because software systems are cornerstones of modern societies, influencing everything from the devices we use for personal communication and entertainment, to the systems that support our critical infrastructure and services. Increasingly, it is crucial that software systems perform reliably and predictably to meet societies' needs and expectations.

In general, for modern user-focused systems, such as Software as a Service (SaaS) solutions, Massive Multiplayer Online Games (MMOG), etc., the Quality of Service (QoS) and the Quality of Experience (QoE) are key drivers for social media comments and ratings, which can have significant market and economic impacts for the producers of such systems. On the other hand, cloud computing has become the dominant method for commercially deploying distributed software systems and has transformed cost of service delivery and performance expectations among users and developers. The increasing migrating of service providers towards the cloud, is

also given a rise to increasing demands for such service to provide faster software development processes along with improved QoS/QoE, performance, reliability and security. As a result, approaches such as agile development are common place and drive on-going updates and changes to the deployed systems, which although easily accomplished through cloud deployment APIs and management services, they must not negatively impact system's performance behavior. Furthermore, although the emergence of auto-scaling serverless container-based technologies, such as Docker [6], Kubernetes [7], Knative [8], etc., has helped mitigating deployment and scaling issues, but the effects of cloud management processes, Virtual Machines (VMs), containers, etc. on software systems predictability have not been thoroughly and well-studied neither analytically nor experimentally. Particularly, the significance and difficulties of being able to reliably predict Large-scale Distributed Software Systems (LDSS) run-time performance in cloud environments have been increasingly recognized by the researchers, cloud customers and service providers, e.g. [9–14]. Most importantly, with the on-going emergence of application areas such as smart cities [15–17], smart grids [18–20], autonomous vehicle and vehicular networks [21–24], etc., the impact of software systems' run-time behavior predictability will become even more critical as such domains intersect with significant human and environmental health and safety concerns which can literally mean the difference between life and death.

However, despite extensive research efforts in the field of run-time performance predictability which is tightly related to ergodicity, the challenge of explicating the core factors that contribute to the gain or loss of run-time performance predictability in software systems remains a formidable task because there is no study that proves under which conditions we can assure a system behave predictable. The current solutions have yet to yield a comprehensive understanding of this issue. The following section provides a review of the most widely-used approaches in this field and outlines the challenges that are currently being encountered.

1.2 Solution Approaches and Challenges

The problem of run-time performance predictability is a challenging subject in the software engineering field and a considerable amount of research has been devoted to this topic, resulting in the proposal of numerous techniques and methods that can be organized in the following categories, which include but are not limited to:

- **Architecture Analyses**: In this approach, the documented architecture of the given software systems is examined and then analyzed through available mathematical modeling approaches to determine the systems performance probabilities. As one example among many, in [25], Sharma employs Discrete Time Markov chainss (DTMCs)[26] to predict the overall behavior of the system based on its architecture. Although these approaches typically provide a general understanding of system's likely performance and cost envelop but for non-trivial and complex systems, they do not provide the statistical run-time performance prediction, as will be defined formally in section 1.3.

- **Simulation-based Testing**: System architectures can further be evaluated through simulating the system under a variety of workload conditions. A wide number of simulation platforms and approaches have been developed including for web-sites, web-based and distributed systems solutions [27–31]. As an example, Palladio [27] is a quantitative architecture simulation approach for modeling quality of service in early design states for web applications. A more recent work is provided by [29], suggests a modeling approach based on queuing-based analytical models and simulations for predicting the performance of big data applications in cloud environments. Simulations rest on mathematical models of the systems being simulated, where this may include rich simulation of network packet flows and routings [30; 31], with Monte Carlo approaches [32] generally being applied to account for statistical variations between simulations runs. It is noteworthy that Monte Carlo approaches aim to achieve an ensemble average that is equivalent to the time average. However, this assumption holds valid only for an ergodic system, as stated by BET [33] and to date, there is no proof of ergodicity for software systems, to the best of our knowledge. On the other hand, although simulation based approaches are generally low cost but can abstract out important system characteristics that can impact run-time performance.

- **Emulation-based testing**: A significantly more costly performance testing approach is via emulation in which the full software system executable codebase is deployed into an isolated and controlled computer and network cluster [34–38]. The running systems is then exercised by either synthetically generated workloads or replays of recorded real-world workload in a controlled and repeatable manner. In this way, the full executable software system's performance

can be assessed inclusive of all of its potential behaviours that simulation-based testing may have abstracted out. As an example, [4] uses data from a real-world distributed system to develop a realistic model for workload generation for different scenarios, and then analyzing the impact on the response time distributions across multiple runs on a test deployment of the system on the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) [39].

- **Real-world or Ad hoc testing**: Due to the expense of emulation-based testing, a practical and commonly used alternative is to apply synthetic or real-world replayed workloads to a fully deployed version of the software system to be tested. For example, part of the testing of a new release of a cloud-deployed SaaS may include performance testing of the system against previously collected real-world workloads [40] which may then be synthetically enriched to explore potential "what-if" workload conditions [41]. In this approach, a full version of the new release of the SaaS systems would be deployed into a commercial cloud facility, as per the production version of the SaaS systems, but it would then be exercised and assessed against the generated workloads [42]. If the system has an acceptable performance, it is then released as the production system. This approach has the advantage that the full system in all its richness is performance tested, but it lacks the controllability and repeatability available through emulation-based testing.

- **Ongoing Monitoring**: In this approach, the system's performance behavior is been monitored in operation applying dynamical analysis and self-test techniques [43]. While run-time monitoring has been in use for decades, but there is a renewed interest in this approach because of the increasing complexity and ubiquitous nature of modern software systems [44]. Due to the fact that ongoing monitoring enforces an overhead required by testing and monitoring instrumentation, it may not be suitable for applications with limited resources, therefore, it is very important to determine where and when to observe the system's behavior for a meaningful and effective results. In this approach, the observed behavior is compared to the expected behavior to detect performance variations early enough to add/remove resources if possible. Hand in hand with monitoring, control theory techniques such as feedback loops has received considerable attention particularly in self-adaptive software systems and for performance control but typically the proposed techniques lack generalization

and tend to be ad hoc and suitable for specific types of problems. Moreover, to develop a theoretically sound controller for a software system, it's necessary to have mathematical models of the system's dynamics which are inherently complex models, however, many software engineers don't have the necessary background to create these models [45]. While analytical abstractions of software systems, e.g. [46; 47] can be used for this purpose, they're not enough to bridge the gap between software models and dynamical models for full control synthesis. More importantly, to be able to derive and trust those mathematical models, the underlying system has to be ergodic to assure that there is a path from the current state to the desired state.

The methodologies discussed above primarily focus on the run-time performance testing of software systems, where such testing is usually conducted after the completion of standard functional testing, such as unit testing, integration testing, systems tests, and acceptance testing. Performance testing, as an additional testing regime, has gained immense significance as studies have shown that, for example, web users have increasingly low tolerances for unexpected response time delays, as do users of video stream and on-line gaming services and platforms [48–50].

Typically, a combination of the above approaches may be applied to any given software systems or software intensive system, depending on the available testing resources and time frames. However, deployed systems tend to evolve away from their original software architect descriptions due to feature creep [51], changes required to resolve system bugs, and the impacts of real-world constraints such as limited development and testing time, deployment costs, etc.[52]. Consequently, run-time performance issues often go unnoticed until the system becomes fully operational, as a full-scale production system. Identifying and then resolving critical performance issues at this late stage in the system's deployment can be exceedingly costly as, generally, major performance improvements will need to wait until the system's next major release cycle. Unfortunately, the challenging nature of run-time performance prediction means that this next major release may also introduce its own set of excepted performance limitations and critical issues.

Besides the limitations already mentioned for software systems of any size, the run-time performance prediction of LDSS holds another level of challenges and is proven to be unreliable [14; 4; 53–59]. This is primarily due to the heterogeneous and elastic nature of cloud environments which results in variation in critical components affecting performance measurements of a distributed system such as data bandwidth

variation, overheads of managing distributed data centers, several layers of virtualization which leads to a more complex resource sharing schemes, heterogeneous workload, system size variability, migrations and more importantly lost data due to system overloads. All these variations together has a combined and non-trivial effects on performance prediction and in fact leads to high performance variability in commercial clouds as investigated in [53] more than a decade ago and observed in many recent works, e.g. [14; 54; 55]. As another instance, in the study conducted by [4], the author measured a cloud-based software system's performance under the identical workload distribution and cloud deployment configurations but observed different response time distributions. They concluded their system's performance is highly variable and dependent on initial conditions such as the underlying cloud servers utilization, the network conditions, and random variations in the arrivals of user sessions along with the timing of requests. In [56], authors observed that for similar use-cases and experiments, there exists a variation in public clouds' Infrastructure as a Service (IaaS) performance which made them motivated to investigate the patterns underlying performance variations of IaaS-based public cloud systems. There are several earlier studies as well in the literature such as [57–59] which show the cloud applications performance can vary dramatically, applying identical configurations from cloud's user side, due to the changes in cloud resource management parameters which are usually unknown to the cloud users. The traditional cloud resource management techniques, usually results in either over-provisioning which leads to lose benefits for service providers while wasting a vast amount of energy in data centers as explained by Alibaba cloud [60] or overloading the servers which eventually leads to violating Service Level Agreement (SLA) terms with customers, e.g. the AWS outage incident happened in 2021 [61]. As another example, as Kubernetes cluster gets larger, the performance deteriorates and the cold start delay can get unacceptably long. Cold start can take up to 10-15 seconds (and in some cases, a minute or more)[62]. In this case, developers tend to keep at least one pod replica alive and never scale to zero to defy the pod cold start delay. This approach and similar ones not only waste cloud resources but also increase the cost for cloud customers.

Therefore, it is no wonder run-time performance prediction in cloud environment has attracted considerable research attention and various techniques have been introduced or applied to solve the problem including control theory [63–68], machine learning methods [69–76] and analytical approaches such as applying Layered Queuing Network (LQN) models [77–79]. Typically, as per numerous published works, e.g.

[77–79], the run-time behaviours of these LQNs are then analysed via the mathematics of Markov process or stochastic Petri-nets or through simulation based approaches. Such works have shown impressive results in controlled simulations, such as in [80] where simulated cloud performance levels are shown to be predictable three significant figures for skew and kurtosis. Moreover, there are plenty of research been done in analyzing and modeling the effects of virtualization on performance prediction such as ones introduced in [78; 81–83]. There also exists performance prediction studies that consider applying containers instead of VMs for the sake of virtualization and resource sharing [79; 84–86] and as a result, studies to compare performance of the two, e.g.[87; 88]. Likewise, a large body of performance prediction research exists with respect to cloud management and resource consolidation techniques in virtualized environments [64; 89–92]. Other researchers suggest applying game theory as an optimization method for resource allocation to make the performance predictable [93–96]. Despite the vast amount of research and work done in this field, it is obvious that there is a gap between academia results and the observed reality in industry as explained earlier [14; 4; 53–59] regarding performance prediction and performance variation especially for larger scale software systems which brings the following questions to the mind:

- What is the core difference between academic research and real-world systems which results in the observed gap?
- Why are some software systems perfectly predictable under test and in the simulation but exhibit unpredictable behaviours in production environments?

As a matter of fact, in most academic studies with promising results, many real-world workload traffic characteristics, such as self-similarity, long-range dependence, and heavy-tailed distributions, are not considered due to the loss of analytical tractability [77–80]. Moreover, the complexity of academically simulated and analyzed systems are generally several orders of magnitude smaller in scale than commercial LDSS [64; 89–92]. The latter can easily cost into the low millions to build, run concurrently on thousands of interacting VMs, have hundreds of thousand a year in cloud deployment costs, and servicing active user bases numbering in the millions to hundreds of millions, placing them well beyond the contexts of academic research.

As well explained by Linda Northrop and her colleagues in their report on ultra-large scale systems[97], scale changes everything and that there exist fundamental gaps which are strategic not tactical in current understanding of software and software

development at the scale. They explain how these gaps are unlikely to be addressed by incremental research within established categories and the need for a breakthrough research which brings broad new conception of both the nature of such systems and new ideas for how to design, develop, control, monitor and assess them. In the next chapters, we explain how we tackle the existing gap, providing a novel strategic approach to the analyses of software-centric systems based on a solid mathematical foundation.

1.3 Formal Problem Definition

The problem which this dissertation is about to investigate is the lack of a solid and reliable mathematical model foundation which is able to explain the behaviour of today's complex software systems and also be able to fill the existing gap between the performance models and QoS management techniques in academia and industry. Therefore, this dissertation focuses on assessing and assuring **BET-compliance** conditions for arbitrary scale software systems within normal operational condition, i.e., excluding the impacts of potential attacks.

We define a predictable software system as one in which its future behaviors and their likelihoods can be learned through the collection and analyses of historical run-time data from the system's operation or from the operation of a sufficiently similar system. More formally, within the dissertation, statistical run-time predictability is defined as the ability to determine that any given run-time performance measure of interest $x(t)$ e.g. response time, throughput, etc., will fall within a set of upper and lower bounds, denoted $B_{lower}(t)$ and $B^{upper}(t)$, with probability $(1-\epsilon)$ over some future time period, $t \in (\ 0, T^{+}]$, based on the analysis of the past history of a set of performance measures $\boldsymbol{x}(t)$ collected from the same system for some historical time period $t \in [T^{-}, 0)$, where $t = 0$ denotes the current time as demonstrated in Figure 1.1 and equation 1.1.

$$\hat{P}_x[\ x(t) \in [B_{lower}(t), B^{upper}(t)]] \geq (1-\epsilon) \tag{1.1}$$

We focus on when the tightest bound of Equation 1.1 can be determined in non-trivial cases for software systems considering current bounds in commercial cloud's SLA is too loose [60; 98]. This leads to develop the understandings of what formal software engineering issues lead to a loss of run-time predictability, and how these

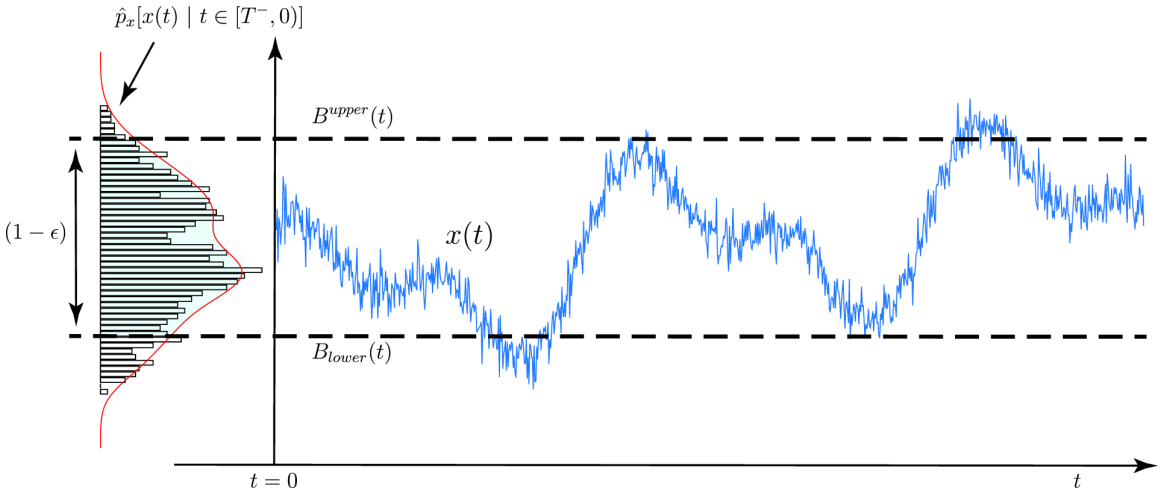


Figure 1.1: Statistical run-time behavior of any performance measure of interest $x(t)$

identified issues can be addressed. More generally, $x(t)$ may be a collection of run-time performance measures, i.e., $x(t)$ becomes $\boldsymbol{x}(t)$, and $B_{lower}(t)$ and $B^{upper}(t)$ may be constants, denoting standard limit or range checking [99] similar to Figure 1.1, or analytical time dependent bounds that are created as functions of other run-time performance measures as depicted in Figure 1.2 and Equation 1.2:

$$\begin{aligned} B_{lower}(t) &= F[\boldsymbol{y}(t)] \\ B^{upper}(t) &= G[\boldsymbol{y}(t)] \end{aligned} \tag{1.2}$$

where $F[.]$ and $G[.]$ are known functions that are constructed (or learned) through analysis of a system's past run-time data, where the historical data can be denoted as:

$$\mathcal{D}_{sys}[T^-] = \{\boldsymbol{x}(t) \cup \boldsymbol{y}(t) \mid t \in [T^-, 0)\} \tag{1.3}$$

This form of software system performance predictability is inspired by the bound checking and analytical fault detection approaches used within standard industry-scale operational engineering plant Fault Detection and Identification (FDI) methodologies. As such, it is of particular importance within the emerging mission critical software-centric systems arising across areas including smart cities, smart grids, autonomous vehicle and vehicular networks, i.e., systems with strong human and environmental health and safety concerns.

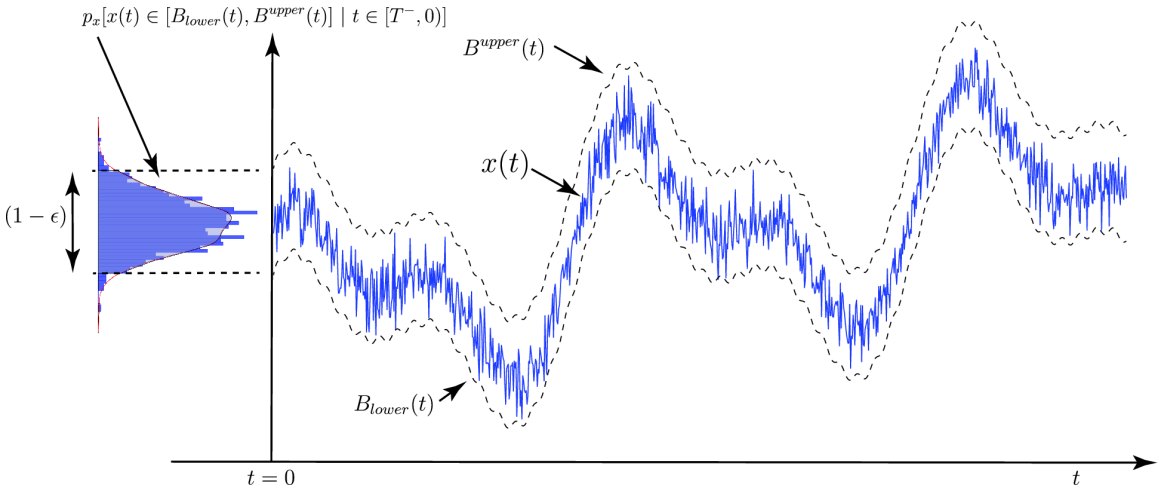


Figure 1.2: Analytical time dependent bounds created as functions of other run-time performance measures as shown in Equation 1.2

Traditionally, run-time software system predictability has been serviced through analysing formal analytical models of the system, e.g., via queuing networks [26; 100–106] and Markovian analyses [26; 104; 107–109] against prescribed incoming on/off workload models [110–116]. But, modern software systems are now commonly several orders of magnitude larger in scale and complexity than the prior generations of systems that gave rise to these classical run-time performance assessment approaches. More particularly, from a queuing network perspective, modern cloud-deployed software systems can require hundreds, to thousands, to tens of thousands of computers to run. Therefore, these systems can involve tens of thousands to hundreds of thousands (or more) of active queues within a sufficiently representative LQN model. Modern real-world cloud-deployed software-centric systems tend to be significantly larger than those studied and simulated within the academic literature [79; 83; 117]. Each queue within a LQN model can be assumed to have a small but non-zero queue-drop rate, denoted as $\zeta > 0$. As all real system queues must be of finite length, if a system's representative LQN model composed of K queues, then the likelihood of a queue-drop from any queue in the full running system is given by $1 - [1 - \zeta]^K$. For a small-scale system to function efficiently, this probability must approach zero, though, for a sufficiently large-scale systems, this probability quickly approaches one even for very small queue-drop probabilities as shown in Figure 1.3, denoting that large-scale operational software systems must always be expected to drop some important information.

This dropped information is then "recovered" via modern system's use of reliable protocols and fault recovery methodologies [118], where this dissertation shows these

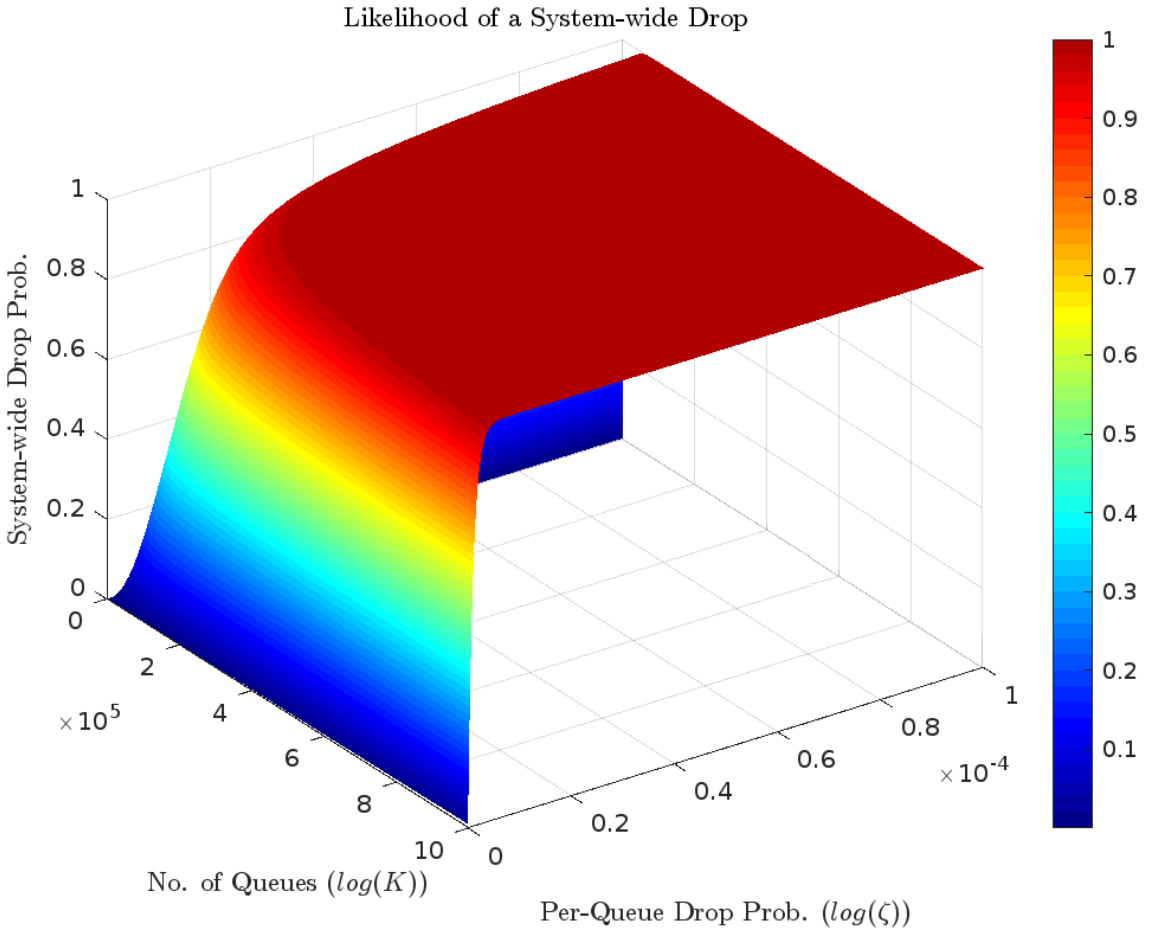


Figure 1.3: Queue-drop probability in any of the system's queues based on ζ and the number of queues in the system (K)

effects then directly impact the resulting system's statistical run-time performance predictability. This provides a core insight as to why modern industry-scale software systems tend to lack sufficient performance predictability, therefore this dissertation formalizes mentioned observation through applying Dynamical System Theory (DST) to an extended version of Maurer's Turing-reducible computer model, denoted as the Extended Maurer Model (EMM) [5], to show that the above queue drop events lead to the production of wandering sets of non-zero measure within the σ-finite measure space the EMM describes. This in turn is shown to lead to a direct loss of statistical predictability, of the form described above, due to the resulting violations of DST ergodic theory requirements and, particularly, the requirements detailed within BET [33], i.e., that such a systems statistical behaviours will change both over time and across different instantiations of the system.

More precisely, this PhD research addresses the following research challenges:

- Will a particular software system behave predictably when processing "typical" workloads? Why?
- When is the performance of a software system statistically predictable (or not predictable)?
- What formal assumptions and constraints are sufficient and/or necessary for predictable behaviours to be guaranteed?
- Can predictable performance measures coexist with unpredictable measures for the same system? And if so why?
- Based on the above issues, do software engineering design approaches exist that can give rise to performance predictability improvements?

Formally, it is shown that the above issues and concern relate to when a software system does (or does not) exhibit BET compliant behaviours. Hence, a core focus of the conducted research is applying the DST concepts underlying BET to derive the necessary and sufficient conditions for it to hold within software systems of arbitrary scales and complexities.

1.4 Assumptions

Section 1.4 outlines the assumptions exist in our research. First of all, we are investigating the run-time performance of an **operational** system in a **production** environment that has already undergone standard software testing, including unit, integration, and system testing, among others. Although our theory works for software systems of any size from embedded systems to edge computing in the clouds, but in this thesis, we are specifically aim the non-linear time-variant systems with stochastic sampling, otherwise the system would be simple and deterministic. Moreover, we're interested in cases where the Law of Large Numbers (LLN), Central Limit Theorem (CLT) and memory-less workloads do not hold. These statistical concepts that are commonly used to make predictions are not applicable when dealing with complex systems with non-linear behaviors. We assume that the cloud is in its day-to-day operational state, meaning it is functioning normally and not facing

any abnormal downtime due to power outages or cyber attacks, among others. Additionally, we assume that there are concurrent clients and shared resources, which may impact the system's performance. We also acknowledge that processing occurs on real machines with limitations, which may lead to high utilization, queue drops, and other issues. We assume that the systems we are investigating are not starved or overloaded(saturated), meaning they are functioning within their capacity. We are only considering causal systems, which means that we cannot have knowledge of the future. Causal systems are those that produce an effect only after an input or action occurs, and thus, it is impossible to know the outcome of a system before the input or action occurs. When measuring performance, we assume that measurable features are any measures that can be measured within a σ-finite measure space. The performance prediction involves learning from past behavior can be done with any technique from traditional analytic methods to more advanced AI and machine learning techniques. We are dealing with a more fundamental issue and that is if the system is predictable at all and if yes, then we can apply any method for the prediction process itself. In other words, we look for predict-ability and not optimal prediction. Lastly, while LQN can model a universal system, our research does not focus on universal systems, but rather dealing with Turing reducible machines and languages.

1.5 Contributions

This dissertation develops a new and novel approach to the analysis of software-centric system run-time performance predictability centred in the mature mathematics of DST and BET. It is shown that the developed approach is scale-agnostic, applying equally well to large- and small-scale systems, while also enabling direct insights into the set of defined software engineering design rules that must be followed if a system is to support run-time performance predictability. To our knowledge, the set of derived software engineering design rules required if run-time performance predictability is to be achievable has not be previously denoted within the literature, nor have such rules been set within formal and strong mathematical foundations. Hence, these represent the principal novel contributions of this dissertation:

- Applying DST to derive the necessary and sufficient conditions for software systems of arbitrary scales and complexities to exhibit BET-compliant run-time performance measures.

- From the derived DST conditions developing a set of pragmatically and easily implementable software engineering design rules required to assess when and why BET-compliance is lost (or gained) for the run-time behaviours exhibited by arbitrary scale and complexity software systems.
- Conducting detailed cloud-deployed software system simulations, via a constructed packet-level simulation environment, to validate the developed theory, understandings, and insights as to how run-time behavioural predictability is gained (or lost) within modern VM and container deployed SaaS applications.
- Identify how and why emerging industry software deployment trends and regimes align with the conducted DST-based BET analyses, so as to provided the mathematical foundations underpinning these evolving industry trends.

Each of the above contributions are novel contributions with research to the software engineering field and discipline. More particularly, a general proof of ergodicity and BET-compliance within arbitrary scale and complexity software systems has not been previously provided within the literature nor has the reduction of these mathematical requirements into a minimal set of easily and pragmatically implementable software engineering design rules.

1.6 Dissertation Organization

As a first step to tackle the ergodicity in software systems problem, a general scale-agnostic approach for modeling software-centric systems inclusive of their statistical predictability is developed which will be explained in detail in Chapters 3 and 4. This approach first models the software system in the application layer as a LQN in Chapter 3 and then applies a DST's model of the performance measures, in which all stochastic processes are known to possess equivalent dynamical system model [33] to explain the behavior of the corresponding LQN model. Moreover, the prior PhD work of Elgamal [5] on the EMM is used to provide a scale-agnostic bridge between Turing models and DST and, therefore, to provide a general modeling approach for software systems of any size in terms of DST. This provided a path by which to apply known ergodic theory results for sigma-finite measure spaces to the analysis of BET-compliance and performance predictability issues within software-centric systems particularly the most complex ones in cloud-deployed distributed systems. This is a novel and largely

unexplored approach that offers the promise of placing such issues on the very solid mathematical foundations of ergodic theory.

As a second step, this theory-based research is then validated using simulation on OMNeT++ platform which is a modular, component-based C++ simulation library and framework, primarily for building network simulators [31]. The simulation provides a scalable, repeatable, and accurate environment to verify theory-based analysis of performance predictability. Our simulation has been structured on top of the latest version of OMNeT++ in a way to support full-stack full-packet level simulation which provides vetting the stochastic properties dependent to load size. Moreover, the simulation applies standard QNM to model distributed systems at the virtual machines and containers level. Therefore, designing different scenarios is as easy as manipulating queuing network parameters in configuration files. The configuration file is being used as an input to the simulation tool. Basically, in simulation-based research we tested and verified results of theory-based analysis of performance predictability of cloud deployed distributed systems. Different scenarios designed in a way that applications execute either in virtual machines or containers or both at the same time which introduce additional levels of complexity. Using the various simulation scenarios, we investigated unpredictable systems and explored where and when QoS predictability may lost based on theoretical results. We then applied and evaluated the proposed techniques to bring the system back to a predictable state.The result of simulation experiments explained and discussed in Chapter 6.

The remainder of this dissertation is structured as follows:

- Chapter 2: Defines the scope of the research and provides a rigorous review of relevant literature on software systems performance modeling, particularly focusing on large-scale distributed software systems (LDSS). This chapter explores the existing studies on the performance variability of cloud applications and highlights the challenges faced by both industry and academia in ensuring predictable run-time behavior of software systems. It also reviews cloud service models, including the emerging trends of Containers as a Service (CaaS) and Functions as a Service (FaaS), which are increasingly used for deploying microservices in the cloud. Furthermore, the chapter examines existing scheduling techniques in cloud environments, workload modeling, and prediction methods, alongside the current challenges in these areas. . Importantly, it introduces recent works considering the importance of ergodicity in software systems, noting that while earlier efforts in computer science attempted to prove ergodicity, con-

temporary research often assumes it without formal verification. This review underscores the gap between theoretical models and real-world implementations, setting the stage for the novel approaches and analyses presented in the later chapters.

- Chapter 3: Introduces a unified layered queuing network (LQN) model tailored for software-centric systems, ranging from embedded systems to large-scale distributed systems with serverless microservice architectures. This model serves as the basis for understanding the run-time performance behavior of software applications. The chapter details the construction of LQN models, illustrating how they capture the hierarchical nature of modern software deployments, including the on/off scheduling effects of virtualization. By mapping the software system's application layer into an LQN framework, this chapter sets the stage for integrating dynamical systems theory (DST) to analyze performance measures. This approach enables a more accurate representation of the system's behavior under various workloads and operational conditions, forming the foundation for the DST analyses and BET-compliance considerations discussed in the subsequent chapters.

- Chapter 4: Provides one of our main contributions and establishes the foundational bridge between queuing network (QN) models and dynamical systems theory (DST) through the Extended Maurer Model (EMM). This chapter develops a scale-agnostic approach for modeling software-centric systems, using QN models to represent the software system's behavior and integrating EMM to map these behaviors into a DST context. The integration is crucial for supporting subsequent BET-compliance analyses. The EMM, based on Maurer's Turing-reducible model, allows for the analysis within σ-finite measure spaces, which is essential for dealing with the complexities of modern, large-scale software systems. Additionally, this chapter extends traditional QN models to incorporate stochastic On/Off switching at the per-queue level, accurately reflecting real-world cloud operations where VMs are time-sliced into available CPU resources. This extension, which addresses a critical gap in existing literature, is vital for modeling the unpredictable nature of cloud environments and supports the detailed theoretical analyses presented in Chapter 5.

- Chapter 5: Conducts a full theoretical analysis to establish the necessary and sufficient conditions for achieving BET-compliance in software-centric systems, leveraging the QN and EMM-based approach developed in Chapter 4. Utilizing Peter Walters' Ergodic Equivalency Theorem 3, it was demonstrated that Markovian approaches rely on the recurrence requirement of Statement 3, whereas Statements 2 and 4 provide alternative conditions. By applying these insights, the research identified that queue drop events and the recovery actions of reliable protocols produce wandering sets of non-zero measure, while time-variant stochastic effects from fair OS scheduling and resource contention lead to loss of measure invariance. These findings underscore that achieving BET-compliance in software systems involves addressing these specific factors, which are often overlooked in traditional Markovian analyses. The chapter's theorems and proofs offer a novel perspective on ensuring predictability in large-scale, complex software deployments.

- Chapter 6: Focuses on validating the results and insights from Chapter 5 using a Monte Carlo simulation framework developed with OMNet++ and INET. This framework models modern container and VM-based cloud computing deployments, incorporating Poisson and bursty workloads, fair and RTOS OS scheduling, and background server VM processes. The simulation provides a scalable and accurate environment for verifying performance predictability, structured to support full-stack, full-packet level simulation. By applying standard QNM models, various scenarios were designed where applications execute in virtual machines, containers, or both, introducing additional complexity. The simulations tested and verified the theoretical analysis results, exploring where and when QoS predictability might be lost and applying techniques to restore predictability. The analysis included 32 experimental scenarios with Monte Carlo ensembles of 100 runs each, and the results aligned with the theoretical expectations from Chapter 5, confirming the insights derived from the DST analyses.

- Chapter 7: Applies the simulation approach from Chapter 6 to explore and quantify the impacts of two modern software system deployment techniques: Apache Spark and Storm-like auto-scaling, and Xen-like hypervisor soft RTOS scheduling, compared to classical load balancing solutions. The focus is not merely on performance improvement but on how these techniques enhance BET-compliance and run-time performance predictability. Consequently, the

study in Chapter 7 places these industry-developed methodologies onto a solid theoretical foundation, demonstrating that while they contribute to improved performance predictability, they are not adequate to ensure complete BET-compliance on their own.

- Chapter 8: Concludes the dissertation by providing the four Software Engineering Design Rules derived from formal analyses of Chapter 5 and outlines future work in this field.

1.7 Summary

In summary, the focus of this PhD work is to address the identified software predictability challenges and to answer the mentioned question of if and why the techniques used for performance management are working by exploring where, when, and why performance predictability is lost (or gained) in software systems of any size and especially in the attractive case of cloud deployed distributed systems. To answer the mentioned questions, we provide a framework rooted in a solid mathematical foundation which is based on understanding of the nature and underlying dynamics of software system. Based on that reliable mathematical foundation, we are able to analyze the software behaviour from the smallest in embedded systems to the most complex ones expanded across several clouds, we can also explain the underlying reasons of observing unpredictable/predictable behaviour in software systems or any other complex software system and provide reliable and promising solutions to fix the unpredictability problem. We can then evaluate the reliability of the existing industry solutions based on the proposed model. Moreover, the result of this research enables us to answer the following important questions about performance measures:

- Whether observable past performance datasets are informative as to any given system's likely future behaviours? (Exploring the stationarity characteristic)
- Whether performance data collected from one deployed instance of any given system are informative as to the behaviours likely to be expressed by any other instance of the same (or sufficiently similar) system? (Exploring the ergodicity characteristic)

Fundamentally, when "informative" is defined in terms of Shannon information, then the above questions can be recast as simply exploring degree of statistical pre-

dictability the observed performance measures possess. Clearly, achieving reasonable levels of such predictability has direct implications on the rate at which service level agreements (SLAs) and obligations (Service Level Obligations (SLO)s) can be met, as well as on wider issues such as how to cloud resource consolidation management problems [90] should best be addressed, etc. To examine our proposed theories and techniques, we are going to apply an Omnet-based simulation tool which we introduced for the first time in this research. We propose and describe the aforementioned simulator in Chapter 6 along with related experiments.

Chapter 2

Related Work

The motivation for this research is ignited when we were investigating if it is possible at all to predict the performance of LDSS in a cloud environment with a multi-tenancy regime in place. As we progressed in the research, reviewing and evaluating the existing work, we discovered there is a missing piece in the literature which can cause a well-tested, well-documented software to behave unpredictably, particularly in multi-tenancy environments. That resulted in expanding our research to dig deeper and investigate the run-time behavior predictability of software systems of any size. In this chapter, we will define the scope of our current research and summarize and discuss related works and studies on software performance prediction, modeling distributed software systems, and cloud applications. We also discuss various approaches taken in an attempt to predict the performance of a cloud application from workload modeling to applying control theory, game theory and machine learning to solve the problem. Moreover, we review experimental studies that have observed significant performance variation in software applications deployed in commercial clouds. These studies suggest that the performance of cloud applications can be highly variable in real-world environments, and further research is needed to understand and address this issue.

In Section 2.1, we define and explore LDSS and its characteristics as well as its role in the modern world. In Section 2.2, we review cloud service models and the new trends of Container as a Service (CaaS) and Function as a Service (FaaS), which are both being applied for deploying micro-services in the cloud. In Section 2.3, we examine the existing scheduling techniques applied in cloud environments. In Section 2.4, workload modeling and prediction are explored along with discussing the existing challenges and issues in this field. In Section 2.5, we discuss the analytical models

for performance evaluation. In Section 2.6, while we discuss existing approaches for performance prediction and resource management in cloud environments, we highlight the predictability issues observed in practice and reported by researchers. In Section 2.7, we introduce recent works that consider the importance of ergodicity in software systems. Section 2.8 summarizes the whole chapter and draws conclusions.

2.1 Large-scale Distributed Software Systems

In this section, we clarify the definition of large-scale distributed software systems we are dealing with and modeling in this PhD research. There are many different definitions of distributed systems in the literature, depending on the specific domain and goals of a particular system. However, one of the most comprehensive definitions is the one proposed by Steen and Tanenbaum, which includes all types of distributed systems. In this definition, a distributed system is a collection of autonomous computing elements that work together in a way that appears to its users as a single entity. These computing elements are called nodes, are independent of each other communicating by exchanging messages through a network which can be wired, wireless or a combination of both [119].

Compared to centralized systems in which a single computer handles all the processing, data storage and communication, distributed systems especially the larger ones are way more complex and need to address issues such as synchronization and coordination within a distributed system, managing membership and organization of nodes, authentication and admission control in closed groups, security mechanisms, message passing protocols, fault tolerance and reliability issues, etc. Also, a distributed system has a dynamic nature considering nodes can join and leave, with the topology and performance of the underlying network almost continuously changing.

The size of a distributed system may vary from a handful of devices, to millions of computers. In this work, we are more interested in the case of large-scale distributed software systems which have at least tens of thousands of nodes scattered across one or more commercial clouds with unreliable communication channel such as the Internet, consist of several heterogeneous interacting software components exist on multiple computing and storage nodes, deals with huge amount of data, has tens of thousands to millions of users and has specific performance requirements such as throughput, response time, latency and availability. Design, development and maintaining such systems to meet the mentioned requirements is extremely challenging due to the

significantly big size and dynamic nature of these systems. Moreover, it is often the case the components of the system are running on multi-tenant virtualized resources with unknown continuously changing scheduling and prioritization policies [120; 121].

Modern examples of large-scale distributed software systems such as e-commerce websites, financial markets applications, social media applications or MMOG must service thousands or millions of requests concurrently and most of them are now shifting toward the use of commodity clouds [122]. These applications are often very sensitive to user experience and failing to meet performance requirements leads to unfavorable reviews and media coverage. However, software, hardware and human failures are not exceptions but the norm in these systems [97]. In the next section, we will review the various service models available in clouds and how they are being applied in the deployment of LDSS in cloud environments.

2.2 Cloud Service Models

Cloud computing, often referred to as simply "the cloud," is the delivery of on-demand computing services via the Internet on a pay-as-you-go basis. The National Institute of Standards and Technology (NIST) has denoted five essential characteristics for cloud models, namely as per the definitions of [123] which are repeated for convenience:

- On-demand self-service."A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider".
- Broad network access. "Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations)".
- Resource pooling. "The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth."

- Rapid elasticity. "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time."
- Measured service. "Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service."

2.2.1 Traditional cloud service models

The aforementioned characteristics have made the (IaaS), PaaS, and SaaS hierarchy highly attractive to increasingly wide cross-sections of industry. As expected by Armbrust et al. in [124], cloud computing has transformed large parts of the IT industry by making larger-scale distributed software systems easier to cost-effectively build, deploy, and manage. The relation between the IaaS, PaaS, and SaaS cloud offering models is illustrated in Figure 2.1.

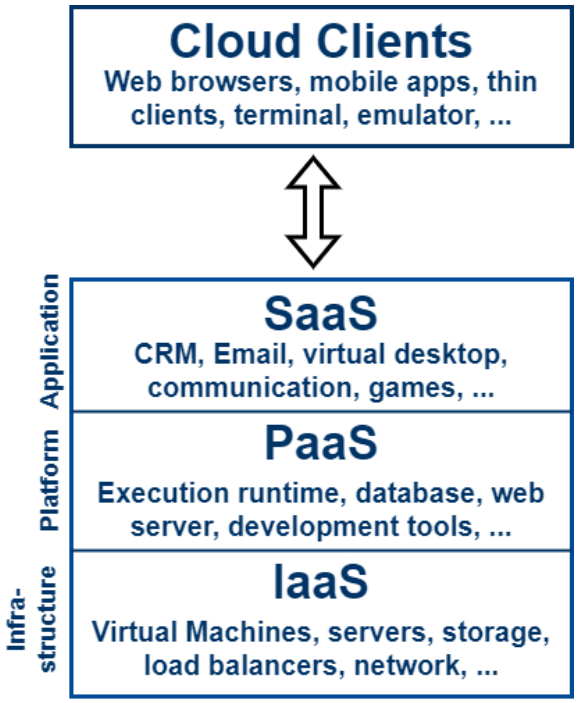


Figure 2.1: Traditional IaaS, PaaS, and SaaS cloud hierarchy service model.

IaaS is the lowest layer of cloud services as it provides cloud clients (or users) with direct access into the VMs they have leased. The user is then fully responsible for managing all aspects of these machines, from installing and patching OSes or Operating System (OS) images, to installing and configuring all of the per-VM software components and applications required, to then configuring, deploying, and managing all but the most basic network layers of their full cloud-deployed distributed software systems.

Platform as a Service (PaaS) removes many of these lower level cloud deployment details by packaging them up into easier to use and access software platforms. Common examples of commercial PaaS offerings include Google App Engine (GAE) [125] and AWS Elastic Beanstalk [39]. A drawback with PaaS approaches is that they are typically limited to only being able to support software applications that are written against the PaaS's specific services. For example, GAE does not support the full Java Runtime Environment (JRE) libraries. This restricts GAE deployed applications to those that only use GAE supported JRE subset [126]. Such PaaS service decisions also tend to evolve over time, typically necessitating on-going reworking of PaaS deployed applications by PaaS user communities. SaaS is the highest tier layer and typically designed to allow the easy turn-key incorporation of new cloud-based services into an existing software service or solution. From the perspectives of this research, SaaS is of less interest as many key QoS/QoE software engineering design and architecture decisions would have already been incorporated into offered SaaS solutions. Hence, the core research issues of interest in this work arise in the IaaS and PaaS cloud layers and in the design, implementation, and re-engineering of SaaS offerings.

2.2.2 Container as a Service Model

In general, IaaS as the lowest-level and least restrictive cloud offering, requires highly skilled and knowledgeable people to make effective use of it. PaaS is easy to use, even by lower skilled people, but it tends to abstract out and effectively fix or substantially restrict many of the critical software engineering architecture design decisions. Such issues, combined with a strong industry-wide desired not to have to custom tailor solutions against each cloud providers' offering, has lead to the introduction and rapid adoption of CaaS solutions [127], of which the Docker platform current dominates[6]. CaaS sits between the IaaS and PaaS layers and provides developers with richer tool-

sets to deploy and maintain their applications compared to PaaS while reducing the detail-oriented burdens IaaS requires. This concept illustrated in Figure 2.2.

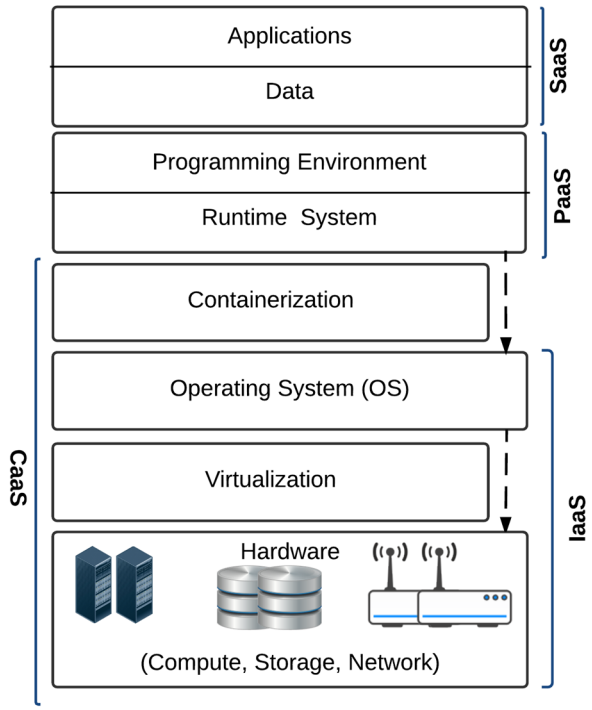


Figure 2.2: The CaaS cloud service model [1].

CaaS solutions have made inter-cloud migrations and non-cloud to cloud migrations trivial by packaging entire application and reducing the need for any re-engineering efforts, therefore, also substantial reducing the associated cost of changing deployment regimes. Since containers provide process isolation, it makes it easy to break apart and run applications as independent components called microservices which is a very popular architecture among LDSS. Moreover, deploying microservices using container makes scaling much easier and faster. Also, one can fit multiple, lightweight containers on a single virtual machine, which increase the efficiency and usage density of resources. These are good reasons for rapidly increasing popularity of CaaS in clouds. According to Flexera in their most recent report on "The State of the Cloud" published in 2022 [128], container use becomes increasingly mainstream. Similarly, Amazon reported 150% year over year growth in its container services usage in 2022 [129]. The following are some of the CaaS offerings from major cloud providers:

- AWS Fargate
- Google Cloud Run
- IBM Cloud Container based on Red Hat OpenShift or Kuberneres
- Microsoft Azure Container Instances

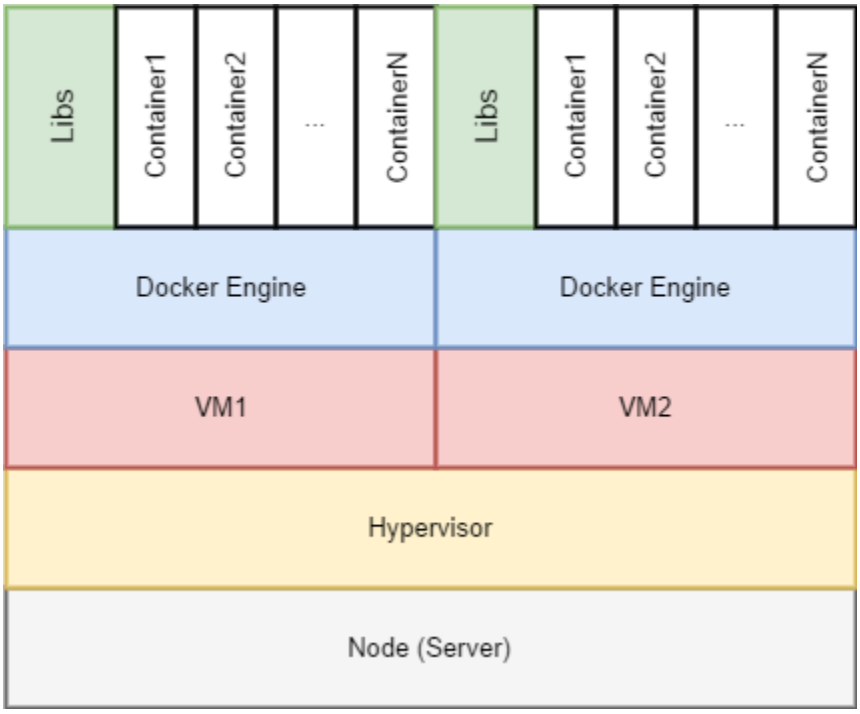


Figure 2.3: An example of CaaS deployment model on top of IaaS.

Figure 2.3 provides an illustration of a Docker-based CaaS deployment approach on top of an IaaS offering. An alternative CaaS deployment approach, which considered less secure and, generally, is only available for semi trusted users [130], is shown in Figure 2.4. In this CaaS deployment model, all Containers on a given physical server share the same kernel host removing the need for intermediate hypervisor monitoring layers. This improves performance by reducing overheads, but as the cost of setting aside VM sandboxing.

This research will focus on the CaaS deployment model on top of IaaS as this is rapidly coming to dominate how commercial cloud platforms are used[131], setting aside SaaS offerings which themselves are increasing based on CaaS deployment regimes. Also, when we mention containers, we mostly talk about Docker containers, however, in 2015, Docker and other industry leaders came together to form the

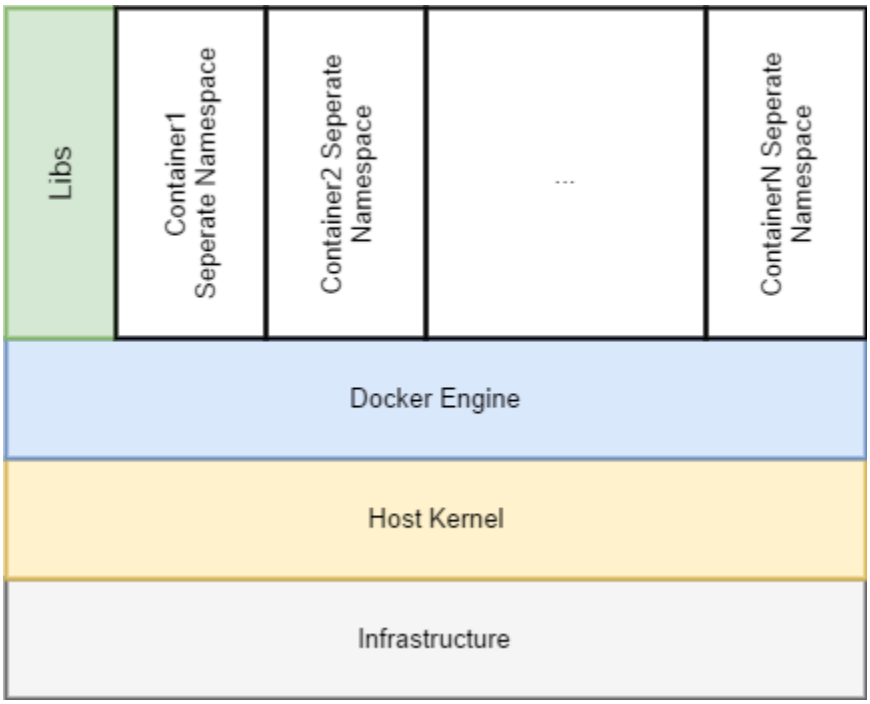


Figure 2.4: CaaS architecture for Semi-trusted users.

Open Container Initiative (OCI). The purpose of the OCI is to create standards for container formats and runtimes [132]. There exist a large body of literature studying performance prediction of applications in cloud environments but none of them considers involving containers in the clouds. Fortunately, our theory is applicable to any size and any deployment. Using Docker Containers, migration between different cloud providers becomes a simple task and clients can distribute their applications easily on several clouds for efficiency. For example, one could deploy compute-intensive services on Google cloud and deploy I/O intensive containers on Amazon cloud AWS. Investigating and modeling the behavior of such a hybrid and complex application deployments is necessary in performance evaluation and prediction. The need for QoS prediction is more obvious in the hybrid case where different chosen clouds and services have different characteristics and costs. It is good to mention that using containers for deployment of distributed software systems in cloud environments provides the following advantages:

- Simplifies and increases the convenience of deploying, running, and testing distributed applications for cloud environments
- Enables platform-independence

- Enables easy tuning of the deployed scale to match workload demands
- Enables easy immigration between cloud providers' cloud offerings

These advantages though come at the cost of introducing an extra abstraction layer between the distributed systems application-layer code and the hardware on which it ultimately runs. This cost can be reduced by providing direct non-VM based container support within a cloud platform, but this in turn generally runs against the industry-standard VM-based cloud billing and resource leasing models.

2.2.3 Function as a Service (FaaS)

Containers are at the core of enabling another relatively new cloud service model which allows a function gets executed as a response to an event. FaaS is a serverless technology which compared to CaaS is more abstract and agile but less configurable. In this model, developers can break their application down into functions and scale each function independently. The focus will be on writing functions with one purpose similar to a microservice but a microservice can contain several related functions. This concept illustrated in Figure 2.5. These functions get triggered by an event and behind the scene, cloud providers execute the triggered function inside a container with required run-time configurations.

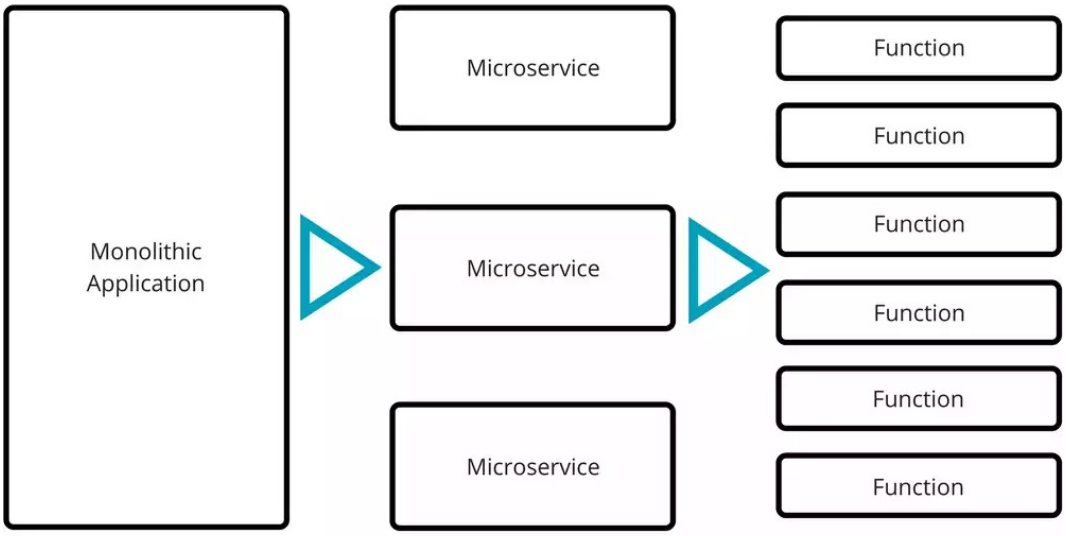


Figure 2.5: FaaS compared to monolithic and microservices architecture.

Moreover, FaaS is cost effective because one's account gets charged only when a function gets executed i.e an event happened in the system. Obviously, FaaS is highly scalable and is a good choice for implementing high-volume parallel workloads such as web application with millions of users, MMOG, Monte Carlo simulations, parallel data processing, healthcare workloads such as DNA sequencing, etc. Figure 2.6 shows how FaaS related to other cloud services. Most modern systems are using a combination of these cloud services.

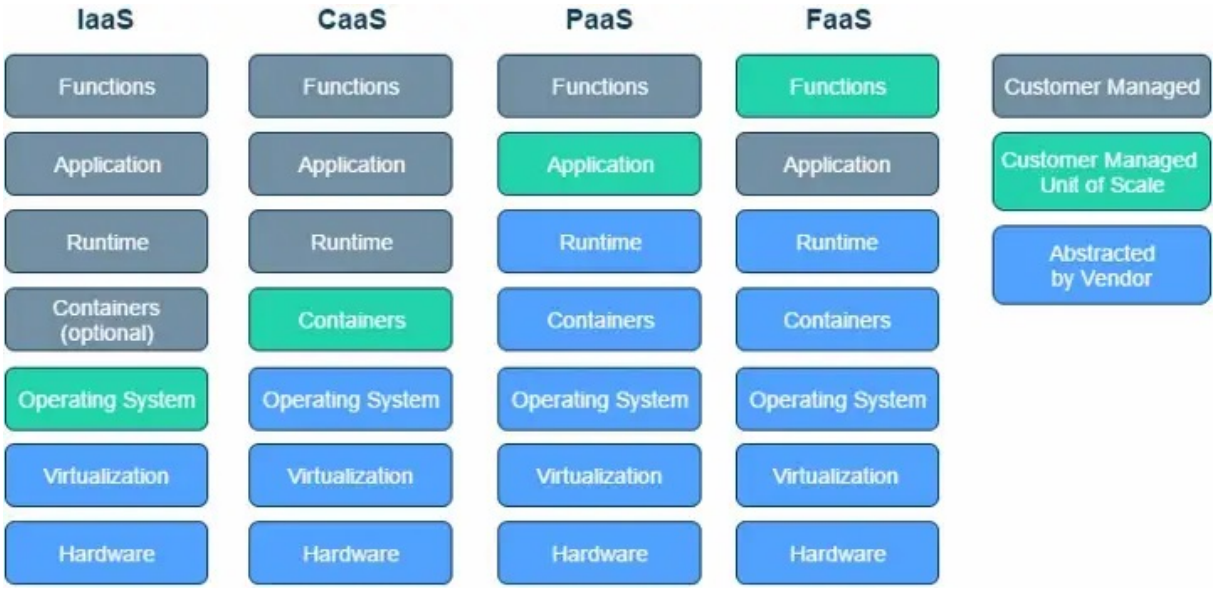


Figure 2.6: Cloud service models comparison[2].

All major public cloud providers offer at least one version of event-driven computing services. The following are some of those offerings:

- Alibaba Cloud Function Compute
- AWS Lambda
- Google Cloud Functions
- IBM Cloud Functions based on Apache OpenWhisk
- Microsoft Azure Functions
- Oracle Cloud Functions

Figure 2.7 shows the comparison between the most popular FaaS providers.

Comparison Parameters	AWS Lambda	Azure Functions	Google Functions
Supported Languages	Node.js, Python, Java, C#, Go	Node.js, Python, Java, C#, PHP	Node.js, Python
Trigger Type	AWS Services + API Gateway	Azure Services + HTTP requests	Cloud Pub/Sub + Cloud Storage+ HTTP requests + Directly Triggers
Concurrent Execution	1000 parallel executions	No Limits	No Limit for HTTP execution
Deployment	edit code inline, Zip Upload	git, dropbox, Visual Studio, Kudu console OneDrive, Zip upload,	Zip Upload, CLI, inline web editor, Cloud Source Repositories
Scalability & Availability	Automatic Scaling	Manual or Automatic	Automatic Scaling
Logging and Monitoring	CloudWatch Logs and X-Ray	App Insights	StackDriver
Maximum Number of Function	No Limit	No Limit	1000 functions per project
Maximum Execution Timer per request	900 seconds	Default 300 seconds	Default 30 seconds

Figure 2.7: FaaS providers comparison[133].

In conclusion, modern application architectures require a more elastic infrastructure with a continuum of services from CaaS to FaaS. CaaS and FaaS make it easy to use the infrastructure resources more efficiently while also giving the flexibility to run many different workloads. Both serverless technologies make the life of developers easier by taking away the complexities of managing the underlying deployment infrastructure such as virtual machines or servers and also relaxing the need for capacity management from developer side, leaving all the heavy lifting jobs for cloud providers. At the same time, cloud customers expect to see similar performance behaviour and QoS measures for different instances of the same application running in the cloud but apparently this is not the case based on several recent studies [4; 53; 57–59]. In the following section, we will examine the most widely used methods and technologies for managing cloud resources in order to address performance issues.

2.3 Cloud Resource Management

We are discussing cloud resource management in relation to other topics here because it plays a crucial role in the performance prediction of cloud applications as we will illustrate in Chapter 3. In cloud computing, resource management is the process of allocating, organizing and controlling the resources available to the cloud. These resources may include hardware resources such as servers and storage devices, software resources such as operating systems, virtual machines, containers and other applications, and networking resources such as bandwidth and virtual private networks. A successful resource management policy or technique, on one hand seeks to meet the performance objectives of the users' applications and on the other hand satisfies the performance requirements of the cloud provider (i.e., data center). The objectives of the cloud providers focus on efficient and effective cloud resource usage within the constraints ofSLA. The objectives of the cloud users are centered around preserving desired application performance such as response time, throughput and availability with cost-effective scaling in place. Often, these objectives come with constraints regarding resource dedication to meet non-functional requirements relating to, for example, security or regulatory compliance [134].

There are typically three terms commonly used in the literature when discussing cloud resource management:

- **Resource allocation** is the process of reservation of or assigning a quantity of resources such as computing, storage, and networking resources, for a tenant's use to ensure guaranteed performance.

- **Resource provisioning** refers to the process of making the allocated resources available for use. It involves the selection, deployment, and run-time management and configuration of allocated resources so that they are ready for use.

- **Resource scheduling** which interchangeably used by terms resource mapping, resource brokering, load balancing, etc. is the process of deciding when and how resources should be used and results in the precise assignment of provisioned resources (e.g. CPU, memory, storage, network) to tasks within the workload.

Resource allocation and provisioning is more of a high-level task versus the scheduling which is more kind of a hypervisor's operating system task. For example, it's the resource allocation duty to decide based on the incoming workload how many other

VMs or containers are needed to start and it's the resource provisioning duty to decide on which physical server to start the VM or on which VM to start the container but then the scheduling algorithm is the one who decides which VM or the container gets to access the physical resources on a particular server as shown in Figure 2.8.

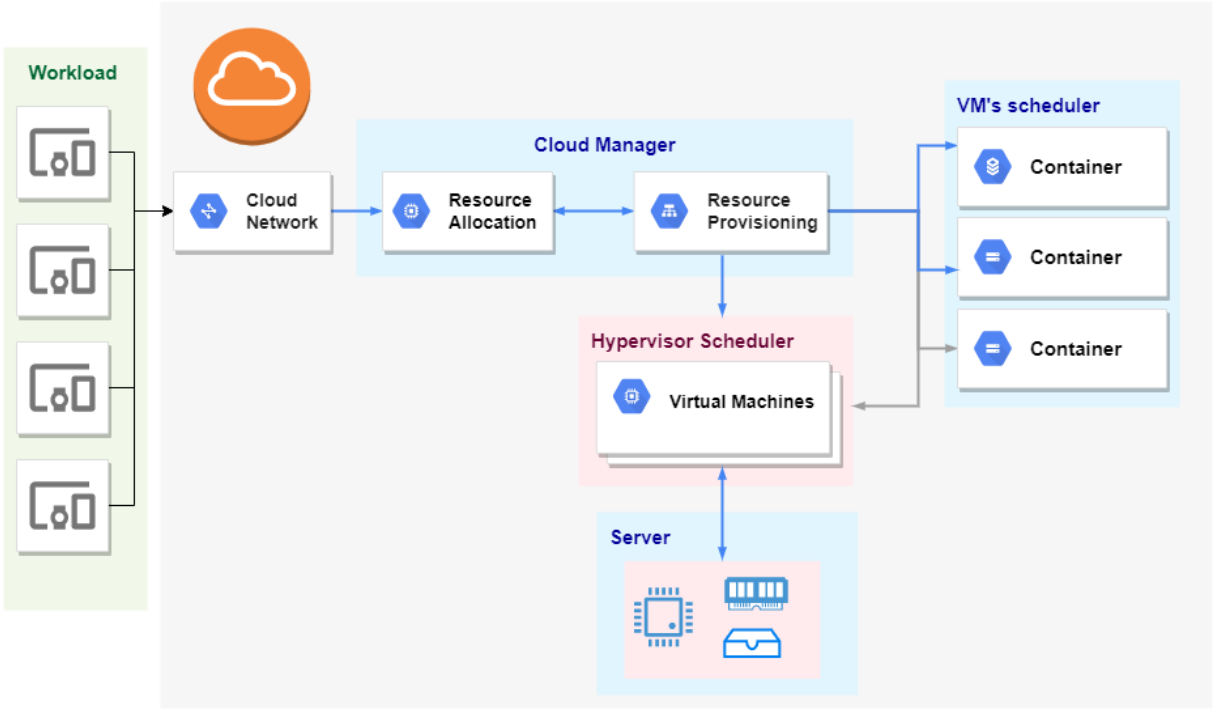


Figure 2.8: Cloud resource management.

Resource allocation is also responsible to meet the user's need based upon the quality of service parameters (QoS), SLA negotiations and match the resources to the upcoming workloads [120]. The basic idea behind the resource allocation and provisioning respectively is to detect the best amount of required resources and select the best candidate servers providing those resources based upon upcoming application demands. The ultimate goal of resource allocation is to optimize the algorithm in a way that minimize the number of resources needed to serve the application demand to maintain a desirable level of service quality and performance which translates to minimum execution time with maximum throughput. Resource provisioning algorithms map upcoming request with running virtual machines or containers so that user gets the services with minimum cost and time while service provider get the maximum profit without affecting the violation of SLA [135]. According to [120], there are three major resource allocation approaches available for the cloud users:

- **On-demand**: Resources allocated based on the application needs and clients billed accordingly.
- **Advanced reservation**: Clients reserve a specific cloud resource (e.g. EC2) for a specific time period in advance.
- **Spot instances**: If there are any unused cloud resources at a given time, clients can bid on them for an immediate use with a lower price compared to advanced and on-demand allocation.

Considering aforementioned general approaches, there exists different techniques for resource allocation, including static and dynamic allocation, quality of service (QoS) based allocation, and capacity planning. Below is a brief overview of each:

- **Static resource allocation**: This technique involves allocating a fixed amount of resources to an application or service. It's under the category of advanced reservations and although this technique is simple to implement, but may not be as efficient as other techniques because it does not take into account the changing needs of the application or service [136].
- **Resource reservation**: This is a technique for advanced reservation approach and involves reserving resources in advance for a particular application or service. This can be useful for ensuring that resources are available when they are needed, but may be less efficient than other techniques because it may result in the allocation of unnecessary resources and over-provisioning.
- **Dynamic resource allocation**: This method is under the category of on-demand allocation approach and involves allocating resources to an application or service as needed, based on factors such as the workload or the performance of the application. Dynamic resource allocation can be more efficient than static allocation because it allows resources to be used more flexibly. A good review of different algorithms in dynamic resource allocation can be find in [137]. Here are two most popular methods of dynamic resource allocation:
  - **Capacity planning**: It involves forecasting the future resource needs of an application or service, and allocating resources accordingly. This can help to ensure that resources are available when they are needed, but may be less efficient than other techniques because it requires forecasting and

may result in the allocation of unnecessary resources or under-provision. Researchers applied different techniques to tackle this issue, for example the authors in [138] developed a new Swarm Intelligence Based Prediction Approach (SIBPA) to predict the resource requirements.

– **QoS based resource allocation**: This involves allocating resources to applications or services based on the level of service that they require. For example, an application that needs a high level of performance may be allocated more resources than an application that is less critical. In a recent work, Haji et al in [139] explore different algorithms proposed for QoS aware resource allocation.

There are different techniques and algorithms introduced in the literature to optimize cloud resource allocation and provision methods mentioned before from optimization and game theory to the use of machine learning. For example, [140] presents a technique for optimizing cloud resource allocation using online learning, which involves adapting resource allocation decisions based on real-time data. The authors demonstrate the effectiveness of the technique through simulations and show that it can improve resource utilization.

No matter which approach, technique or algorithm used for cloud resource management, there is no work in the literature that formally studies the roots and fundamentals of how these techniques can affect the predictability of software systems deployed in cloud.

2.4 Workload modeling

Characterization of the software system's workloads and understanding of their properties and behavior is essential for an effective resource allocation, performance predictability and QoS management to achieve the desired service levels. As stated by Feitelson [111] understanding the workload is more important than scheduling algorithms and other optimizations. Essentially, the workload model should have statistical characteristics which are representative of the real one. As Ferrari states in one of the early researches about workload characterization:

> The performance of computer systems cannot be seriously and meaningfully measured unless the workload the system is dealing with during the

> measurement sessions is carefully selected. ... The main objective in selecting a workload should always be representativeness, i.e., the resulting workload should faithfully represent the real workload [141].

According to [112], the term workload refers to all types of inputs including inputs from applications, services, transactions, data transfers, etc. which are submitted to an e-infrastructure. In the framework of cloud computing, aforementioned inputs usually come from online interactions of the users with web-based services hosted in the cloud or from the jobs processed in batch mode. It is important to note that typical cloud workloads almost never refer to hard real-time applications. While the general principles applied to parallel and distributed systems workloads are still valid, the cloud workloads characterization has its own challenges because it consists of a collection of many diverse applications and services, each characterized by its own performance and resource requirements and by constraints specified in the form of SLA [142]. The aim of this section is to survey the most relevant studies in the area of cloud workload characterization and identify and discuss the main issues in this area.

Despite their importance, the characterization of cloud workloads has received a little attention in the literature and mostly at the level of the VMs without taking into consideration the features of the individual workload components running on the VMs themselves. Therefore, here we have to discuss general workload modeling alongside distributed systems and cloud workload modeling. The general problem of computer systems workload characterization and prediction has been widely studied in the past (e.g. [114; 143; 144]). Calzarossa et al. [114] started a comprehensive survey on workload characterization in early '90s and then revisited it in 2000 [145] to talk about issues and methodologies and in 2016 [113] to include growing domains such as social networks, video servicing, mobile devices and cloud computing.

In the late '80s and early '90s, network engineers observed that some properties of computer systems and networks are related to distributions with very long tails [146–148] and then in the mid '90s, researchers started to consider heavy-tailed distributions in computer systems [115; 149–152]. In early 2000s, Crovella in [153] explores the effects of heavy-tailed distributions on performance evaluation of computer systems in general. In heavy-tailed distributions, tails are not exponentially bounded, in contrast to traditional distributions (e.g., Gaussian, Exponential, Poisson) whose tails decline exponentially. Among the most famous heavy-tailed distributions are Pareto [3], Cauchy, and t-distribution.

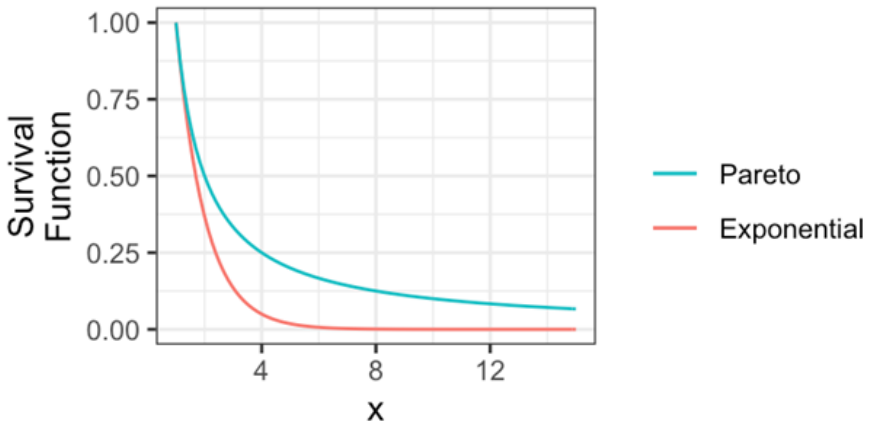


Figure 2.9: The comparison between a heavy-tailed and exponential distribution

Figure 2.9 compares Pareto and exponential distribution and shows that the exponential survival function becomes essentially zero very quickly, whereas there is still lots of probability well into the tail of the Pareto distribution. If a distribution's tail is "too heavy", then its mean will not exist. For example, the above Pareto distribution has no mean.

Some of the early works done toward characterizing the workload in a cloud environment presented in [116; 154–156]. However, these studies are concerned with statistically understanding and reproducing computing tasks (e.g., MapReduce tasks) scheduled on a cloud. The work on capacity management and VM placement typically employs some workload modeling and prediction techniques [157–162]. These methods depend on the statistics of individual workload time series to predict future resource demand.

2.4.1 Workload Classification

Here, we analyze the behavioral characteristics of cloud workloads (i.e., their qualitative and quantitative attributes) to identify some broad categories in the literature, specified in terms of arrival rate, processing models and resource utilization.

One of the most basic criteria to classify workloads is the ***arrival rate pattern***. We can explain the workload characteristics based on the different patterns observed in the coming requests. Before we continue, it is good to remind that the arrival rate is the number of request arrivals per unit of time and is often used as a parameter of a

queue in queuing network analysis. Followings are the most common workload/traffic classification based on the arrival rate [3]:

- The "On and Off" pattern reflects applications that have a short period of activity and can be switched off after producing a result. The advantage of workloads with this pattern for shared resources environments such as clouds is that cloud resource manager might schedule it with other "On and Off" workloads with similar resource requirements at different times. The time spent between the ON and OFF states, commonly referred to as the transition time, is expected to follow an exponential distribution [3]. This model is originally used for broadband networks, but a modified version is used for modeling distributed web applications where users have an idle time between sending requests to the system [143]. This model is also quite useful to model application queues when queues have two ON and OFF states based on scheduling scheme. We can assume a system is in ON state when events in the queue can be processed and in OFF state when the queue is paused by scheduler. The On/OFF model is used to simulate workloads discussed in [143; 163; 164].

- The "Growing Fast" pattern is the scenario of start-ups with workload growing exponentially along with resource requirements. The effect of fast-growing workload on QoS predictability is one of the issues investigated in this thesis.

- When a workload contains "Unpredictable Bursts", it is not known when the peak demand will be. This type of workload makes the consistency of the scheduling system vulnerable especially if several cloud customers experience it at the same time. It may affect the resource manager and QoS to the point that customers mistakenly taken long response times as denial of service attack. It actually is a failing with respect to the QoS. In this thesis, using Extended Maurer Model (EMM) to model dynamical systems, we explore which parameters of the system is responsible for these unpredictable failures and what is the solution to this situation.

- "Predictable Burst" pattern unlike previous pattern has similar characteristic to the "On and Off" pattern and it has been known in advance when demand will be high. Scheduling this type of traffic is not a difficult problem and is usually modeled by Poisson on/off traffic model which based on the Poisson process, a statistical process that describes the random arrival of events over

time. The on periods represent the time when traffic is present, and the off periods represent the time when traffic is absent. The rate of arrival of packets or messages during the on periods is assumed to follow a Poisson distribution, and the duration of the on and off periods are also assumed to be exponentially distributed.

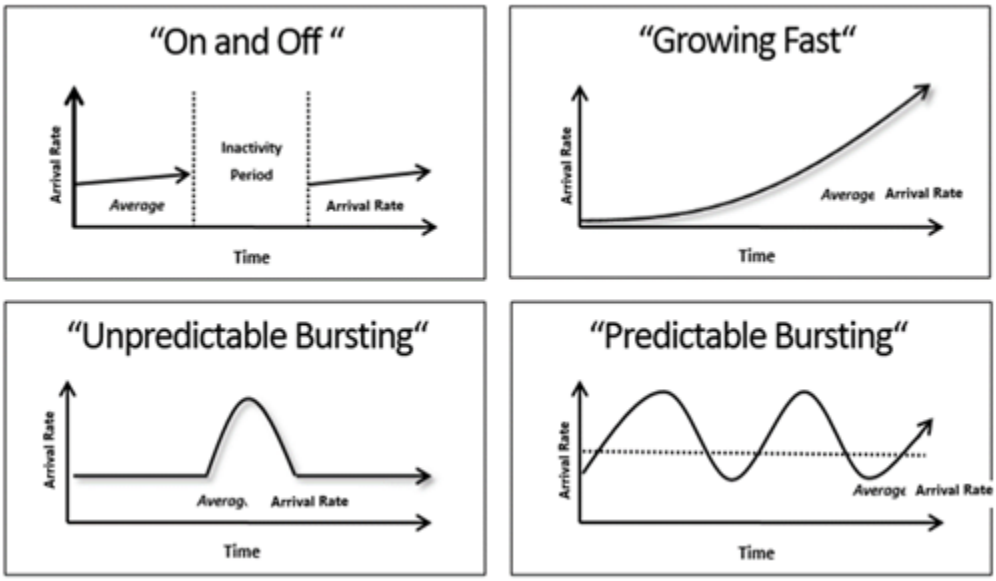


Figure 2.10: Basic building blocks of workload incoming patterns [3].

Among all different possible workloads, fast growing and unpredictable bursting workloads are the most problematic ones for effective resource management of any system. However, the arrival rate of cloud workload can not be simply identified as one of those main categories. The cloud workload has a dynamic nature due to the collection of many diverse applications running at the same time and consists of streams of requests submitted at unpredictable times. The intensity of the workload can grow or shrink depending on the types of applications running in the cloud at the same time and their workload characteristics.

In practice, the co-location of heterogeneous workloads concurrently executed on a given virtual machine (VM) can be partially responsible for performance unpredictability issues due to incompatible temporal patterns of the resource usage as stated in[58]. The workload classification based on ***processing model*** is a high-level categorization refers to interactive (online) or batch (offline) processing of the requests. An interactive workload typically consists of short-lived processing tasks

submitted by a variable number of concurrent users, whereas a batch workload consists of resource intensive long-lived tasks. These categories are characterized by diverse behaviors and performance requirements. Likewise, they have different impacts on management policies such as resource scheduling, VM placement, VM migration, etc. [112].

Another important criterion which can be used for workload characteristics classification is the ***run-time behavior***. Related to this, there are qualitative attributes such as priority and termination status as well as quantitative attributes such as workload intensity and resource utilization patterns. These characteristics classify workloads as either compute intensive or I/O intensive. On the other hand, the bandwidth utilization classifies them as elastic or bandwidth sensitive workloads. These attributes can directly affect on resource consolidation strategies. Especially, the quantitative attributes should be analyzed carefully to avoid over-provisioning or under-provisioning of the resources.

In [165], workload intensity is quantified in terms of task submission rate and clustering is applied to highlight variability in the submission rate across groups of tasks. Other papers model the workload intensity by means of stochastic processes. It has been shown that simple Poisson processes generating independent identically distributed inter-arrival times are not suited to represent real cloud workloads [166].

2.4.2 Workload Prediction

Researchers tackled the workload prediction problem from different perspectives. Some are focusing on time series analysis[162; 167–169] and others applying machine learning techniques [170–172; 71]. From time series perspective, Roy et al. [168] applied Auto-regressive Moving Average model to estimate the incoming workload of a system for future time periods. The model considers the workload patterns up to the current time period. Similarly, Calheiros et al. [169] using an auto-regressive integrated moving average model to address the prediction of workloads characterized by a seasonal behavior. The model is based on historical workload data and get updated real-time by applying feedback from latest observed loads. The predicted load is then used to dynamically provision cloud resources. On the other hand of time series perspective, there are experimental works such as one explained in [167] trying to characterize and predict the workload of VMs in the cloud. They searched data traces obtained from a private cloud for repeatable workload patterns by exploring

cross-VM workload correlations resulted from the dependencies among applications running on different VMs. They use Hidden Markov Model (HMM) to explore the temporal correlations in workload pattern changes which assumes that the workloads are ergodic and stationary.

Applying machine learning techniques, Islam et al. [170] developed prediction models to be used for adaptive resource provisioning in the cloud. They applied machine learning algorithms, particularly error correction neural network and linear regression along with the sliding window technique to show the superior prediction of the neural network models with an optimal window size (i.e., the Mean Absolute Percentage Error is 0.195 compared to 0.364 for linear regression). Yang et al. in [171] integrate a workload predictor based on a simple linear regression model to propose a cost-aware auto-scaling approach for on-demand resource allocation. On the other hand, in [172], Chang et al. apply recurrent neural network model for workload forecasting.

In the framework of data center workload prediction, Kumar and Singh [71] propose a prediction approach based on three-layer neural network trained using a self adaptive differential evolution algorithm able to explore the solution space in multiple directions. A cyclic window learning approach is applied in [173] to predict the probability distribution parameters of the number of task arrivals to a data center during every predetermined period, while a Seasonal ARIMA model is proposed in [174]. Liu et al. [175] propose an adaptive approach for workload forecasting based on the idea that workloads exhibit different change patterns. According to this approach, different models – based on linear regression and support vector machine – are associated with different workload classes. The Mean Relative Percentage Error of this approach (i.e., 0.4677) is lower than errors obtained with other methods (e.g., linear regression, ARIMA). It is important to note that all the works applying machine learning techniques are basically assuming the systems under training is ergodic and stationary (at least weakly), otherwise it is not possible to learn from historical data to be able to predict the future behaviour.

A recent work [176] on modeling and predicting dynamics of heterogeneous workloads for cloud environments combines various workload characterization techniques to obtain an integrated approach for predicting workload access patterns. Those patterns then used for resource provisioning in cloud environments[72; 74; 73].

2.4.3 Workload Modeling Challenges

Workload scheduling corresponds to the mapping between tasks and VMs is a challenging problem in cloud environments because of the heterogeneity of workload characteristics. The problem of finding an optimal mapping is NP-complete [112] and with large number of VMs and tasks in cloud it becomes intractable with exact methods. For this reason, meta heuristics based on methods, such as neural networks, evolutionary algorithms, or set of rules, are being applied in solving optimization problems related to scheduling.

Another challenge in workload modeling is the observed self-similarity characteristics in the cloud environments workloads. In distributed systems, similar to most cloud applications, modeling workload using exponential distribution based on individual user's behaviour fails to accurately represent the real workload due to the fact that real web traffic shows self-similarity characteristics [149]. As discussed in [177], to simulate the self-similarity effects, one or both of the ON and OFF states in "On and Off" model, need to be drawn from a heavy-tailed distribution such as Pareto or Weibull. According to [149] self-similarity happens where bursts happen at many different time scales, and so appears similar to itself at different resolutions. To describe the time-varying behavior and self-similar effects, metrics, such as index of dispersion and coefficient of variation, are complemented with models based on two states Markovian arrival processes, parameterized with different levels of burstiness [178]. The two states represent the bursty and non-bursty request arrival processes, respectively as shown in Figure 2.11. Markovian arrival processes are integrated in [179] with analytical queuing models to predict system performance.

A different approach based on fractal techniques is proposed in [146; 180] for representing workload dynamics in terms of job arrivals. The arrival process is modeled using fractional-order differential equations with time-dependent parameters, whereas fitting is applied to identify statistical distributions for CPU and memory usage.

In general, a cloud cannot transform a non-stationary or non-ergodic workload into a stationary or ergodic one by conventional means. One simplistic approach is to discard the workload entirely and return a constant "error" message, effectively bypassing any execution of the workload. However, other strategies can also be employed, such as introducing controlled delays or buffering to smooth out fluctuations, applying statistical methods to approximate the workload's behavior over time, or partitioning the workload into smaller, more manageable tasks that can be processed

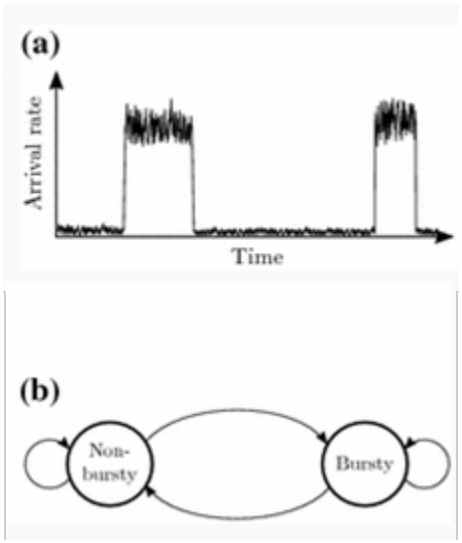


Figure 2.11: Example of concurrent non-bursty and bursty arrivals and (a) the corresponding 2-state Markovian arrival process (b).

independently in a stationary and ergodic manner. Each of these approaches involves trade-offs and may not fully address the underlying non-stationarity or non-ergodic behavior of the original workload. The literature review presented in this section, shows the importance of considering workload characteristics to effectively model the behavior of software systems with respect to performance and QoS metrics.

2.5 Analytical Performance Modeling

In this section, we discuss existing analytical models for performance evaluation of software-centric systems from a simple embedded system to complex distributed applications deployed in commercial cloud environments. Analysis technique is a popular tool in the area of performance prediction because it provides less expensive approach to study and estimate performance and QoS measures in planning and design stages of new services. In most of those studies, queuing theory, the classical approach for modeling computer systems, has been applied to evaluate service performance. There are different perspectives in how to model software systems and cloud environment with queuing networks which results in different approaches and techniques which we discuss here.

For example, queuing theory is used in [77; 80; 181] to obtain those indicators necessary to make online control decisions such as a method introduced in [182] to

apply Lyapunov optimization to design an online admission control and resource allocation algorithm in clouds. It basically works as a dynamic loadbalancer for all applications running in the cloud based on application throughput and energy consumption. There are different admission levels in clouds and although the main one is on VM level which decides if start a VM or not, the proposed approach in [182] admission control is on application request level.

In another work, an adaptive feedback control scheme alongside with a queue model of the application was employed in [66] and authors in [67] modeled cloud services using queuing theory and controlled them through adaptive proactive controllers that estimate whether services need some of the resources in the near future or not. The Markov decision process framework proposed in [183] modeled admission control in cloud, while approximate dynamic programming paradigm was utilized to devise optimized admission policies. There are other approaches to model cloud for resource management and performance analysis such as one based on gaming theory proposed in [93] and another one in [68] which employs network calculus to determine effective bandwidth for aggregate flow. Still, the most popular and accurate way to model clouds is queuing theory because it can model scaling behaviour as well as competition for limited resources.

In [100], the cloud is modeled as queuing network with two tandem servers as shown in Figure 2.12. The first server represents a web server, and the second server models the cloud service center, both modeled as M/M/1 queues. The percentile of response time to service requests was evaluated as the performance metric in this paper to study the relationship among the maximal number of customers, the minimal service resources, and the highest level of service performance. The proposed model with two tandem M/M/1 queues lacks the ability to represent the large number of interconnected servers and their collaboration affect on the performance and also lacks the ability to model virtualization and containerization in cloud computing.

As cloud computing became more popular, more studies started considering virtualization in their performance modeling. For example, Goswami et al. [78] modeled applications as queues and the allocated virtual machines as servers. The number of servers is variable and can be used to represent the elastic feature of the cloud by being dynamically adjusted based on the queue length. Similar to the previous model in [100] both request inter-arrival time and server service time are assumed to have an exponential distribution and so the model is an M/M/m/N queue with m servers and a finite buffer of size N. A steady queue size distribution state was obtained

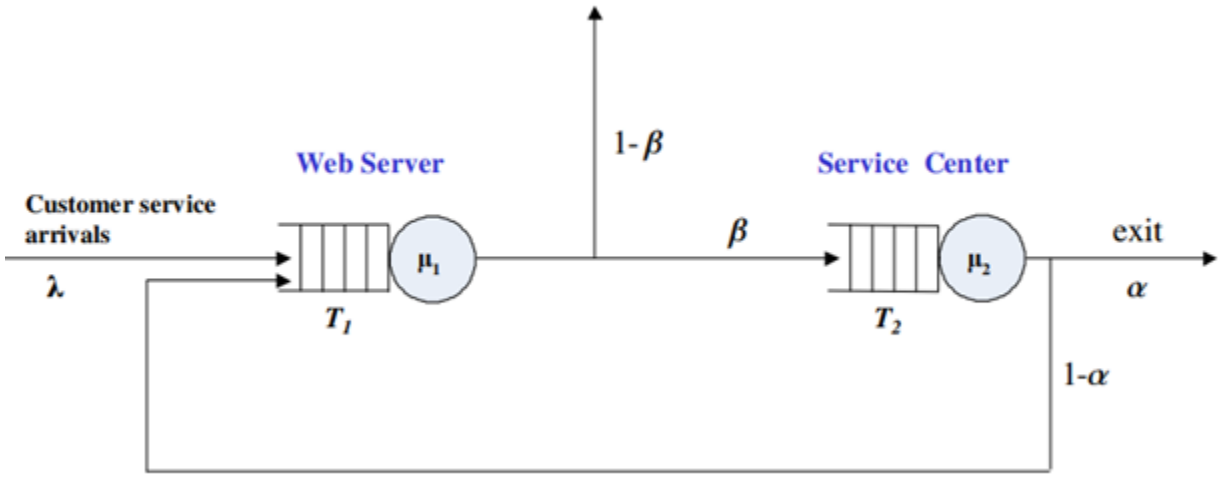


Figure 2.12: A queuing performance model for customer services in cloud computing [100].

using a recursive method and the service response time was evaluated as the main performance metric in this paper.

Liu et al. [81] addresses resource sharing among virtual machines and its affects on performance. They also studied the impact of various types of failures, such as VM failures, physical servers' failures, and network failure on the cloud performance. Each service request is assumed to comprise multiple sub-tasks, which will be assigned to different virtual machines that share the underlying physical infrastructure. The requests arrival and the service rates are both follow an exponential distribution. All the failure's rate modeled as a Poisson process. They showed that resources scheduling strategy of cloud centers have great influence on performance. Ellens et al. [82] developed an M/M/C/C queuing model for cloud computing centers with multiple priority classes for users with different Service Level Agreements (SLAs) as shown in Figure 2.13.

There are C servers in the model and the total system capacity is C which means there is no buffer before the servers. All servers are split into two categories: reserved servers that are assigned to process client requests by following priority scheduling, and shared servers that are used to serve the request from any client based on a FIFO policy. The model assumes that the service request arrival process to be a Poisson process and that the service time is exponentially distributed. The main performance

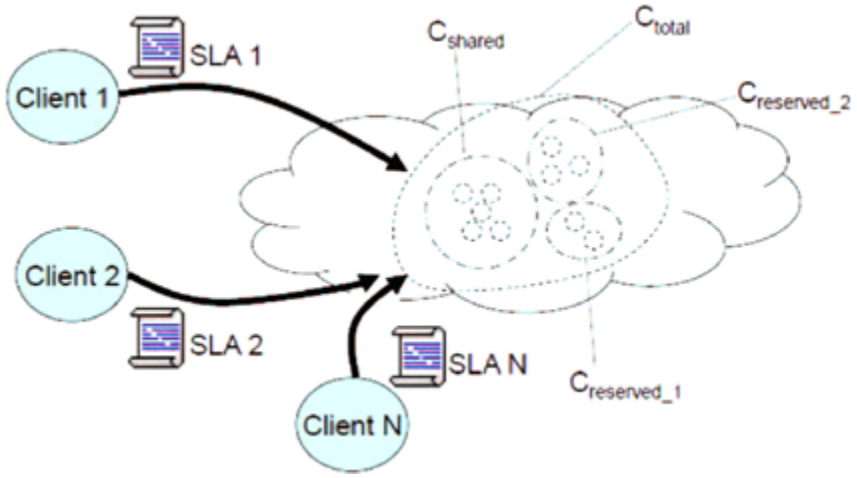


Figure 2.13: Performance of cloud computing centers with multiple priority classes [82].

criterion in this paper is the rejection probability for different customer classes, which can be analytically determined.

All the mentioned works assume the service time distributed exponentially. Although this assumption simplifies the modeling and analysis, does not precisely represent the practical service times in cloud infrastructures with heterogeneous implementations and technologies. For that reason, researchers in this area, turned their attention to general distribution as a more appropriate representative for modeling the service time of a cloud server. However, applying general service time distribution may lead to higher analysis complexity. For example, the solution for response time in M/G/m models and its variations cannot be obtained directly in a closed form and needs suitable approximations.

Khazaei et al. in[181] proposed an approximate analytical model where cloud server farms modeled as M/G/m/m+r queuing system. The arrival process of the task requests is assumed to be a Poisson process and the service policy is FCFS. The performance is being evaluated using a combination of a transformed-based analytical model and an approximate Markov chain model to obtain a complete probability distributions of service response time and the number of tasks in the system. The authors also discussed the immediate service probability and blocking probability and determined the buffer size required for keeping the blocking probability below a threshold. In this work, the burst arrivals of request have not been considered. To validate analytical solutions, a simulator based on Petri net has been developed and they concluded that in cloud centers with diverse services, there may exist relatively

long response times for some tasks compared to others or even there are tasks that could get blocked. This happens due to the observed heavy tail on the right-hand side of the response time distribution. They later extend their work to represent the effects of bursty arrival workloads of cloud infrastructures in service performance evaluation [77]. In their new research, Khazaei et al. modeled cloud computing centers as an M[x]/G/m/m + r queuing system. In such a model, the arrival service request process is assumed to be a sequence of super-tasks, each of which consists of a burst of tasks. The inter-arrival time of super-tasks is exponentially distributed and the service time of each task in a super-task has a general distribution. The system has m servers and a buffer size of r. A super-task will be rejected if there are not sufficient resources for the whole super-task.

Modeling and predicting cloud service performance requires appropriate models that cover a vast parameter space. A monolithic model may suffer from intractability and poor scalability due to the large number of parameters[184]. An approach to reducing the complexity of cloud service performance analysis is to divide the system model into sub-models and then obtain the overall solution by iteration over individual sub-model solutions. In [185], end-to-end cloud service provisioning is considered to have three main steps: resource provisioning decision, VM provisioning, and run-time execution. Compared to a single one level monolithic model, this analysis method is more tractable and scalable. This model has two disadvantages, first, the performance analysis is only applicable to service requests with a single task and second, the effect of virtualization in cloud infrastructures was not explicitly reflected in the analysis reported in [185]. [186] while applying the sub-model analysis idea explained earlier also considering the important features of cloud environments, such as batch arrival of tasks and resource virtualization. The authors developed sub-models for resource allocation and virtual machine provisioning in an interactive way such that output of one sub-model is the input of the other one and vice versa.

The sub-models implemented using an interactive Continuous Time Markov Chain (CTMC). The super task arrival process considered to be a Poisson process and will get serviced in a FIFO basis. The overall solutions for performance metrics such as task blocking probability and total waiting time incurred on user requests were obtained by iteration over individual sub-model solutions. In [187], the authors combined the system details, such as Physical Machine (PM) occupation/release, PM warm-up/cool-down, PM fail/recover, and job rejection, into a general model in order

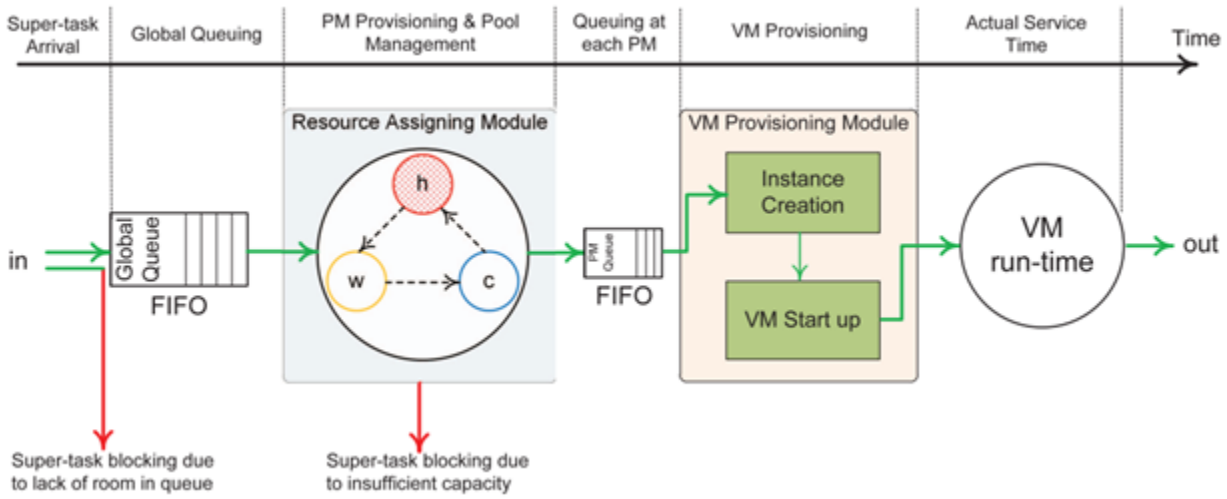


Figure 2.14: The steps of servicing and corresponding delays [186].

to consider all service provisioning details having simultaneous impacts in determining the overall QoS as shown in Figure 2.14.

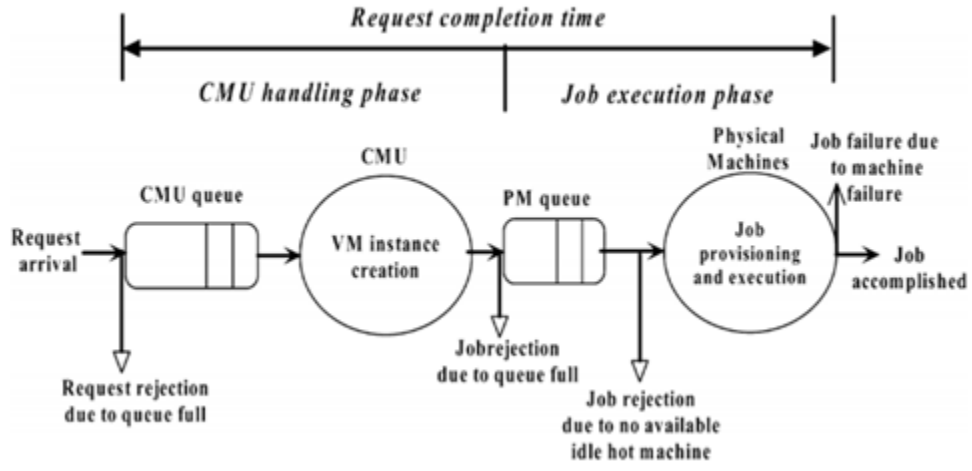


Figure 2.15: Request handling and provisioning steps [187]

This model captures system behaviors in a general state transition model and considers the inter-influence of these behaviors in deciding the final QoS. This research focuses on expected service completion time and rejection probability as QoS metrics. The state explosion issue of the Markov chains employed in the paper, results in a too complex model and limits the developed techniques only to be able to evaluate small scale cloud infrastructures. If we compare this combined model against the sub-model techniques proposed in [185; 186], we realize that there has to be a trade-off

between accuracy and complexity in developing analytical models for cloud service performance evaluation.

The literature review presented in this section, shows the importance of analytical modeling in performance and QoS analysis of cloud environments. The majority of analysis has been done by applying queuing network techniques, but cloud centers differ from traditional queuing systems in following manners:

- A cloud center can have a large number of server nodes, typically of the order of hundreds or thousands while traditional queuing analysis could become too complex to solve by considering systems of this size.
- As we mentioned earlier, service times must be modeled by a general, instead of a more convenient exponential probability distribution.
- Dynamic nature of cloud environments, diversity of user's requests and time dependency of load, makes it difficult for cloud centers to provide expected quality of service at widely varying loads. The literature review here has shown that it is very difficult to find a suitable trade-off between complexity and accuracy of analytical models to be able to show such a dynamic nature.

2.6 Performance predictability in cloud environments

Across the composite of all the deployment regimes, performance and QoS management remains an open industry as well as an academic research challenge which can be defined in terms of seeking to dynamically allocate resources to the cloud-deployed distributed software systems so as to guarantee service levels based on performance, availability, reliability, etc. [89]. As stated in [188], "QoS is fundamental for cloud users, who expect providers to deliver the advertised quality characteristics, and for cloud providers, who need to find the right trade-offs between QoS levels and operational costs". Any violation of service level agreements (SLAs) entails a loss for both cloud providers and cloud users [189]. Such QoS/QoE including performance measures are becoming increasingly important as more and more commercial services are transitioned to cloud deployments, as evidenced by the impacts of Amazon AWS recent system-level outage [190].

The cloud providers are applying the traditional approach to address such issues by simply over provisioning the resources. But this could be wasteful in terms of both resource and energy costs [191]. Moreover, cloud resource optimization still is an important open research problem that largely hinges on developing improved understandings of cloud performance predictability, inclusive of the characteristics that lead to reduced predictability. Though QoS properties have received considerable attention, particularly in the prior generations of network engineering research, the heterogeneity and resource isolation mechanisms of cloud platforms combined to significantly complicate cloud performance analysis, prediction, and assurance. This is leading researchers to explore approaches such as automated QoS management that is designed to leverage the high programmablity of hardware and software resources in the cloud [192]. The viability of such approaches though rests, again, on the degree that cloud computing environments support or can be designed to support performance predictability. This in turn rests on developing more detailed and formal understandings of performance predictability within cloud computing's contexts. A significant body of literature has sought to investigate various aspects of PaaS and SaaS performance, but less research has focused on CaaS dand FaaS deployment regimes. In general, the introduction of CaaS works to further complicate these prior PaaS and SaaS's performance models through its introduction of an additional layer of abstraction. Moreover, different applications have different QoS/QoE measures that are of interest, a rough classification of which has been propose in by Ghahramani et al. [193] and is depicted in Figure 2.16. According to [193], the most interested performance measures are response time, processing time, service throughput, data transfer rate and latency. All these can be calculated applying a proper QN model as discussed before and will be explained in detail in Chapter 3 but there is no overt reason to believe that all application-layer performance measures produced by a given instance of a given application would necessarily possess the same (or similar) degree of predictability. Moreover, different instances (or different cloud deployments) of the same application-layer CaaS or FaaS code-base may exhibit different degrees of performance predictability.

The current literature has largely does not provide a detailed exploration of such issues, even though industry is becoming well aware of the propensity of different cloud deployments of the same software application stack to produce widely different behaviors. As an example, following studies show the different aspects of unpredictable behavior of clouds:

- The same workload with the same cloud configuration for the application resulted in different response time distributions over several runs. O'Dwyer in [4] shows that performance of a system against the same scenario is highly variable and dependent on initial conditions of instances.

- The IaaS instances with exactly the same configuration behave different on different clouds. In [57], performance of scientific workload which are coming from very large systems and demanding high performance has been evaluated. They have done empirical evaluation of four public IaaS clouds and realized that the performance of instances using the exact configuration can vary due to the behavior of other collocated tenants. In a similar study in [56], authors motivated to explore performance predictability of IaaS clouds by observing variations in performance results of instances with same configuration in similar experiments.

- The same instance configuration on the same cloud shows different behavior with respect to performance. In [58] authors conclude that EC2 performance varies a lot and often falls into two bands having a large performance gap in-between.

- Existing QoS management techniques can lead to unpredictability. Cerotti et al. in [59] showed that flexible CPU allocation policy used by cloud providers will lead to performance unpredictability.

- Work of Lou et al. in characterizing microservice performance on Alibaba clusters [194] using a 2021 trace has revealed that microservice graphs are dynamic in run-time, most graphs are scattered to grow like a tree, and the size of call graphs follows a heavy-tail distribution which means there has to be variability in their performance.

All these works are a sign that researchers are becoming more aware of unpredictability in cloud environments but none of them could tell what the exact source of unpredictability in clouds is. As mentioned earlier, cloud performance predictability has direct impacts on cloud resource allocation, consolidation, and management strategies. A significant body of literature seeks to address dynamic resource allocation in virtualized environments e.g., [63; 64; 69; 195–199], where such research predates cloud computing's emergence. Common to these approaches is that applications of control theory methods to tune application performance in a fine-grained

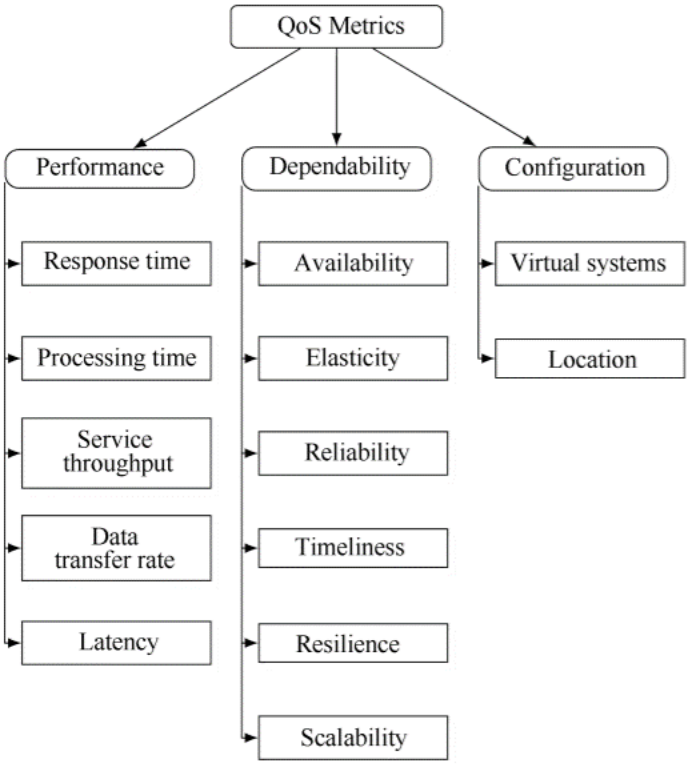


Figure 2.16: Classification of QoS metrics[193].

manner, as such these methods rely on workload predictions models in order to solve the main control theory problem. Although, such methods have been migrated to cloud computing environment [200–202], it is less clear the required levels of predictability hold. This research has proposed several resource allocation policies which were then evaluated against different workloads. Such works have generally focused on applying batch-queue models to multi-tier web application domains while focusing on improving QoS performance without, necessarily, considering operational costs. A clear unaddressed concern in such works is how do the proposed approaches generalize to wider classes of workload and application domains, in that tuned approaches are easily viable in highly predictable environments whereas they will tend fail in more dynamic environments.

Prior research has also sought to address cost issues through proposing cost-efficient dynamic resource allocation for data centers [90; 200; 203–205]. In most cases the objective is to maximize cloud provider profits while maintaining SLA. Again, this rests on underlying assumptions about the cloud computing environments QoS/QoE predictability. For example, Mao and Humphrey [206] introduced a

cloud auto-scaling strategy to address trade-offs between performance and cost while maintaining user defined application QoS levels. This presupposes that the effects of the auto-scaling strategies on the application-level QoS behaviors are a priori known, i.e., predictable.

Hassan et al. in [207] studied and tested workload associated with Big Data problems by running a group of typical Big data jobs on Amazon cloud EC2, i.e., video surveillance services. They collected statistics of workload by running the applications on Amazon EC2. Then based on the statistics they created a large emulation scenario and compared their proposed method with other approaches. They propose an online resource allocation method based on optimization. It has been assumed the workload of Big Data applications is unpredictable and so non-stationary. Therefore, they apply optimization on independent time slots. They showed that their proposed approach was cost effective, but their assessment did not seek to consider more detailed performance metrics such as delay, delay variability, throughput, etc.

M. Xu et al. [105] introduced a multiple QoS constrained scheduling strategy of multi-workflows (MQMW) to address the issue of multiple workflows with different QoS requirements. The proposed strategy was designed to schedule multiple workflows with arbitrary start times and multiple QoS requirements. They focused on a few aspects of performance such as the mean execution time or mean execution cost of all workflows. QoS constraints such as availability and reliability were not added to workflows. Moreover, it has been assumed that all application layer deployments are the same and all workflows have access to all resources. At the very base core, it is more like scheduling priority queues. The system is assumed deterministic and predictable and simulation is considered small with 25 users/workflows and 20 services/resources.

There are researches that consider predicting behavior and performance of large-scale distributed systems in clusters without considering their behavior in virtualized environments. Ganapathi et al. [156] proposed a model to predict the execution time of MapReduce jobs in order to maximize performance while minimizing costs. A workload generator based on statistical models was used to guide the prediction. However, the authors did not consider the full distribution to build the model, instead, the distribution is estimated from the 1st, 25th, 75th and 99th percentiles, causing information loss that compromises the accuracy of the model. Kavulya et al. [208] characterized resource utilization patterns and sources of failures to predict job completion times in MapReduce applications. They did not present an analysis

of data justifying the distributions used. Furthermore, they use only the method of Maximum Likelihood Estimation (MLE), which is sensitive to outliers. Kavulya et al. [208] presented Kolmogorov–Smirnov (KS) values with no critical value. Thus, it is not clear which set of distributions and parameters provide a good fit to the modeled data.

Other than dynamic resource allocation and workload modeling techniques to handle QoS issues, there are other projects which target QoS directly and propose several QoS management and monitoring techniques [209–216]. As an example, P. C. Hershey et al. [209] introduce a 5-step procedure named SoS (System of systems for Quality-of-Service) approach for QoS monitoring, management and response for large complex computing systems deployed in a cloud environment. They describe a mathematical model for the QoS metrics considered in SoS in order to identify a QoS breach whenever happens. Then, possible response actions to breach has been discussed. This work lacks ways to quantify effectiveness of this approach to restore QoS in the presence of anomalous enterprise events. In analysis of collected QoS metrics and reactions to breach, it has been assumed that cloud systems are stationary and ergodic for a specific application deployment which makes them predictable. This is the question that we aim to answer in this thesis.

Although many works have looked at performing QoS assessments via actual cloud deployments which can provide analyzable data but does not and cannot answer questions as to the QoS predictability of cloud systems or what does and does not influence it, i.e., why does the data collected from one deployment instance or set of instances say anything about the QoS behaviors expected for any other deployment instance or set of instances.

2.7 Ergodicity in Software Systems

In the early days of computing, ensuring the ergodicity of software systems was a fundamental aspect of engineering system design [217]. However, as computing technology advanced and systems grew more complex, the practice of proving ergodicity gradually waned. Instead, ergodicity and BET-compliance was often assumed without formal verification, leading to potential unpredictability and stability issues in modern software systems.

Recently, there has been a renewed interest in revisiting the concept of ergodicity within the context of software systems. This resurgence is driven by the critical

need for reliability and predictability in especially mission-critical applications, such as those found in cloud computing, nuclear power, and aerospace industries. A few pioneering works have started to re-examine the importance of ergodicity, but non advocating for formal checks and proofs to ensure that modern software systems can maintain stable performance and behavior under varying conditions.

In the paper "Towards Developing of Oberon System with Specific Requirements of Ergodicity," D.V. Dagaev [218] explores the development of an Oberon-based system designed to meet the stringent functional safety standards of IEC 60880 category A, essential for mission-critical software in industries such as nuclear power. The focus is on achieving ergodicity to prevent the degradation of system properties over time. The system's runtime environment and application software were prototyped using the Active Oberon A2 system, a high-reliability industrial real-time operating system. Key aspects include memory management, real-time processing, exception handling, and network reliability, all designed to maintain system stability and predictability. The system incorporates strict design principles, such as no memory allocation post-initialization, non-preemptive scheduling, and fixed time intervals for data processing, ensuring high reliability and stability. These measures prevent memory leaks, deadlocks, race conditions, and data loss, crucial for safety-critical applications. Despite potential inefficiencies in CPU usage, the approach significantly enhances system reliability, making it suitable for critical applications.

Although this work provides valuable insights into achieving ergodicity and reliability in software systems, it does not offer formal proof of why and when ergodicity is broken. Instead, the author applies a simulation method, demonstrating that 100 hours of simulation is not equivalent to 100 simulations of one hour each. While it is commendable that the author checks for BET-compliance, this method is not a reliable way to prove that BET-compliance does not hold.

In another recent work in the paper "The Application of the Theory of Dynamic Systems to Software Quality Estimation," A.V. Kopyltsov [219] explores the application of dynamic systems theory to evaluate and classify software quality. The author proposes that both hardware and software can be viewed as dynamic systems, placed in the phase space of their respective variables. By treating programs as dynamic systems, the paper introduces the use of spectral analysis methods, such as the Fourier transformation and auto-correlation functions, to classify programs based on their behavior in phase space. This approach aims to reveal important quality indices, such

as fractal dimensions and information entropy, which serve as measures of program complexity and quality.

Kopyltsov's work emphasizes the need for a unified approach to assess both hardware and software quality, drawing on fundamental concepts from modern physics and ergodic theory. The study shows that applying dynamic systems theory to programming allows for the evaluation of quality indices that are traditionally difficult to measure. This method not only separates the quality assessment of hardware and software but also highlights mixed indices, such as reliability, that pertain to both. The findings suggest that integrating these theoretical principles into software engineering can transition the field from an art to a science, grounded in rigorous mathematical foundations.

However, the paper has several shortcomings including that the approach faces scalability issues when applied to large, heterogeneous systems. Furthermore, the practical implementation of these methods is not detailed, limiting their adoption in current software engineering practices. The metrics proposed, such as fractal dimensions and information entropy, may be innovative but are not readily interpretable or accepted by industry practitioners. Additionally, the paper does not provide a comparative analysis with existing software quality assessment methods, which would help contextualize its advantages and limitations.

We are not aware of any other works that specifically address ergodicity in software systems.

2.8 Summary

In this chapter, we conducted a comprehensive review of the existing approaches for evaluating and predicting software performance, with a particular focus on the complex task of predicting performance in cloud services. We explored various methods for modeling software systems, encompassing both workflow modeling techniques and analytical approaches predominantly based on queuing theory. We highlighted the increasing popularity of cloud computing among service providers, which has led to a heightened emphasis on the quality and reliability of the services they offer. The ability to deliver fast and reliable services is crucial in the competitive landscape of cloud computing.

Furthermore, we discussed the growing importance of adopting new technologies like Docker containers, particularly in cloud environments. These technologies

have become necessary due to the agile development trend and the frequent updates required in modern applications. However, the use of Docker Containers in cloud environments introduces new considerations regarding the performance predictability. Actually, the combination of cloud computing and the utilization of containers presents both opportunities and challenges, because while cloud services offer scalability and flexibility, the introduction of containerization technologies raises concerns about maintaining the desired level of quality in the services provided and raise the question of statistical predictability in such systems.

Chapter 3

Formal Performance Modeling via Queuing

Analytical performance modeling is a popular and less expensive method of evaluating the performance of a system using formal analysis and modeling techniques. This often involves using mathematical models, such as stochastic processes to represent the system and its components. A popular method in performance modeling is Markov chain [220], which is a memory-less model for system's different states depicts transitions from one state to another, between a finite or countable number of possible states. In other words, it is a random process that describes a sequence of possible events in which the probability of each event depends only on the state attained in the previous event.

Another common method used in analytical performance modeling is Queuing Network (QN) which is a network of interconnected queues and can be used to predict the performance metrics of the system such as the number of requests in the system, the expected waiting time, and the utilization of servers [26]. These models then can be used to predict and analyze the run-time behavior of a system under different conditions, such as varying workloads, different system configurations, and different network conditions. Typically, as per numerous published works (e.g. [77–79]), the run-time behaviors of these QNM are analysed via the mathematics of Markov process or stochastic Petri Nets [221] or through simulation based approaches [26; 222]. Such academic works have shown impressive results, such as in [80] where simulated cloud applications performance levels are shown to be predictable three significant figures for skewness and Kurtosis measures. Importantly, though many real-world workload

traffic characteristics, such as self-similarity, long-range dependence, and heavy-tailed distributions, are not considered due to the resulting loss of analytical tractability.

Moreover, the complexity of academically simulated and analyzed systems are generally order of magnitude smaller in scale and complexity than commercial LDSSs. The latter can easily cost into the low millions to build, run concurrently on thousands of interacting cloud VM, have many hundreds of thousand a year in cloud deployment costs, and servicing active user bases numbering in the millions to hundreds of millions, placing them well beyond the contexts of academic research. Although the goal of analytical performance modeling is to provide insights into the system's behavior, identify bottlenecks, capacity limits, and other performance issues before the system is deployed, but here, we apply these models as universally approved models for system and performance modeling to model software systems of any size and then in the next chapter we will analyze the behaviour of these QNMs using EMM and DST to investigate how different behaviours such as queue drops can affect the system's performance predictability.

The Chapter begins by reviewing formal notations for performance modeling, then followed by discussing workload modeling and exploring typical workload notation for the purpose of this research and then introducing a unified QNM for software-centric systems from smallest units in embedded systems to most complex ones as in modern LDSSs using micro-architecture. We then discuss solutions of those QNMs.

3.1 Performance Modeling Formal Notations

In this section, we define the formal notations we used for modeling the run-time behaviour of software systems. We used the most common notations applied in the literature in the context of performance modeling. The notations presented in this Section are: Markov processes, workload modeling and QN.

3.1.1 Markov Processes

Markov processes [220; 223], also known as Markov chains, are a fundamental tool in the mathematical analysis of a system. They are a specific class of stochastic processes characterized by the Markov property, which states that the conditional probability distribution of future states of the process, given the present state, is independent of the history of the process. This property implies that the future behavior of the

process is dependent only on the current state and not on any past state. Markov processes play a central role in the quantitative analysis of systems, particularly in performance modeling. They can be used as a primary notation for the dynamics of a software system or as a method for solving various types of performance models such as QN and Stochastic Petri Nets.

A stochastic process is a family of random variables, denoted by $X = X(t) : t \in T$, where $X(t) : T \times \Omega$ is defined on a probability space Ω and the index set T, typically referred to as time, with a state space S. The state space can be either discrete or continuous, and the time parameter can also be either discrete or continuous. The statistical dependencies among the variables $X(t)$ are described by the joint distribution function. Stochastic processes can be classified according to the state space, the time parameter, and the statistical dependencies among the variables $X(t)$. For example, processes with discrete state space are known as chains, while processes with continuous state space are known as processes. A stochastic process is a Markov process if $X(t)$ has the Markov or memory-less property introduced earlier. More formally, the Markov property holds if and only if, given the value of $X(t)$ at some time $t \in T$, the future path $X(t_f)$ for $t_f > t$ does not depend on the past history $X(t_p)$ for $t_p < t$, i.e. for $t_0 < t_1 < \cdots < t_n < t_{n+1}$:

$$P(X(t_{n+1}) = x_{n+1}|X(t_n) = x_n, ..., X(t_0) = x_0) = P(X(t_{n+1}) = x_{n+1}|X(t_n) = x_n) \tag{3.1}$$

A Markov process is usually represented either as a labeled graph or as a transition matrix. An example of a labeled graph Markov Chain is shown in Figure 3.1 which models a web application in which the user interactive requests progresses from the initial node in the chain through to subsequent nodes, provided the required preconditions for the proceeding node have been met [4]. The equivalent transition matrix for the same system is shown in Equation 3.4. In this matrix, each row i and each column j represent a state and $A(i, j)$ represent the probability for the transition from j to i. Obviously, in both cases the probabilities of all transitions leaving the same state must sum up to 1.

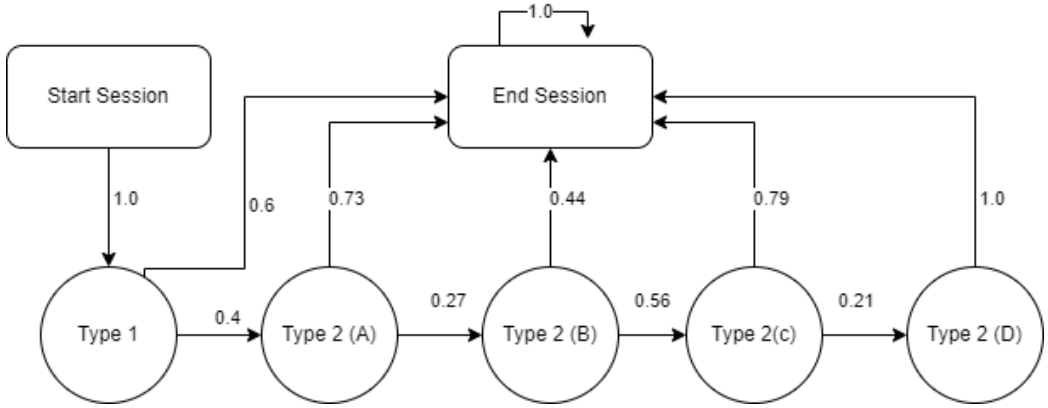


Figure 3.1: The exemplary industry-held workload Markov Chain for a user session[4].

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.27 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.56 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.21 & 0 & 0 \\ 0 & 0.6 & 0.73 & 0.44 & 0.79 & 1 & 1 \end{bmatrix} \tag{3.2}$$

The primary objective of the analysis of a Markov process is to calculate the probability distribution of the random variable $X(t)$ over the state space S, as the system reaches a steady pattern of behavior. This is known as the steady-state probability distribution. It is obvious from Figure 3.1 and the transition matrix in Equation 3.2 that the steady state is the "End Session" state and we achieve this by 6 steps as shown in Equation 3.3.

$$Steady\ Sate = \bar{X} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.3}$$

If we consider the aggregated traffic, then we have a slightly modified version of the original Markov Chain by removing the "Start Session" and "End Session" states. Figure 6.1 depicts the modified Markov Chain for the workload and Equations 3.4 and 3.5 shows the transition and steady state matrix respectively.

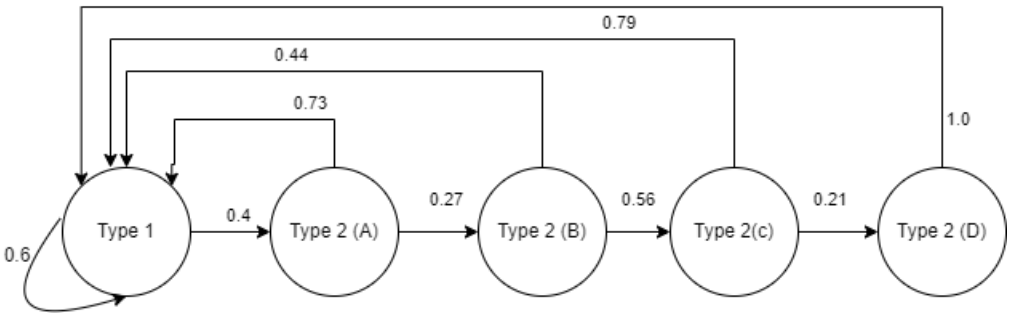


Figure 3.2: The exemplary industry-held workload modified Markov Chain[4].

$$A = \begin{bmatrix} 0.6 & 0.73 & 0.44 & 0.79 & 1 \\ 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0.27 & 0 & 0 & 0 \\ 0 & 0 & 0.56 & 0 & 0 \\ 0 & 0 & 0 & 0.21 & 0 \end{bmatrix} \tag{3.4}$$

$$Steady\ Sate = \bar{X} = \begin{bmatrix} 0.632 \\ 0.253 \\ 0.068 \\ 0.038 \\ 0.008 \end{bmatrix} \tag{3.5}$$

If a state $i \in S$ is entered at time t and the next state transition takes place at time $t + \tau$, then τ is the sojourn time in state i which is the time takes between state transitions. As of the Markov property, at any time point τ, the distribution of the time until the state changes again is independent of the time of the previous state change. It means sojourn times are memory-less and in any state of a Markov process is an exponentially distributed random variable. Performance measures based on subsets of states can be derived from this probability distribution. The solution of Markov processes is closely tied to their representation by matrices and finding the average behavior of the model involves solving the matrix equations. This is accomplished by solving the balance equations, also known as the global balance

equations, which connect the steady-state probabilities of the different states of the chain to one another. $X(t)$ converges to a strictly positive vector only if the transition matrix A is a regular transition matrix which means there is at least one A^n with all non-zero entries. Since $\bar{X}$ is independent from initial conditions, it must be unchanged when transformed by A. This makes it an eigenvector (with eigenvalue 1), and means it can be derived from A. In other words, the steady-state vector is the vector that, when we multiply it by A, we get the exact same vector back. If the transition probability from state i to k is given by q_{ik} and equilibrium distribution given by π then the balance equations for a Markov chain can be written as Equation 3.6,

$$\pi_i = \sum_{k \in S} \pi_k q_{ki} \tag{3.6}$$

where π_i is the steady-state probability of being in state i, and the sum is over all states k. These equations form a system of linear equations, and can be solved using matrix algebra which is computationally intractable for most queuing models [224].

Another method is to use the eigenvectors of the transition matrix. The eigenvectors of this matrix can be used to find the steady-state probabilities of the chain. The corresponding eigenvalues can be used to analyze the stability and convergence of the chain. It is important to notice that for a Markov Chain, all eigenvalues have magnitude less than or equal to 1 and one eigenvalue is equal to 1. The eigenvector corresponding to the eigenvalue of 1 is the steady state distribution. It's also important to mention that the eigenvalues and eigenvectors of a matrix are closely related to the matrix's geometric and algebraic properties, and understanding the eigenvectors and eigenvalues of a matrix can help understand the behavior of the system represented by the matrix. In our example, the steady state probabilities is obtained after 8 steps and is shown in Equation 3.5 which explains how busy each node will be in the steady state. In this example, we assumed the "OFF" period for ON/OFF workload model has a uniform distribution. For complex or high-dimensional Markov chains, it may be difficult or impossible to find an analytical closed form solution. In these cases, numerical methods or simulation can be used to find an approximate solution.

In an ergodic Markov chain, every state in the chain is accessible from any other state, either directly or through a sequence of transitions. This property implies that the chain possesses a single, well-defined stationary distribution, which describes the long-term behavior of the system. More formally, an ergodic Markov chain satisfies

two key conditions of irreducibility and periodicity which means there should be no fixed intervals at which the chain returns to a given state. When these conditions are met, an ergodic Markov chain guarantees convergence to a unique steady state regardless of the initial state. The ergodicity property makes ergodic Markov chains suitable for analyzing systems with steady-state behavior, as they ensure that the system will eventually settle into a stable state distribution.

3.1.2 Workload Model Assumptions

The workload arriving into a software system may or may not be predictable on its own, for example if it follows self-similar, heavy-tailed, long-range dependent behaviors, it would be hard to predict. In some cases the workload distribution might be non-stationary and that makes it unpredictable as well. In software systems with interactive users (not necessarily human), ON/OFF models are commonly employed to simulate user session-based workloads [143; 111; 164] such as one discussed in previous section. This model is particularly useful for simulating independent web traffic, where each session starts with a request and the ON time represents the time required to service that request. When the start of a user session is initiated by the user, the OFF times are typically assumed to follow an exponential distribution [151].

Actually, for analyzing workload or traffic patterns in a system, it is essential to consider the distribution of "ON" and "OFF" times. If these durations follow heavy-tailed distributions like Pareto or Weibull, the resulting traffic exhibits self-similarity, which is characterized by bursts of increased activity that occur at different time scales, creating a pattern that appears similar to itself when observed at different resolutions. This self-similarity increases queuing delay and queue drops rate [] which leads to unpredictability as we will discuss in the next Chapter.

Accurately capturing self-similarity effects in simulations of workload is crucial for assessing software performance. Failure to reproduce these effects can lead to unreliable predictions. To determine the appropriate modeling approach, it is necessary to conduct statistical tests. These tests help determine whether the traffic follows a Poisson model, or if a more complicated self-similar model is required to capture the bursty nature of the workload traffic accurately. The choice between these models has implications for accurately predicting system behavior and more fundamentally the predictability of the system in the first place. As an instance, if the workload follows a Poisson ON/OFF traffic model then it is reasonable to predict the incoming

workload where "prediction" is in the sense that the workload $x(t)$ is within some upper and potentially lower bounds B^{upper} and B_{lower} with some known probability ϵ as formulated in Equation 3.7.

$$\hat{P}_x[\ x(t) \in [B_{lower}(t), B^{upper}(t)]] \geq (1 - \epsilon) \tag{3.7}$$

Poisson ON/OFF traffic can be modeled as a two-state Markov process where the system alternates between an active state ON and an inactive state OFF with certain probabilities. The state transition probabilities are defined by the arrival rate λ and the service rate μ of the traffic. The arrival rate describes the rate at which new traffic arrives in the system and the service rate describes the rate at which the traffic is processed by the system. The arrival process is modeled as a Poisson process, which means the inter-arrival time between successive traffic arrivals is exponentially distributed. The time spent in the active state ON is modeled as an exponential distribution with a mean of $1/\mu$. The time spent in the inactive state OFF is modeled as an exponential distribution with a mean of $1/\lambda$. The system's behavior can be characterized by the steady-state probabilities of the system being in the On and Off states, which can be then found by solving balance equations.

If workload follows a non-exponential ON/OFF traffic model then it will not generate a memory-less Markovian model which generally removes the analytical solutions. Although the Markovian chain could still be simulated, if one takes to account all possible ways one could have arrived in each given state. There are some analytical solutions for Gaussian ON/OFF models.

Finally, if we assume that the incoming workload is stationary and statistically known then what we are looking at, are the impacts that their execution environment could be cloud or otherwise has on the predictability of the performance measures of the executing system. In that sense, the execution environment acts as a transform, or more accurately - since it is non-linear - a mapping, on the incoming workloads. Therefore, the incoming workloads could be stationary and ergodic and then the cloud can shift that into a non-ergodic non-stationary measurement regime, i.e., the nature of the cloud could take something that was predictable and mapping it into something that no longer is predictable.

If $x_k(t)$ is the k^{th} instanced of the incoming workload and $G_{t,k}(.)$ is the impact of the execution environment then the measurable performance features contained in y_k could be defined as Equation 3.8.

$$y_k(t+\tau) = G_{t,k}[x_k(t)|x_j(t')] \tag{3.8}$$

for all other incoming j workloads and all $t' < t + \tau$ which means how all the other workload that has entered the system impacts how $x_k(t)$ produces the measurable performance features contained in $y_k(t+\tau)$ where $\tau > 0$ denotes the delay the system takes to begin processing $x_k(t)$.

In order to demonstrate the profound impact of the environment on performance measures, we present an industry-held system as an illustrative example. As discussed in the previous section, the exemplary industry-held system operates in a well-defined environment where the incoming workload follows a statistically predictable and well-behaved pattern, allowing for the application of an easily solvable Markov Chain model. Moreover, the workload Markov Chain satisfies the ergodicity conditions. In the experiment, the production system in a commercial cloud, underwent a 30-minute load test applying several scenarios, during which the necessary parameters were collected to assess the response time distribution for user requests. To gain a comprehensive understanding of the response time distribution, they repeated each scenario 10 times without changing any configurations. The results of their experiments revealed that the response times exhibited highly variable behaviour as shown in Figure 3.3 and dependent on the initial conditions such as the underlying cloud servers, the network conditions, and random variations in the arrivals of user sessions and timing of requests. They concluded on one hand, there is a need for multiple runs of the same load for load testing and capacity planning purposes but on the other hand, using summary statistics to evaluate the system's performance over many tests may provide misleading results, as it would be averaging over modes of behaviour with different underlying response time distributions[4].

The widespread occurrence of performance variability and the reported issues within industry settings serve as the primary motivation for undertaking this research. The next Chapter aims to uncover the underlying factors responsible for such inconsistencies in system behavior, even when subjected to identical loads and configurations.

3.1.2.1 Typical Workload

We can borrow the idea of typical sequences from information theory and define a typical workload for a software system which means within the given software

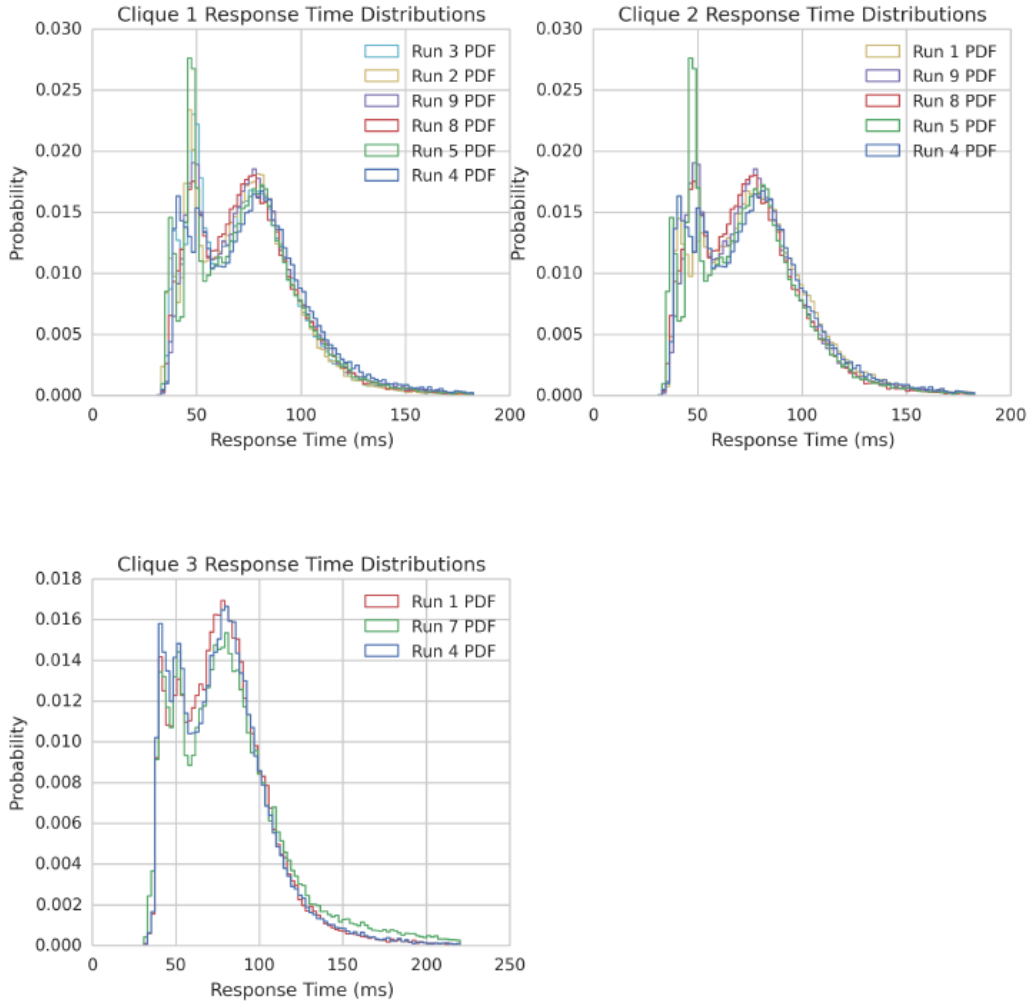


Figure 3.3: Increased Traffic Source Scenario Clique Distributions (AD-test) [4]

system, of all of its potential execution paths we are concerned with those that are typically executed when the system is processing "typical" or indeed normal/common workloads. Based on the concept of entropy in statistical mechanics, if we draw a sequence of n independent and identically distributed (i.i.d.) random variables, the probability of a typical sequence is about $2^{-nH(X)}$ and that there are about $2^{nH(X)}$ of such typical sequences, where $H(X)$ is the entropy of the random variable X measured in bits as shown in Equation 3.9.

$$H(X) = -\sum_{x} p(x) \log_2 p(x) \tag{3.9}$$

The asymptotic equipartition property (AEP) shows that most sequences are typical in that they have a sample entropy close to H. So attention can be restricted to these approximately 2^{nH} typical sequences [226]. If a sequence $x_1, ..., x_n$ is drawn from an i.i.d. distribution X defined over a finite alphabet $\mathcal{X}$, then the typical set, ${A_\epsilon}^{(n)} \in \mathcal{X}^{(n)}$ is defined as those sequences which satisfy Equation 3.10:

$$2^{-n(H(X)+\epsilon)} \leq p(x_1, x_2, ..., x_n) \leq 2^{-n(H(X)-\epsilon)} \tag{3.10}$$

As a trivial example, assume we have eight different types of requests and suppose probabilities of having each request are $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$. The entropy is 2 and having let's say a sequence of 10 consecutive requests, then the number of typical sequences will be $2^{10\times2}$ or 2^{20} while the total number of all possible sequences is 2^{30}. We should note that typicality here is only concerned with the probability of a sequence and not the actual sequence itself. Equivalently, we need software systems to be predictable for the most probable workloads not necessarily the unusual ones. In summary, we assume that the incoming workload to the system is statistically predictable as defined in Equation 3.3 and consists of typical workload and repeatable.

3.1.3 Queuing Networks

Queuing Networks(QN) is a network of connected queues governed by rules of Queuing theory and often used to model, analyze and predict the behaviour of computer systems and networks [107; 227]. Basically, in computer systems, everything is stored in queues from instructions waiting to be executed on CPU, packets of data in routers and switches, http requests in a load balancer to VMs in cloud waiting for the physical servers to become available. The behavior of the system is then described using Queuing-theoretic performance measures such as the average number of entities in the system, the average waiting time in the queue, and the probability of the system being in a certain state. This model can be represented and solved using a set of differential equations or a Markov chain model. It's important to note that QN models can become very complex when modeling large systems with multiple service stations and customer classes. As the number of service stations and customer classes increases, the number of equations required to model the system also increases, making it difficult to solve the model analytically. In such cases, simulation-based approaches may be more appropriate.

There are two main types of queuing networks: open and closed. In open networks, customers arrive from an external source and leave the system after being served. In closed networks, there is a fixed population of customers that circulate among the service stations [26]. Kendall notation, also known as Kendall's notation, is a standard system used to describe the characteristics of a queuing system and named after the British statistician David George Kendall, who introduced it in 1953 [228]. The notation consists of a string of letters and numbers that describes the system, with the format $A/S/m/k/n/D$. For example, M/M/1 queue as shown in Figure 3.4 represents a queuing system with a Markovian arrival process λ, a Markovian service time $\boldsymbol{S}$, one server, infinite queue capacity, infinite requests and a FIFO service discipline. This notation is explained in more detail in the following:

- $\boldsymbol{A}$ represents the arrival process of requests in the system. The most common ones represented as M (Markovian or memoryless), D (Degenerate distribution), or G (for general)
- $\boldsymbol{S}$ represents the service time distributions. The most common ones are M, D and G similar to arrival process distributions.
- $\boldsymbol{m}$ represents the number of servers in the system.
- $\boldsymbol{k}$ represents the maximum queue capacity.If this number is omitted, the capacity is assumed to be unlimited, or infinite.
- $\boldsymbol{n}$ represents the size of requests population. If this number is omitted, the population is assumed to be unlimited, or infinite.
- $\boldsymbol{D}$ represents the service discipline, or the order in which requests are served. It can be represented as FIFO, LIFO, PQ (priority queuing) or PS (priority service). The default discipline is FIFO.

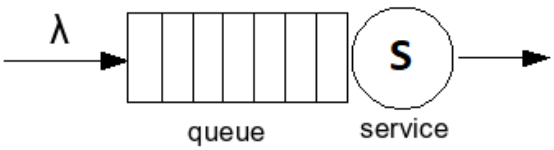


Figure 3.4: M/M/1 Queue

The different types of queuing systems can be analyzed mathematically to find performance measures of steady state in which all transient behavior has ended, the

system has settled down, and the values of the performance measures are independent of time. The system is then said to be in statistical equilibrium; i.e., the rate at which jobs enter the system is equal to the rate at which jobs leave the system. Such a system is also called a stable system. Transient solutions of simple queuing systems are available in closed form, but for more general cases, we need to resort to Markov chain techniques. The most important performance measures are as follows:

- $\boldsymbol{K}$: The probability of K number of requests in the system is denoted by $P[K]$. We can describe the behavior of a queuing system by the probability vector of the number of requests in the system. The mean values of most of the other interesting performance measures can be deduced from $P[K]$.

- $\boldsymbol{\lambda}$: The mean arrival rate of requests is denoted by λ which is the average number of requests arriving per unit of time.

- $\boldsymbol{L}$: The average number of requests in the system is denoted by L. This measure, also known as the mean number of requests, is the expected value of the number of requests in the system at any given time and can be calculated using the Equation 3.11 usually known as Little's result, theorem, lemma, law, or formula [229].

$$E[L] = L = \sum_{L=0}^{\infty} L \times P[L] = \lambda W_q \tag{3.11}$$

  where where $E[.]$ denotes the expectation operator and λ is the arrival rate of request to the queue and W_q is the mean waiting time of a request in the queue.

- $\boldsymbol{W_q}$: The mean waiting time in the queue is denoted by W_q. This measure, also known as the mean waiting time, is the expected value of the time a request or a job spends waiting in the queue before being serviced.

$$W_q = \frac{L_q}{\lambda} \tag{3.12}$$

  where L_q is the mean number of requests waiting in the queue.

- $\boldsymbol{s}$: The mean service rate of the servers is denoted by $\boldsymbol{s}$. Service time is $\frac{1}{\boldsymbol{S}}$.

- ***U***: The utilization of servers is denoted by U. This measure, also known as the server utilization, is the proportion of time that servers are busy servicing requests or jobs. It can be calculated using Equation 3.13.

$$U = \frac{\lambda}{s} \tag{3.13}$$

- ***Z***: The throughput Z is the number of requests are serviced by the system per unit of time. It can be calculated using Equation 3.14.

$$Z = \lambda \times (1 - U) \tag{3.14}$$

- ***R*** The response time R is the total time a request spends in the system and calculated by Equation 5.19.

$$R = \frac{D}{1 - U} \tag{3.15}$$

where $D = (1/s) \times V$ and V is the average number of visits to the server in the general case of feedback queues where fully processing a workload element may require more than a single visit to a queue as shown in Figure 3.5.

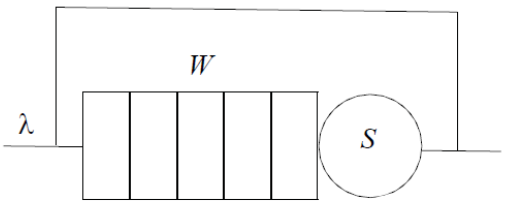


Figure 3.5: Feedback queues

3.1.4 Layered Queuing Networks

A LQN is an extension to QN and is a mathematical model that is used to analyze the performance of complex computer systems composed of multiple interconnected queues or resources, such as distributed systems, client-server systems, and web-based systems. In an LQN, each layer represents a different level of service, and each

layer may have its own set of queues, servers, and arrival processes. The layers are interconnected, with requests flowing from one layer to the next. Each layer may also have different service times, queuing disciplines, and resource sharing policies. LQNs are particularly useful for analyzing the performance of distributed systems, where multiple layers of service may be provided by different components or subsystems. LQNs allow for the modeling of interactions between the layers and the analysis of the impact of these interactions on the overall system performance.

The solution of LQN models typically involves finding the steady-state probabilities of the different states of the network, as well as performance measures such as the mean response time and throughput of the system. It's important to note that LQN models can be complex and computationally intensive, so it's important to have a clear understanding of the system being modeled and to use appropriate tools and techniques to solve the model. In this Chapter we use the mathematics of LQN for calculating the performance metrics.

3.2 Performance Modeling Using Queuing

To model a software system using layered QN, the first step is to identify the different layers of the system architecture, representing the different functional components of the system. These layers should have distinct responsibilities in processing user requests. For each layer, the service demand must be defined as the amount of work that the layer must perform to process a request. This service demand distribution can be expressed in terms of time or CPU cycles required to complete the work. The arrival rate can be estimated based on historical data or expected usage of the system. Once the service demand and arrival rate are known, a QN can be created with required queues and a servers for each layer. The servers are connected to the next queues with arrows indicating the direction of the flow of requests.

The QN mathematics can then be used to calculate performance metrics such as response time, throughput, and utilization as described in Section 3.1.4. The total response time for a request can be calculated as the sum of the service times for each layer plus the time spent waiting in each queue. The throughput can be calculated as the number of requests processed per unit time in the whole system. Usually, as more data is gathered about the system's performance, the model can be refined to make it more accurate. This may involve adjusting the service demand or arrival rate

based on new information or adding additional layers to the model to capture new components of the system.

In this section, we model software systems via QN from the most trivial ones running on one machine called single server to the most complex ones containing millions of queues deployed in commercial clouds such as the case in most SaaS applications. We assume all our software systems are dealing with typical workload and our main modeling objective is to obtain the performance measures as shown in Figure 3.6. We build our QN models gradually on top of each other as the software systems becomes more complicated.

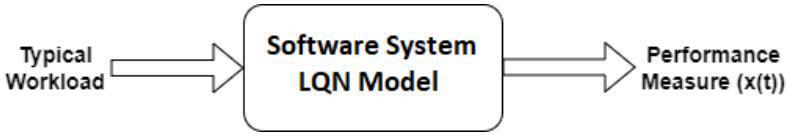


Figure 3.6: LQN models as a general performance modeling tool.

3.2.1 Single Server Software Systems

Single server software systems refer to systems where a single server or processing unit is responsible for handling all incoming requests. These systems typically have a CPU or a dedicated server that executes the required computations. In such systems, often all requests are processed sequentially by the single server. This means that each request must wait for the previous request to complete before it can be processed. The server allocates its resources, such as processing power, memory, and input/output capabilities to handle each request in a sequential manner.

Examples include embedded systems, traditional client-server applications, where a single server responds to requests from multiple clients, as well as stand-alone applications running on a single machine or server. One of the simplest software systems are ones on embedded systems as these systems are often dedicated to a particular function or a set of functions and are typically embedded within a larger product or infrastructure and been used in various industries such as automotive, healthcare, industrial automation, consumer electronics, and more. They are usually running independently and have their own operating system and hardware. Figure 3.7 shows the architecture diagram of a typical embedded system consisting of two main parts, embedded hardware and embedded software. The embedded hardware primarily includes the processor, memory, bus, peripheral devices, I/O ports, and

various controllers. The embedded software usually contains the embedded operating system and various applications which can also include middle-ware and device drivers.

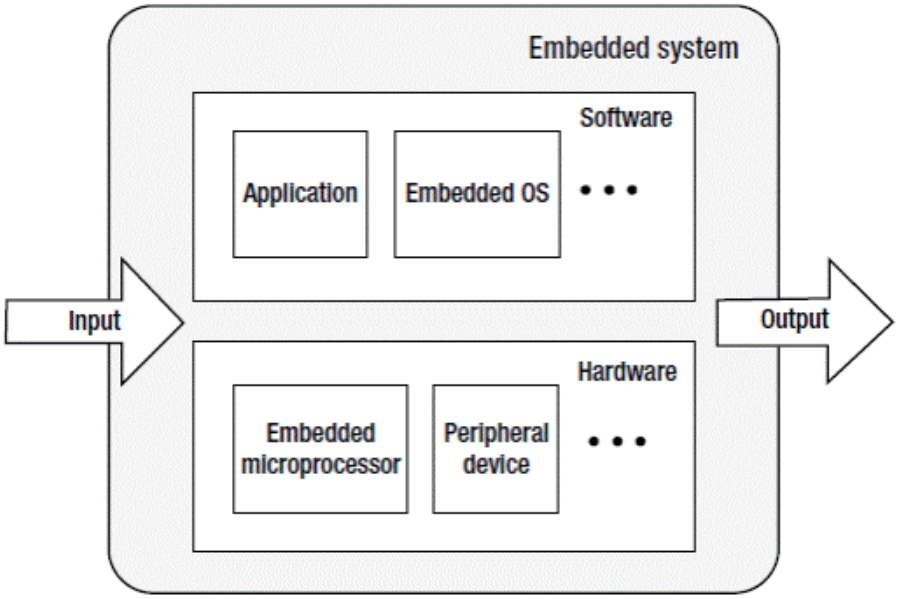


Figure 3.7: Basic architecture of an embedded system.

While embedded systems may be considered simpler compared to larger software systems, they still consists of multiple layers of interconnected queues due to the presence of various layers/components and interactions within them as depicted in Figure 3.8. As shown in this figure, each request to the system will break down to several sub-requests and accordingly several instructions going thorough different processing stages.

In addition, even single server software systems often require task scheduling to ensure timely execution of different activities. Each scheduled task can be associated with a specific queue representing its execution requirements. For example, in a real-time control system, there may be queues for high-priority control tasks, lower-priority monitoring tasks, or background maintenance tasks. The reader should keep in mind that the result of later DST analysis applies to all levels of queues in the system.

The QNM characteristics of a single server software system is described in below:

- **Open Network**

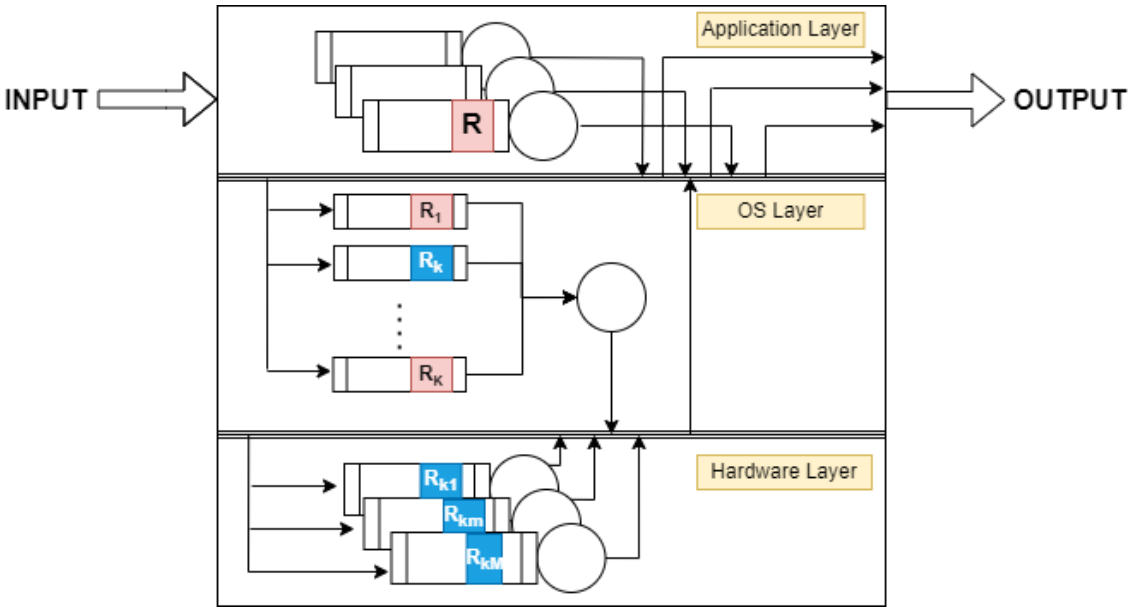


Figure 3.8: Several layers of QN in a simple system.

- **Single Server**: The system has a single server that processes requests or transactions in a sequential manner.

- **Finite Queue**: The system has a finite queue capacity that can hold limited number of requests or transactions waiting to be processed. Once the queue reaches its maximum capacity, the flow of requests into the service center is halted, leading to a blocking situation. Extensive research literature has defined and analyzed various blocking mechanisms such as Blocking after service (BAS), Blocking before service (BBS) or Repetitive service blocking (RS), which accurately represent the distinct behaviors exhibited by real systems with limited resources [230; 231]. When assuming exponential distributions, it is possible to define and analyze the continuous-time Markov chain that underlies the queueing network. In certain special cases, queueing networks with blocking exhibit a product-form solution, subject to specific constraints, for different types of blocking.

- **Arrival Process**: The arrival of requests or transactions to the system follows an ergodic and stationary random process.Typically the arrival process is modeled as Poisson but it doesn't have to.

- **Service Time**: The time required to process a request or transaction by the server which typically follows an exponential distribution.
- **Scheduling Discipline**: The requests or transactions in the queues are processed according to the scheduling process which is usually FCFS or round robin in the order in which they arrived, without any preference or priority given to any specific request or transaction.
- **No Resource Contentions**: The system does not have any resource contentions, such as locks or shared resources, that can cause delays or contention among requests or transactions.
- **Typical Load**: The system is subject to a typical load according to the definition of typical workload in Section 3.1.3.

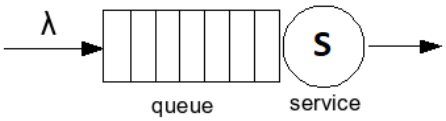


Figure 3.9: Single Server software system queuing model in the application layer.

3.2.2 Distributed Software Systems

A distributed software system can be modeled as a LQN consisting of multiple interconnected queues, each representing a component or a service in the whole system. The queues can be connected in various ways depending on the structure/architecture of the system, such as in a hierarchical or mesh topology. In such a distributed software system, requests arrive at one or more entry points, which may be load-balancers or front-end servers as shown in Figure 3.11. These requests are then processed by a series of interconnected services, each with their own processing times, service rates, and queue sizes. The output of one service becomes the input to the next, forming a chain or a network of services.

To model such a system, we can assign a queue to each service, with the input and output flows represented by the arrival rates and service rates of the queues, respectively. The arrival rates can be based on historical data or estimates of the traffic patterns of the system, while the service rates can be estimated based on

the performance characteristics of the servers. These connected queues often called tandem queues in the literature as shown in Figure 3.10.

Calculating the response time of a tandem queue can be complex, especially when dealing with general service times and finite buffers. One approach to approximate the response time in a tandem queue is to decompose the tandem queue into single-buffer subsystems and iteratively estimate the unknown parameters of the service times of each subsystem [232]. For a tandem network with two single-server nodes with infinite buffers and exponentially distributed service times, we can analyze the mean sojourn/response times at each node as explained in [233]. Let $T_k(n, m)$ be the expected sojourn time spent at queue k by a tagged request that joins a system being in state $(n-1, m)$. The total expected sojourn time for a request in the tandem network can be calculated as Equation 3.16.

$$T(n, m) = T_1(n, m) + T_2(n, m) \tag{3.16}$$

In more general cases, we may need to use simulation, numerical methods, or other techniques to analyze the response time in tandem queues, depending on the specific characteristics of the system, such as the arrival process, service times, and buffer capacities [234].

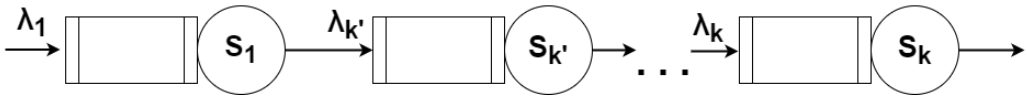


Figure 3.10: Distributed system QN as tandem queues.

The arrival rate has a significant impact on the response time in tandem queues. As the arrival rate increases, the system may become saturated or unstable, leading to longer response times. The maximum throughput of a tandem queueing system is the maximum arrival rate that the system can support before becoming saturated [235]. The response time is also affected by the different arrival processes, such as batch arrivals, Poisson arrivals [236] or heavy-tailed traffic arrivals [237]. The dependency of the steady-state mean of the sojourn time on the arrival rate can provide insights into the behavior of the tandem queuing model and can be used as a base for developing approximations for it [238].

Moreover, the number of servers in a tandem queueing system can affect the response time. As the number of servers increases, the maximum throughput of the system converges. This means that the system can handle more requests and maintain

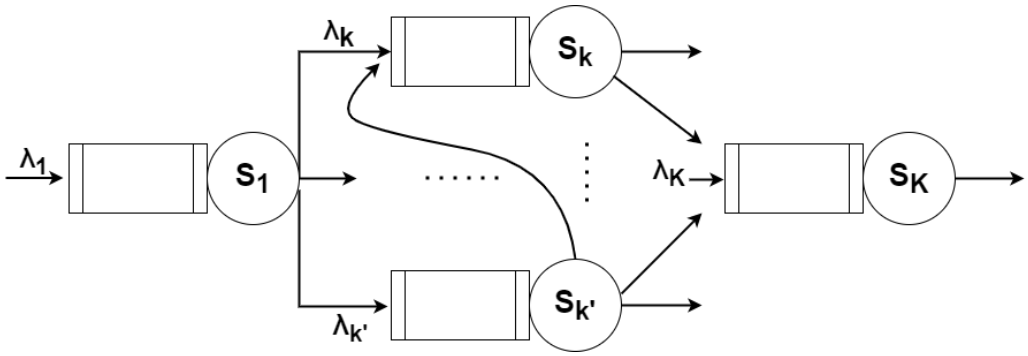


Figure 3.11: Distributed system QN as tandem queues with an arbitrary topology.

stability, which can lead to reduced response times. However, the analytic formula for the maximum throughput of the system tends to become more complicated as the number of servers increases [235].

In systems with heterogeneous servers, the fastest server has a larger effective rate of service than the slowest server. This implies that the overall maximum throughput of the system depends on the efficiency of the individual servers and their service rates. Also, the buffer size in a tandem queueing system affects the response time by influencing the throughput and mean sojourn time. In a study of single-server tandem queues with finite buffers, jobs were served according to the Blocking-After-Service protocol, and the tandem queue was decomposed into single-buffer subsystems. The service times in these subsystems included starvation and blocking, and the approach attempted to include dependencies due to blocking by employing the concept of Markovian Arrival Processes. This indicates that the buffer size plays a significant role in determining the response time in a tandem queueing system, as it affects the system's ability to process requests and maintain predictability [232]. In the next Chapter, applying DST, we will show how the buffer size and queue drops can affect the stability and predictability of software systems.

3.2.2.1 Shared Servers in Cloud Environment (IaaS)

In a distributed system running on a shared server, the architecture becomes slightly more complex as the components are encapsulated within VMs. Each VM within the shared server represents a self-contained environment with its own operating system, resources, and configurations. The distributed system's components, such as servers, databases, or services, are deployed within these VMs. The host server's underlying hypervisor manages resource allocation and scheduling among the VMs.

It is still possible to model this system using tandem queues and all the elements discussed earlier such as the number of servers and their speed, buffer size, arrival rate, etc., which affect the performance of the system, still hold true. However, in this case, the VMs and their associated queues dynamically go on and off based on the scheduling decisions made by each server's hypervisor scheduler as shown in Figure 3.15. In this figure, each queue is an aggregated queue presenting one service in a VM. As we can see from the QN in the picture, software systems that have dependency between services affected more compared to the software systems where the components are more independent and can run in parallel. This is one of the reasons in popularity of architectures applying parallelism such as microservice architecture and MapReduce systems.

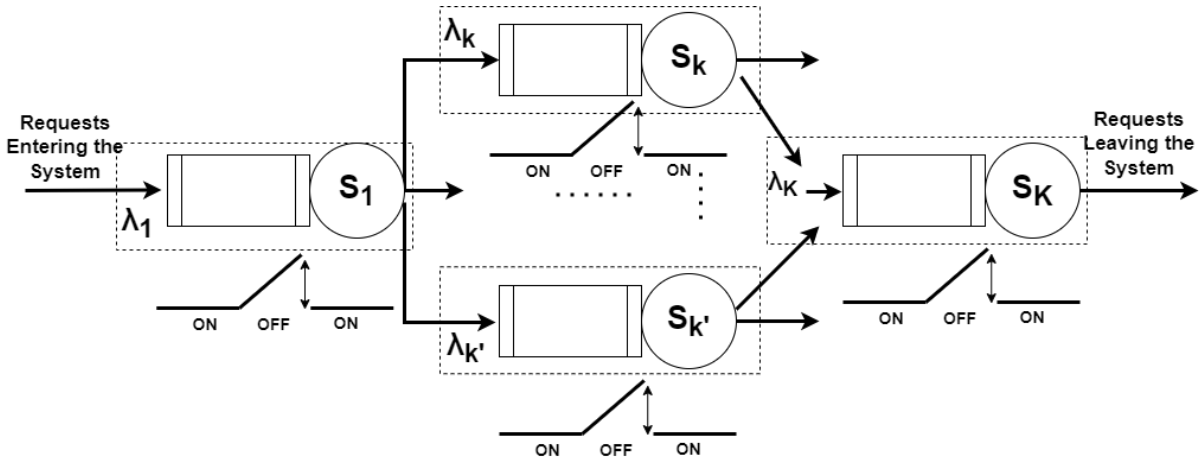


Figure 3.12: Distributed system QN as tandem queues with on/off switches on queues.

Here, the scheduling decisions made by the hypervisor can impact the response time variability of the VMs [239]. If the scheduling algorithm is unpredictable or favors certain VMs over others, it can result in increased response time variability. Individual queues may experience inconsistent response times which has a cascading effect in a tandem queue and amplify the impact on the overall response time as delays propagate through the system, leading to potential performance issues or user dissatisfaction. Conversely, a well-designed scheduling algorithm can reduce response time variability and provide more predictable and consistent performance across the VMs.

What happens in practice though is that hypervisors often support over-commitment, allowing for more VMs to be provisioned than the physical resources can fully accom-

modate. Here, the scheduling decisions become crucial in managing resource sharing and preventing resource contention. The hypervisor scheduler needs to efficiently allocate CPU resources to the VMs, considering their priorities and resource demands, to ensure adequate performance and prevent excessive queuing and response time delays.The hypervisor scheduler plays a crucial role in determining the overall performance of a cloud environment, and different scheduler types are best suited for different virtualization scenarios. In scenarios where specific VMs or applications require guaranteed QoS levels, the hypervisor scheduler should become critical. By prioritizing certain VMs or allocating dedicated CPU resources to them, the hypervisor ensures that performance and response time meet the defined QoS requirements. This enables critical applications or VMs to receive the necessary resources to perform predictable, even under high loads or contention situations. In the next Chapter, applying DST, we'll show how scheduling can affect performance predictability.

More specifically, Hyper-V hypervisor supports several modes of scheduler logic that determine how the hypervisor schedules virtual processors on the underlying logical processors. These scheduler types include the classic scheduler, the core scheduler, and the root scheduler. The classic scheduler uses a round-robin approach to map a VM's virtual processor to an available simultaneous multi-threading (SMT) thread that can execute instructions on behalf of the virtual processor. The core scheduler offers a strong security boundary for guest workload isolation and reduced performance variability for workloads inside VMs that are running on an SMT-enabled virtualization host. The core scheduler is used by default starting in Windows Server 2019. On Windows Server 2016, the core scheduler is optional and must be explicitly enabled by the Hyper-V host administrator, and the classic scheduler is the default. The older root scheduler hands over control of work scheduling to the root partition, in this configuration, the NT scheduler within the root partition's operating system instance takes charge of all aspects related to scheduling work on the system's Logical Processs (LPs). It manages the allocation and distribution of workloads across the LPs, determining which processes and threads are executed on which LPs at any given time [240].

As another example, the Xen based hypervisors which is widely recognized and adopted as a popular virtualization technology [241], employs a scheduling mechanism that efficiently allocates virtual CPUs (vCPUs) to physical CPUs (pCPUs). With modern CPUs supporting multiple threads on each core, the Xen hypervisor can take advantage of this feature by assigning vCPUs to different threads within a physical

core. The Xen hypervisor utilizes a two-level hierarchical scheduling framework which allows for the allocation of pCPUs to virtualized hosts and enables the execution of multiple VMs simultaneously. Within each VM, multiple vCPUs can be assigned, allowing for parallel execution of tasks inside VM. This helps with yet another layer of virtualization and parallelism in containers which we will discuss in the next section.

One notable scheduler available in the Xen hypervisor is the Real-Time Deferrable Server (RTDS) scheduler. The RTDS scheduler caters to the needs of soft real-time systems, which require deterministic and predictable response times. It ensures that tasks with real-time constraints are executed within specified deadlines and provides guarantees for their timely completion which reduces variability in response time and helps with predictability issues as we will show with DST analysis. It's important to note that the performance and scheduling characteristics of type-1 hypervisors can also be influenced by the underlying hardware, such as CPU architecture, memory capacity, and I/O subsystems [242]. Additionally, specific workload characteristics and deployment scenarios can impact the performance of a hypervisor and consequently the software systems running on top of it.

3.2.2.2 Applying Container Technology (CaaS or PaaS)

Container technology, such as Docker or Kubernetes, has revolutionized the deployment and management of distributed software systems. When applying container technology to a distributed software system, we can still model it as a QN to analyze its performance characteristics. Each container represents a specific functionality or microservice within the system and can have its own queue, representing the waiting time for requests to be processed by that specific component. The arrival rate of requests to each container and the service rate of each container needs to be modeled to estimate the performance of the system. Interactions between containers can be represented as communication links or channels connecting the queues of different containers in the queuing network and communication time between containers should be considered in the overall response time analysis. Container orchestration platforms like Kubernetes handles the scheduling and placement of containers across a cluster of machines. The scheduling algorithm determines which containers should be deployed on which hosts, considering factors such as resource availability, load balancing, and fault tolerance. The scheduling decisions impact the performance and can be modeled in QN by varying the distribution of workload and resource utilization.

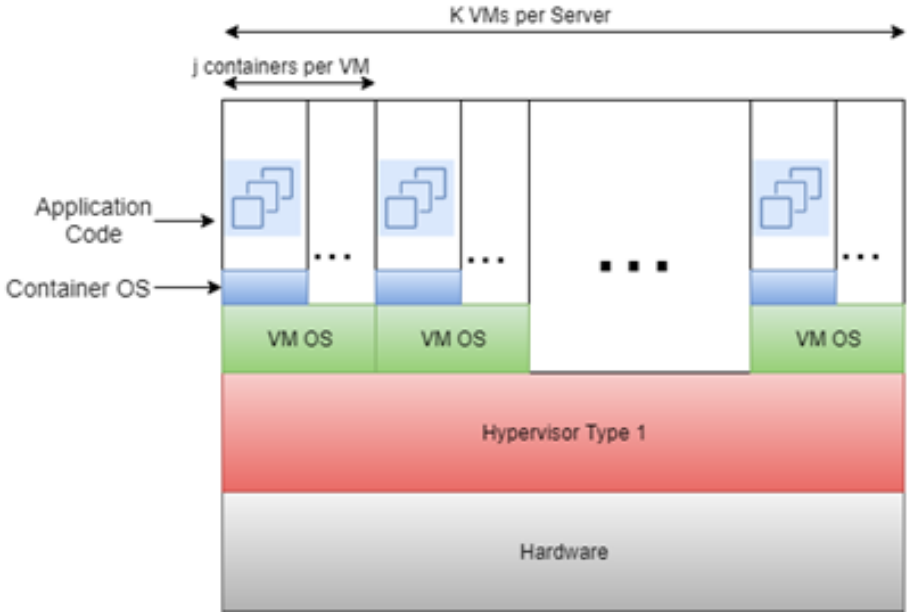


Figure 3.13: An application running in a Docker container inside a VM on a server in cloud.

The additional level of virtualization in CaaS platforms introduces some scheduling complexities and performance implications. Figure 3.13 depicts the multiple layers of virtualization found in a cloud server, illustrating how an application operates within a Docker container, which in turn runs within a VM on a server. At the bottom of the diagram, we have the physical hardware of the server, which provides the underlying computational resources. On top of the hardware, a hypervisor or virtualization layer is employed to enable the creation and management of VMs. Within these VMs, an application is running inside a Docker container. The Docker container operates within the VM and benefits from the resources allocated to the VM by the hypervisor. The container encapsulates the application, including its code, libraries, and configurations, providing a self-contained and portable unit of deployment can be used for scaling and migration to other hosts.

In Figure 3.14, the focus shifts to the levels of schedulers involved in the system. The hypervisor scheduler is responsible for managing and scheduling the VMs on the server. It needs to ensure fair resource allocation and efficient utilization of the underlying hardware resources. Within each VM, there is an operating system scheduler, which is responsible for scheduling tasks and processes within the VM including containers. This scheduler determines the allocation of resources, such as CPU and memory, among the applications and containers running within the VM. However,

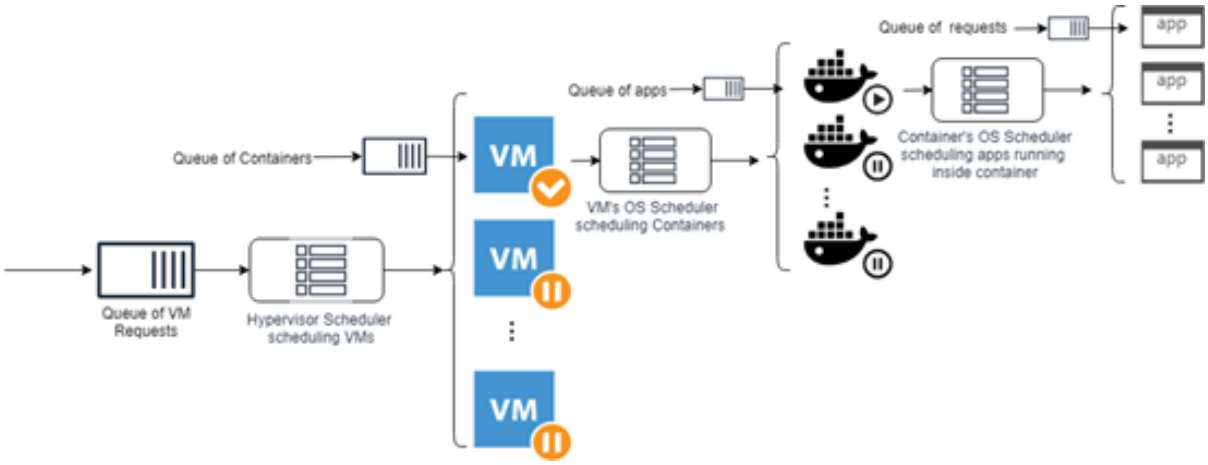


Figure 3.14: Different level of scheduling in a server

there is a third level of virtualization and scheduling introduced by Docker Engine, which is responsible for managing containers within the VM. Docker Engine acts as a container run-time orchestrator, providing the necessary tools and services to manage and schedule containers efficiently. Inside Docker Engine, there is another layer of scheduling that occurs within containers themselves. Containers have their own internal schedulers, which handle the allocation of resources and scheduling of jobs and applications running inside the container. It's important to note that containers run their own operating systems, separate from the underlying VM's operating system. This allows for greater flexibility and portability, as each container can have its specific dependencies and configurations.

As a result, the levels of virtualization and scheduling in a cloud server involve the hypervisor scheduler at the VM level, the operating system scheduler within the VM, the Docker Engine scheduler for managing containers, and the internal scheduler within each container. Each level of scheduling plays a crucial role performance predictability and in ensuring efficient resource utilization and task coordination within the distributed software system. In such a shared physical server environment, VMs, containers, and applications all utilize the same available server resources. However, they can only execute when they are allocated time slices by their respective base operating system scheduler. Figure 3.15 visually depicts the synchronization of schedulers necessary for the application layer processes to run.

In the LQN modeling approach, each queue represents a specific layer within the system. The queues are activated or deactivated based on the scheduling policies implemented at different levels. Figure 3.16 provides a representation of the dynamic

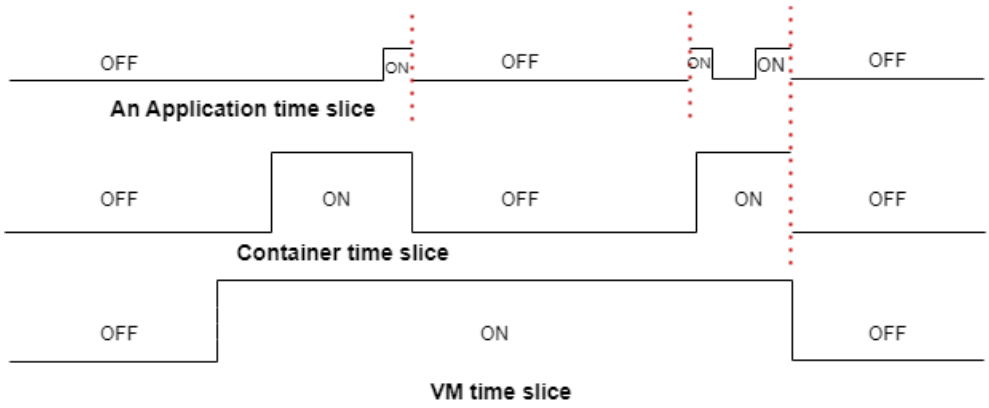


Figure 3.15: Applications can only run when all of the schedulers align

nature of queue activation and deactivation. It demonstrates how the scheduling decisions made at different levels influence the availability and utilization of the queues.

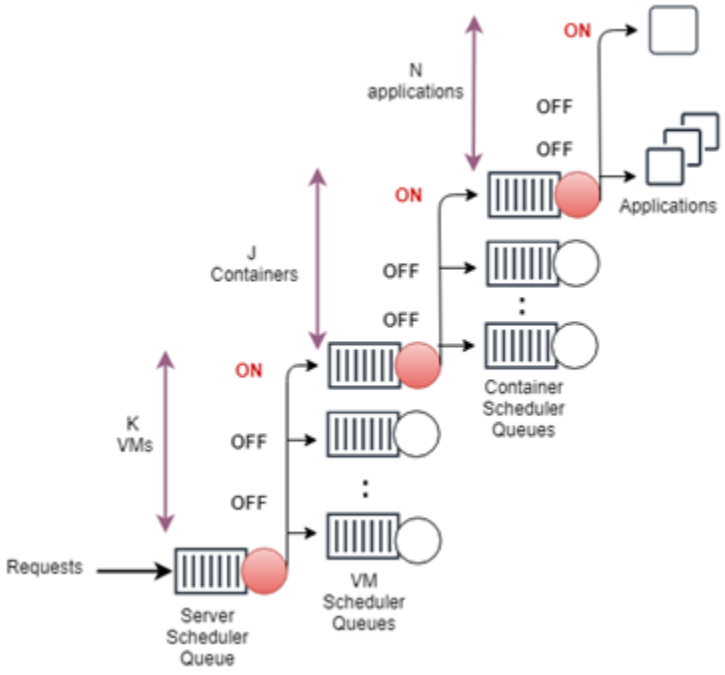


Figure 3.16: Schedulers' queuing network in a server

It is important to note that the fundamental principles and performance characteristics discussed in earlier models remain applicable. However, when applying container technology and utilizing container orchestration managers such as Swarm or Kubernetes, there are additional considerations that impact the performance metrics of the queuing model. One significant difference is the enhanced flexibility and agility

offered by containerization. Containers can be easily moved or scaled up, which has a direct impact on the performance of the system and the QNM. Container orchestration managers, like Swarm or Kubernetes, play a crucial role in managing these dynamic operations. When a node fails or becomes heavily loaded, the container orchestration manager detects the issue and takes appropriate action. In the case of a node failure, the manager automatically moves the affected container to another healthy node in the cluster. This migration ensures the continuity of service and minimizes disruptions which can affect performance predictability. By redistributing the containers, the arrival rate to the queues is adjusted, potentially reducing congestion and improving overall system performance. Moreover, container replication is another mechanism employed by container orchestration managers to optimize performance. If a container becomes overloaded or experiences high demand, the manager can create additional replicas of that container. This replication process distributes the workload across multiple instances, mitigating performance bottlenecks and reducing waiting times in the queues. Consequently, the response time of the system is positively influenced as the increased capacity allows for faster processing of requests.

3.2.2.3 Serverless FaaS Architecture

FaaS is a cloud service model where developers can execute individual functions or pieces of code in a serverless environment. This means that developers do not need to manage or provision servers and the cloud provider handles the underlying infrastructure and resource allocation. In this architecture, the execution of functions is triggered by the events, but the specific location where the functions will be executed is unknown. The servers responsible for executing the functions are distributed across nodes within a cloud network zone, and they periodically change their placement, taking the existing queues of requests with them. Compared to containers, the functions have a time constraint for execution, for example, as the time of writing this thesis, AWS has a 15 minutes limit per execution and for IBM functions is 10 minutes. Also, there is a fixed limit on the concurrent execution including ones waiting in queues to run for each function which is 1000 for IBM and AWS which means the number of servers plus items waiting in the queue should be less than 1000. Customers are able to increase these limits by individually requesting higher service levels.

To model a FaaS system using QN, we consider each function as a separate queue in the network. Each queue has a limited capacity to hold incoming requests, repre-

senting the maximum number of requests that can be queued for a specific function. The size of the queue may depend on factors such as the resource availability and the cloud provider's policies. Associated with each queue, there are multiple servers allocated to handle the requests. The number of servers assigned to a specific function can vary dynamically based on the current load on the system. This flexibility allows the system to scale up or down to efficiently utilize available resources and meet the demand. Additionally, in FaaS systems, each server has a time limit during which it remains active and available to process requests. This time limit is typically set by the cloud provider and helps manage resource allocation and optimization. After the time limit is reached, the container running the function becomes deactivated and needs to be restarted which means a new placement within the cloud network. The QNM of FaaS is similar to QNM in Figure 3.10 but each node can have multiple servers as shown in Figure 3.17.

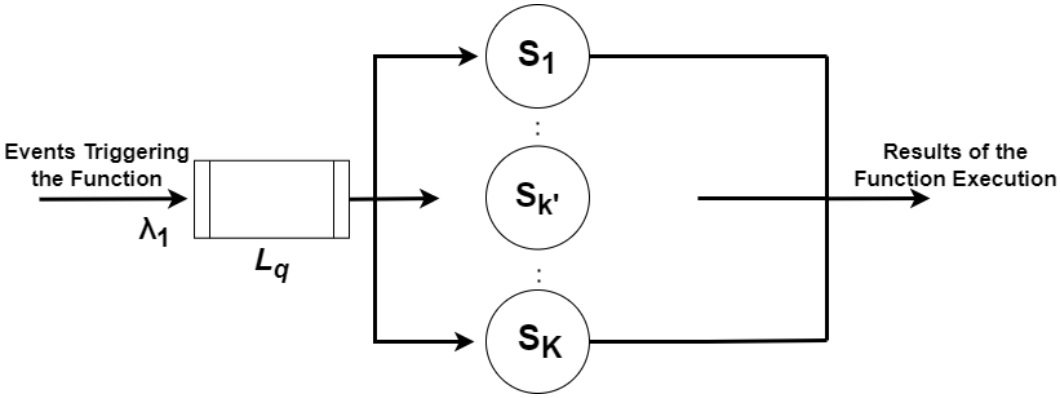


Figure 3.17: A node in the FaaS QNM

One of the technical challenges in analyzing such queueing networks lies in the complexity of the multidimensional Markov chain that describes the system's behavior. However, recent advancements in the analysis of semi-open queuing networks without server movement, using structured generator Markov chains, have helped overcome this difficulty. In a multidimensional Markov chain, each state variable represents a different aspect or characteristic of the system. For example, in a queuing system, one state variable could represent the number of requests in the queue, while another state variable could represent the number of available servers. The values of these state variables change over time based on transition probabilities. The transition probabilities in a multidimensional Markov chain determine the likelihood of moving from one combination of states to another. These probabilities are typ-

ically defined in a transition matrix, where each entry represents the probability of transitioning from one combination of states to another in a single time step.

In a structured generator Markov chain, the transition matrix has a specific form that allows for efficient computation and analysis. The structured generator matrix has a block structure, where each block corresponds to a specific subset of states in the Markov chain. The blocks represent different subsystems or components of the overall system being modeled. The advantage of using structured generator Markov chains is that they allow for the decomposition of the analysis of a complex system into simpler subsystems. Each subsystem can be analyzed separately, and then the results can be combined to obtain the overall behavior of the system. [243; 244].

3.2.3 Single Server Queues Analysis

These characteristics of a single server makes the system simple enough to be modeled using a basic queuing network, such as an $M/M/1/m$ queuing system, where the inter-arrival times and service times follow an exponential distribution and the system has a single server and a finite queue with capacity m. We may also have $M/G/1/m$, $M/D/1/m$ or $G/G/1/m$ where arrivals are Poisson but service times are general or deterministic respectively. Under exponential assumptions a queueing network model with finite capacity can be represented by a Markov process. Let $S = (S_1, ..., S_m)$ denote the state of the network and let E be the state space, i.e. the set of all feasible states. The network model evolution can be represented by a continuous-time ergodic Markov chain with discrete state space E as shown in Figure 3.18 and with transition rate matrix Q .In the next Chapter, we explore if the ergodicity assumption holds for complex software systems.

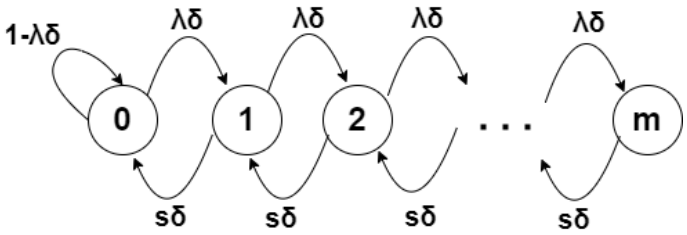


Figure 3.18: Markov chain with discrete state space E

Based on Poisson definition[245]:

$$\begin{aligned} P(exactly\ one\ arrival) &= \lambda\delta + o(\delta) \\ P(zero\ arrivals) &= 1 - \lambda\delta + o(\delta) \\ P(more\ than\ one\ arrival) &= o(\delta) \\ Where\ o(\delta)/\delta \to 0\ as\ \delta \to 0 & \end{aligned} \tag{3.17}$$

Here, δ is a small time interval and o(δ) represents a function that is negligible compared to δ based on Taylor series, as $\delta \to 0$. It can be shown that [245],

$$P[n\ arrivals\ in\ interval\ T] = \frac{(\lambda T)^n e^{-\lambda T}}{n!} \tag{3.18}$$

The transition probabilities in the matrix Q are defined in Equation 3.19

$$Q = \begin{bmatrix} 1-\lambda\delta & \lambda\delta & 0 & 0 & \cdots \\ s\delta & 1-(s\delta+\lambda\delta) & \lambda\delta & 0 & \cdots \\ 0 & s\delta & 1-(s\delta+\lambda\delta) & \lambda\delta & \cdots \\ 0 & 0 & s\delta & 1-(s\delta+\lambda\delta) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{3.19}$$

where the transition probabilities $q(i,j)$ are defined in Equation 3.20

$$q(i,j) = \begin{cases} \lambda\delta & \text{for } j = i+1 \text{ (arrival transition from state } i \text{ to } i+1) \\ s\delta & \text{for } j = i-1 \text{ (departure transition from state } i \text{ to } i-1) \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

The diagonal elements $q(i,i)$ are calculated such that the row sums of the matrix Q are equal to zero, ensuring that the probabilities add up to 1 for each row. The stationary and transient behavior of the network can be analyzed by the underlying Markov process. The performance metrics such as response time and throughput can be calculated using following formulas applying the same notations as explained in Section 3.1.3.

- Inter-arrival times pdf assuming exponential arrivals:

$$f(t) = \lambda e^{-\lambda t} \tag{3.21}$$

- Service times PDF assuming exponential service time :

$$f(t) = se^{-st} \tag{3.22}$$

- Probability of k requests in the queue in equilibrium state:

$$P[k] = (\lambda/s)P[k-1] \tag{3.23}$$

$$P[k] = (\lambda/s)^k \times (1 - (\lambda/s)) \tag{3.24}$$

The Equation 3.24 holds for infinite capacity queues based on the probability axiom $\sum_{i=0}^{\infty} P[i] = 1$. For the case of limited capacity, it follows:

$$P[k] = \frac{(\lambda/s)^k}{k!} \times \frac{1}{1 - (\lambda/s)} \tag{3.25}$$

- Average Server Utilization:

$$U = \lambda/s \tag{3.26}$$

- Average response time of a request:

$$R = \frac{1/s}{1 - U} = \frac{1}{s - \lambda} \tag{3.27}$$

Note: The average response time doesn't depend on scheduling regime but the distribution of response times experienced does depend on scheduling discipline and that what concerns from performance predictability perspective.

- Average number of requests in the system (Little's Theorem[246]):

$$L = \lambda R = \frac{\lambda}{s - \lambda} \tag{3.28}$$

Note: Little's Theorem is true for ergodic systems [247] and λ doesn't have to belong to a Poisson process.

- The amount of time spent in the queue:

$$W_q = R - \frac{1}{s} = \frac{1}{s - \lambda} - \frac{1}{s} \tag{3.29}$$

- The average number of requests in the buffer/queue can be obtained from little's formula:
$$L_q = \lambda W_q = \frac{\lambda}{s-\lambda} - \frac{\lambda}{s} = \frac{U^2}{1-U} \tag{3.30}$$

- For the case of $M/G/1$ from the Pollazcek-Khintichine [248] formula:
$$L_q = \frac{\lambda^2 \sigma_s^2 + U^2}{2(1-U)} \tag{3.31}$$
Where σ_s^2 is the service time variance.

- for the case of $G/G/1$ there is no exact result but the following approximation is popular [249]:
$$L_q \approx \frac{U^2(1+C_s^2)(C_a^2+U^2C_s^2)}{2(1-U)(1+U^2C_s^2)} \tag{3.32}$$
Where C_s^2 is the squared coefficient of service time variation:
$$C_s^2 = \frac{\sigma_s^2}{(1/s)^2} \tag{3.33}$$
And C_a^2 is the squared coefficient of inter-arrival variation time:
$$C_a^2 = \frac{\sigma_a^2}{(1/\lambda)^2} \tag{3.34}$$
There are other approximations which depends on the application.

Note that if $U \geq 1$ in a queue where either the inter-arrival or service time or both following random processes, the queue becomes unstable because the length of the queue and the wait become infinity. If both are constants, $U > 1$ also leads to instability. For these queues we need additional servers for stability.

3.2.4 Multi-server Queues Analysis

Here, we consider the identical (homogeneous) server case in which there are c identical servers in parallel and there is just one waiting line (i.e., the queue is a single-channel queue) similar to the load balancers or scaling up containers in the cloud. Let c denote the number of identical servers:

- The utilization can be calculated:

$$U = \frac{\lambda}{cs} \tag{3.35}$$

- For the $M/M/c$ queue [248]:

$$L_q = \frac{P[0](\frac{\lambda}{s})^c U}{C!(1-U)^2} \tag{3.36}$$

  Where:

$$P[0] = 1/\left[\sum_{m=0}^{c-1} \frac{(cU)^m}{m!} + \frac{(cU)^c}{c!(1-U)}\right] \tag{3.37}$$

  Therefore:

$$W_q = L_q/\lambda \tag{3.38}$$

- For the $G/G/c$ queue, we have the following popular approximation [250]:

$$W_q^{G/G/c} \approx W_q^{M/M/c} \frac{C_a^2 + C_s^2}{2} \tag{3.39}$$

  This works well for $M/G/c$ queues, but does not always work well when the inter-arrival time is not exponentially distributed.

For multi-server queues, it has been shown that data on two moments is usually not sufficient to generate good approximations for the mean waiting time or queue length [251]. When the distributions are known, it is often possible to deduce expressions for these metrics, but they often involve calculus and computational methods. [252] explains the situation in bulk queues and [250] discuss the general case including the Lindley equation [253].

3.3 Summary

In this chapter, we focused on using queueing networks as a prominent analytical modeling technique to evaluate the performance of software-centric systems. We delved into the formal notations and concepts associated with Markov chains and queueing networks. Our modeling approach encompassed a wide range of software systems, starting from the simplest ones represented as single-server systems and extending to the more complex scenarios of large-scale distributed systems deployed

in cloud environments. We explored how these systems can be modeled using queueing networks, taking into account factors such as the number of servers, different arrival processes, service times, and queuing disciplines.

In addition, we extensively explored the various deployment options that are prevalent for distributed software systems, including IaaS, CaaS, and FaaS. Throughout our discussion, we emphasized the challenges and complexities associated with modeling these deployment models within the framework of queueing networks. One key aspect we delved into was the impact of multiple layers of virtualization and scheduling on the performance metrics of these distributed systems. We analyzed how the virtualization of resources, such as servers and containers, as well as the scheduling policies employed in these deployment models, can influence the overall system performance. These factors introduce additional sources of variability and intricacies into the modeling process.

Moreover, we introduced a novel concept known as "queues' on/off switches" to capture the unpredictable nature of these distributed systems. These switches represent random processes that affect the predictability and behavior of the queues within the system. By incorporating this concept into our modeling approach, we acknowledged the dynamic nature of workload processing, considering scenarios where queues may become active or inactive based on various factors such as scheduling, resource availability and demand fluctuations. By examining the complexities arising from virtualization, scheduling, and the introduction of on/off switches, we aimed to provide a more comprehensive understanding of the performance dynamics and predictability challenges faced by distributed software systems. This knowledge serves as a foundation for developing more accurate and realistic analytical models that can effectively capture the behavior and performance of software systems.

Going ahead, in the next chapter, we apply DST to gain a deeper understanding of the behavior exhibited by these queues. By leveraging DST, we aim to uncover insights into the dynamic characteristics, and performance prediction of the queueing systems we have modeled. This will provide a more comprehensive perspective on how these systems operate and evolve over time, contributing to a more nuanced analysis of their performance predictability.

Chapter 4

Bridging Queueing Network Models and Dynamical Systems Theory via the Extended Maurer Model

In Chapter 3, we applied Queuing Network Model (QNM)s as a general and well-known technique for modeling computers and distributed systems, irrespective of scale and complexities. As explained in Chapter 3, under exponential assumptions, a QNM with finite capacity can be represented and solved by a Markov process. As per [223], Markovian analysis to solve QNM, presumes ergodicity holds, specifically via a recurrence property, such as Poincare's recurrence. This equals having a unique steady-state transition matrix. Similar ergodic presumptions are made within stochastic Petri-Net analysis [223]. Additionally, Monte Carlo simulation is generally structured to exploit the ability to equate time-averaged behaviors with the ensemble-averaged behaviors, where this directly requires assuming that (BET) [33] holds. Similarly, machine learning-based approaches innately must assume similar Shannon information between their training data sets and the data to be processed during production deployments, i.e., that ergodicity must holds across the Machine Learning (ML)'s utilized measurement features. Additionally, control theory approaches innately assume that a transition path exists between any operational states x and desired state x' such that a control rule exists to bring the system from state x to state x'. As with Markovian analysis, this requires ergodicity to hold [254].

Early work exists showing that BET held for the first generations of software systems and network protocols [255], but no works appear to exist that show BET continues to hold for modern LDSSs. Moreover, it is well known via Physics that BET holds for systems that can be described by conservation laws as a consequence of Noether's theorem [256], i.e., systems which can be reasonably modeled in terms of Hamiltonian or Laplacian differential equations. Such systems are the commonplace and majority systems of interest within the bulk of traditional engineering domains, i.e., system were conservation of energy, momentum, etc. apply. Within cloud-deployed LDSSs, it is difficult to argue what is conserved as clearly energy is continually pumped into such systems, Shannon information is continually being processed and changed, etc. Hence, Noether's theorem is not applicable, meaning such systems are not likely to be well behaved in the BET sense.

This observation suggests a potential explanation as to why performance predictability appears as a largely solved problem within the academic literature whereas it is a known hard problem within industry-scale systems[53; 186]. Clearly, if the well-developed academic models accurately represented real-world complex and large software systems behaviors then performance predictability should be an industry-solved problem, particularly as knowledgeable graduate students and post-doctoral researchers, transition from academia to industry, i.e., industry is not lacking knowledge of these academic solutions and approaches. The fundamental question as to whether or not BET indeed holds in modern LDSSs would provide a potential explanation of this disconnect between industry and academia's experiences with respect to LDSS performance predictability.

BET is set within the context of DST. As such, to prove (or disprove) if a software system follows BET, a DST approach must be undertaken. Fortunately, the Extended Maurer Model [5] has been already proposed as a bridge between Turing reducible computer systems and DST as defined over σ-finite measure spaces. This allows the QNM of Chapter 3 to be easily mapped into a DST context such that the BET required conditions can then be directly tested while being mapped to QNM behaviors, thereby enabling formal DST-based answers to be developed for predictability problems.

This Chapter begins by formally defining, for the purposes of this research, what is meant by run-time performance measurements within software systems of any scale and complexity i.e. scale agnostic performance predictability. This is then followed by a review of DST within σ-finite measure spaces, the EMM is then briefly reviewed

related to the QNM of Chapter 3. The Chapter continues with analyzing the QNM via the EMM-enabled DST perspective to provide a platform for directly and formally assessing software systems performance predictability. The analysis of which is done in Chapter 5.

4.1 Run-time Performance Measurements

In general, for most software systems, the run-time measurement features of interest are macro-level observables, such as response time, end-to-end delay, lag, etc., that arise through the composite execution of millions to billions of individual processor level instructions across potentially thousands of collaborating processors and/or servers. Such a performance measurement feature will be defined as $x(t) \in \Re$, where t is assumed discrete and, for convenience, $t = 0$ denotes the current time with $t > 0$ denoting future behaviors and $t < 0$ denoting past behaviors. Although microprocessor instruction level executions are deterministic given known instruction inputs, the $x(t)$ macro-level will exist as stochastic processes as $x(t)$ incorporates the collection of all execution timings, input values, error conditions, re-transmissions, etc. of the composite set of instructions that pertain to the given macro-level measurement feature. For example, $x(t)$ could be selected to denote the response time to customer web requests. In this case, $x(t)$ incorporates all of the back-end operations required to service those requests, inclusive of disk read operations, data base look-ups, web page population events, etc. Clearly, at this macro-scale, the $x(t)$, is a stochastic process, outside of all but the most trivial systems and environments.

Generally, performance measurements of interest are obtained by calculating the measure's average, e.g. average response time, over a specific time window W [257]. This is due to software systems operating in event-driven manner, where actions occur only when events arrive to be processed. The selection of the window size W, or equivalently the sampling time T, depends on the rate at which events arrive in the system. It is desirable for W to be set in such a way that during normal operation, events are almost always present within the selected averaging windows. For instance, setting W to $10^{-9}s$ would likely result in most windows being empty of events. In general, within industry settings, appropriate values for W would be on the order of seconds to minutes to hours, reflecting the expected arrival rate of events into the system.

Assume there exists a collected past history of $x(t)$ measurement for $t \in [-T, 0)$ and denote this composite set of past observations as:

$$x(T^-) = \{x(t) \mid \forall t \in [-T, 0)\} \tag{4.1}$$

As $x(t)$ is a stochastic process, the goal is not that of standard time series prediction. More formally, in standard time series prediction, it is assumed an estimate $\hat{x}(t) = g[t|x(T^-)]$ can be produced of $x(t)$ from the collection of past history provided via $x(T^-)$, i.e. via constructing a Kalman filter, etc. The second goal would then be to construct $\hat{x}(t)$ such that,

$$0 < |\hat{x}(t) - x(t)|^2 < \epsilon \qquad \forall t \in [0, T^+], \tag{4.2}$$

Assuming a standard L^2-norm is applied. Instead, for software system predictability, the goal is to use $x(T^-)$ to generate a sufficient accurate estimate of $x(t)$'s PDF designed by $\hat{p}_{x(t)}(x|x(T^-)$ such that it then provides sufficiently good (or accurate) estimate of $x(t)$'s true probability density over T^+. More formally if

$$p_{x(t)}[B_{lower}(t) \ < \ x(t) \ < \ B^{upper}(t) \mid \forall t \in T^+] \ \geq \ 1 - \epsilon \tag{4.3}$$

Then

$$\hat{p}_{\hat{x}(t)}[B_{lower}(t) \ < \ x(t) \ < \ B^{upper}(t) \mid \forall t \in T^+] \ \geq \ 1 - \epsilon \tag{4.4}$$

where $\epsilon > 0$.

Pragmatically, industry tends to be more concerned with the tail behaviours of $p_{x(t)}(x)$ than the true full shape of this distribution, as the tail behaviors denote the portion of customers receiving poor performance and, therefore, for example, posting negative social media reviews, comments, and ratings for their software systems and services.

Following standard industry limit (or range) checking approaches [257], upper and lower bounds can be placed on $x(t)$ to denote when $x(t)$ has left the region between the denoting acceptable performance for the given measurement features. In general, for many common software performance features only the upper bound will be of interest given it denotes problematic system performance and, generally, the lower bound would be zero. For example, if $x(t)$ denotes delay then the upper bound denotes those users who have received poor download speeds, whereas the lower bound denotes those who have good experiences. Similarly, if $x(t)$ denotes response times

then the lower bound denotes those users who have experienced exceptionally fast responses, whereas the upper bound denotes those experiencing poor performance levels. On the other hand, if the measure of interest is the system's throughput then the situation reverses with the now upper bound denoting the users who have received an excellent service, where as the lower bound now specifies those users who have received poor performance. In this case, the lower bound is more interesting to watch than the upper bound.

As we will see in Section 5.1, the formal nature of what $x(t)$ can measure can be defined via the EMM as any σ-finite measurable feature that arise through any information processing transform (or set of transforms) that can occur within the composite Turing-reducible model of the cloud-scale computing environment in which the given software system is deployed and executing.

This research focuses on normally behaving systems under typical workloads, as predicting performance in these standard conditions is industry's primary concern. Systems in overload or starvation conditions or under substantial targeted attack are outside intended scope of this research.

4.1.0.1 A Predictable Performance Feature

A software performance measurement feature $x(t)$ is defined as stochastically predictable over some future time period $T^+ = [0, t^+)$, where $t^+ > 0$, with respect to the chosen bounds $B_{Lower}(t)$ and $B^{Upper}(t)$, where

$$-\infty < B_{Lower}(t) < B^{Upper(t)} \tag{4.5}$$

And based on the available past history $x(T^-)$ if and only if it can be shown that,

$$\hat{p}_{x(t)}(\ x(t) \in [B_{lower}(t), B^{upper}(t)]) \geq 1 - \epsilon \tag{4.6}$$

where

$$\hat{p}_{x(t)}(x(t)) = G[x_k(T_k^-)|k = 1..K] \tag{4.7}$$

where $x_k(T^-)$ denotes the available past histories of k instances of the given software system and $x(t)$ is observed from the system's current running instance. More particularly, only being able to predict $x(t)$ for specific time instances (or time periods) and/or only for specific run-time instantiations of a given software system is insufficient from an industry perspective, i.e., potentially losing performance predictability

when a software system is re-instantiated into a cloud regime is not an useful industry characteristic.

It should be noted that, in general, the distribution $p(x(t))$ will not be known and instead must be estimated based on the available past history $x(T^-)$. The performance predictability definition introduced in Chapter 1 denotes a weaker form of predictability than is generally sought in, for example, in time series analysis methodologies, but it is appropriate to this research as it suffices to support industry's software system's performance predictability needs.

4.2 Dynamical Systems Theory

Dynamical Systems Theory (DST) provides a mature formal framework for modeling how systems behave over time, as it was developed within Physics and Mathematics to modeling complex systems [258]. More particularly, a DST is denoted as a mapping, $T : \Omega \rightarrow \Omega'$, occurring over the spaces of possible events, Ω and Ω'. Generally, T may define a time shift mapping, denoting how a system evolves over time, or an ensemble mapping, defining the relationships between the results seen in different experimental runs (or system instantiations), or a spatial mapping defining the relationship existing between measurements taken in different locations within a system. For the purposes of this research, the first (time shift) and second (ensemble) mappings will be considered as these underlying the formal definitions of statistical stationarity and statistical ergodicity that are of interest within this research and will be discussed in details in Chapter 5. For notational simplicity, Ω is generally assumed to denote a generalized event space such that Ω' can be replaced with Ω such that $T : \Omega \rightarrow \Omega$. In mathematics this allows for a significant simplification of the notation as the Ω need no longer be specified. Hence, the n^{th} event space can just be written as $T^n[\Omega]$. More formally, a dynamical system consists of following components:

1. Ω : A universal space of events such that $|\Omega| < \infty$
2. Σ : A σ-algebra on Ω such that the pair (Ω,Σ) is a Borel space β
3. $T[.]$: A Transformation $T : \Omega \rightarrow \Omega$ denoting a mapping between event spaces
4. $\mu(.)$: A σ-finite measure on Ω such that $\mu : \Omega \rightarrow \Re^+$

Of interest in DST, is how the measure $\mu(.)$ changes with application, (or repeated application) of the transformation T. More particularly, a mapping $T[.]$ is defined to

be ***measure preserving*** (or alternative, the measure $\mu(.)$ is measure invariant with respect to $T[.]$) if for the dynamical system $T : \Omega \to \Omega$, the measure $\mu(.)$, and for all events $A \in \Omega$ if and only if

$$\mu(T^{-1}[A]) = \mu(A) \tag{4.8}$$

Where $T^{-1}[.]$ denotes that pre-image of A. $T^{-1}[.]$ equates to the inverse of $T[.]$ if and only if $T[.]$ is a bijective transform, i.e., one-to-one and onto. As per [33], all stochastic processes have an equivalent DST representation. Hence, an observable performance measure, $x(t)$, can be described in terms of a dynamical system through defining the space of events Ω containing events A such that $A = [A_{min}, A_{max}]$ $\forall A \in (-\infty, +\infty)$. The particular sets A of interest are the ones that produce the cumulative density function (cdf) of $x(t)$ as shown in Figure 4.1 such that,

$$\hat{P}[A_{min} < x(t) < A_{max}] = \hat{P}[x(t) < A_{max}] - \hat{P}[x(t) < A_{min}] \; \forall A \in (-\infty, +\infty) \tag{4.9}$$

If the measure $\mu(.) : \Omega \to \Re^+$ is normalized to meet the Axioms of Probability then it gives the pdf (or cdf) of $x(t)$. As they are related to each other through integration and differentiation either may be used. Within σ-finite measure spaces this normalization of $\mu(.)$ can always be done, hence, it is assumed in Equation 4.10.

$$\mu_{x(t)}(A) = p(x(t) \in A) \quad \forall A \in \Omega \tag{4.10}$$

To achieve the desired property where the measure $\mu(.)$ remains invariant, it is necessary for the underlying software system that generates the performance measure $x(t)$ to exhibit both stationarity and ergodicity such that the pdf/cdf doesn't change over time or with different runs of the system. Ergodicity and stationarity are related through Birkhoff's Ergodic Theorem (BET), discussed in more detail in Chapter 5 Theorem 1, states a random process is ergodic if and only if Equation 4.11 holds.

$$\lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} 1_A(T^n[x]) = \mu(A) \tag{4.11}$$

Within Chapter 5, the equivalency theorem of Peter Walter [254] for ergodicity within σ-finite dynamical systems will be used to directly test the Right-Hand-Side (RHS) of BET, within the context of software systems. The Left-Hand-Side (LHS)

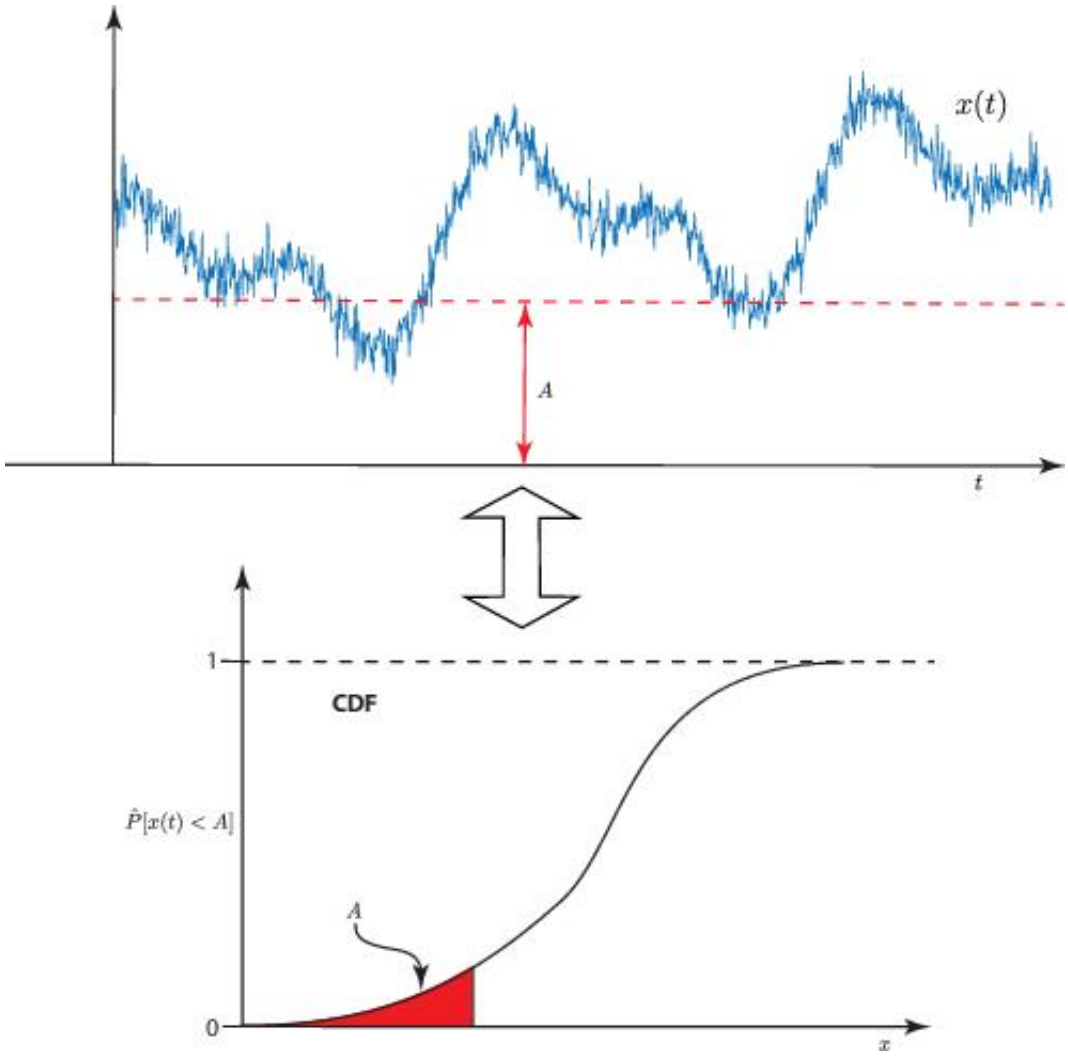


Figure 4.1: The cdf of performance measure $x(t)$.

will be directly tested via the formal dynamical system's definition of stationarity within σ-finite dynamical systems.

In the remaining sections of this chapter, we focus on the process of modeling software systems as dynamical systems while maintaining Turing reducibility. To achieve this, we employ the Extended Maurer Model (EMM) [5], which allows us to capture the scale-agnostic dynamic behavior of software systems while preserving their Turing reducibility of the model.

4.3 Review of Extended Maurer Model

The EMM is a formal modeling framework designed to represent modern computers and networks. It is an extension of the existing Maurer computer model [259], a Turing-reducible model of computer operations. The EMM framework builds upon the foundational components of the Maurer model and introduces additional structures to facilitate the modeling of modern programs, concurrency, multiple processors, and networks. As demonstrated by [5], EMM retains its Turing-equivalent nature, making it suitable for modeling complex programs like modern virtual machines, web browsers, cloud computing regimes, etc. It also provides a scale-agnostic approach to software system run-time performance modeling.

This provides a significant contribution as, to the best of our knowledge, provides the first formal connection between Turing-reducibility and measure theory. As a result, it enables a formal explanation of the behaviors of modern computers introduces dynamical systems theory. We'd like to note that the original Maurer model lacks essential components required to represent modern computers such as programs, security policies, and concurrency, among others.

The Maurer model is primarily concerned with modeling computers in terms of the effects instructions execution have on how the information stored in a computer's memory change over time as the computer executes its instructions. This memory encompasses the full variety of components capable of holding or storing data (e.g., RAM, CPU registers, hard drives, disk drives, hard-coded memory). Given the finite nature of memory, the set of all possible memory states is also finite. Hence, if Ω_t denotes the computer's memory space of events at time t then $|\Omega_t| < \infty$ for all t. The Maurer model therefore denotes a σ-finite measure space describing how instruction executions enact changes in the memory space.

In this thesis, our focus lies on the aggregation of memory locations across servers responsible for maintaining application-level queues. Therefore, only these specific parts are modeled using EMM. Important to Maurer's definition, instructions include the values of their operands. Hence, $ADD\ R1, R2$ denotes a different instruction than $ADD\ R2, R3$ under Maurer's definition. This critical distinction is what allows the Maurer computer model building a bridge into DST analysis.

4.3.1 The Probabilistic View of Memory, M

Most of the modern computing systems are comprised of of detachable components such as USB devices or permanent components such as CPU registers. Given that all these components are included within the model's memory, it becomes necessary for the memory to exhibit time-dependent characteristics especially to cover the detachable components. Therefore, in the EMM, the Maurer model's definition of memory M is extended as follows. At any time instant t, the EMM's memory is denoted as $M(t)$ and is given by Equation 4.12 [5],

$$M(t) = \{m_k | k = 1, ..., N_M(t)\} \tag{4.12}$$

where N_M is the number of existing memory components, each m_k is disjoint with any other elements $M(t')$ and t is discrete. It is assumed that in general $M(t) \neq M(t')$. The detailed structure of M can be found in [5]. This extension of the memory allows the modeling of the dynamic nature of the memory of modern computers as well as the information flows from outside world into and out of the computer. For digital computers $|\Omega_{M(t)}| = 2^{M(t)}$ where $\Omega_{M(t)}$ defines the space of all possible memory states within $M(t)$ at time t. Instruction executions change memory, to describe such changes, Maurer introduced the input and output regions of an instruction i denoted as $IR(i) \subseteq M$ and $OR(i) \subseteq M$, respectively.

Define by $S(t)$ the power set of all possible events $M(t)$ can represent. Then, an instruction i executes at time t and $s_1 \in S(t)$ is the state of $M(t)$ impacted by i's execution at time t, then,

$$s_2 = i(s_1) \tag{4.13}$$

Where $s_2 \in S(t+\tau)$ and s_2 is the change in the memory caused by i's execution on s_1 which is assumed to take τ time steps (on clock cycles) to complete. Therefore, $S(t)$ represents the universe of events that the given computer can represent and process such that,

$$\Omega_{M(t)} = S(t) = \mathbb{P}[M(t)] \tag{4.14}$$

where $\mathbb{P}[.]$ denotes the power set. For modern computers, $|S(t)| \approx 2^{16,000,000,000} \approx 10^{16B}$ assuming 16 Gigabytes of memory which denotes an extremely large event space given that classic physics gives way to statistical physics as event space eclipse 10^{20} states. As will be highlighted in Chapter 5, the DST approach's power lies in

its ability to reason across these full and extremely large spaces, i.e, it enables a scale-agnostic analysis approach.

$OR(i) \subseteq M$ is the set of all memory elements which their content have changed due to the execution of i. Whereas, $IR(i) \subseteq M$ defines the set of all elements of M that affect $OR(i)$. Figure 4.2 shows an example of how the state of the memory changes from s_1 to s_2 due to the execution of i with the input region $IR(i)$ and output region $OR(i)$ as indicated. For notational simplicity it is assumed that $M(t) = M(t' = M \ \forall \ t, t'$. Maurer also addressed situations where instructions lack input or output regions by introducing the identity (no-op) instruction.

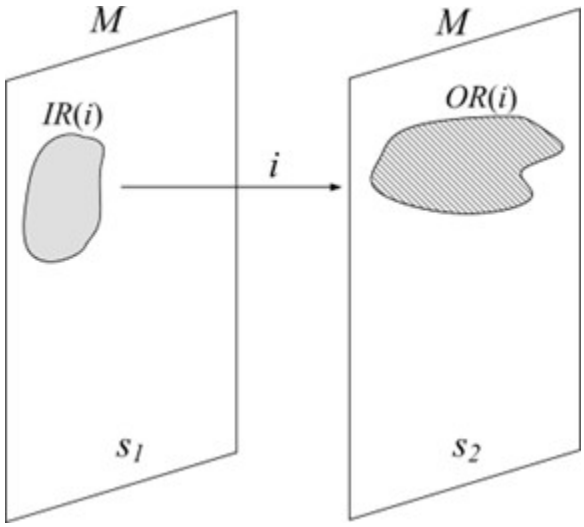


Figure 4.2: Input and Output Regions of instructions [5].

Consider a time interval denoted as $\tau = [-T_1, T_2]$, which starts from some time $t = T_1$ in the past when the computer was powered on, and extends up to some finite time $t = T_2 < \infty$ in the future. For simplicity, assume that $t = 0$ denotes the current time. Consequently, the interval $\tau^- = [-T_1, 0]$ means the computer's historical events, while $\tau^+ = [0, T_2]$ means its future operations. As the time variable t is assumed to be discrete due to the use of digital computers, i.e. closed systems, we can define the collection of all memory transitions occurring within the time span τ as follows:

$$S(\tau) = \{M(t)|t \in \tau\} \tag{4.15}$$

For each $t \in \tau$, $|M(T)| < \infty$ and in the limit as $x \to \infty$, it is the case that $M(\tau)$ for $\tau = [0, T_x]$ defined as finite measure space. Here, $|M(T)|$ is finite where $|.|$ indicates the cardinality of the set. Given the variable nature of the EMM's memory, the set

of all the feasible states of the EMM can also change over time. Specifically, the state of $M(t)$ at a given moment t is represented as $s(t) \in S(t)$, where $S(t)$ corresponds to the set of all possible system states at time t.

It's important to note that the state of the system can only be known for past time, i.e. $\forall\tau \in [-T_1, 0)$, as such $S(\tau^-)$ represents the past history of the system. Whereas, for future times $S_1(\tau^+)$ for all $\tau = [0, T_2]$, is probabilistic in nature. This distinction between $S(T^-)$ and $S(T^+)$ is shown in Figure 4.3. As such there exists a probability distribution (or cdf) that describes the likelihood of $S(T^+)$ future states and over which Shannon Information can be computed. By comparison for $S(T^-)$ this probability distribution has collapsed to a delta function around just the events in $S(T^-)$ that did occur.

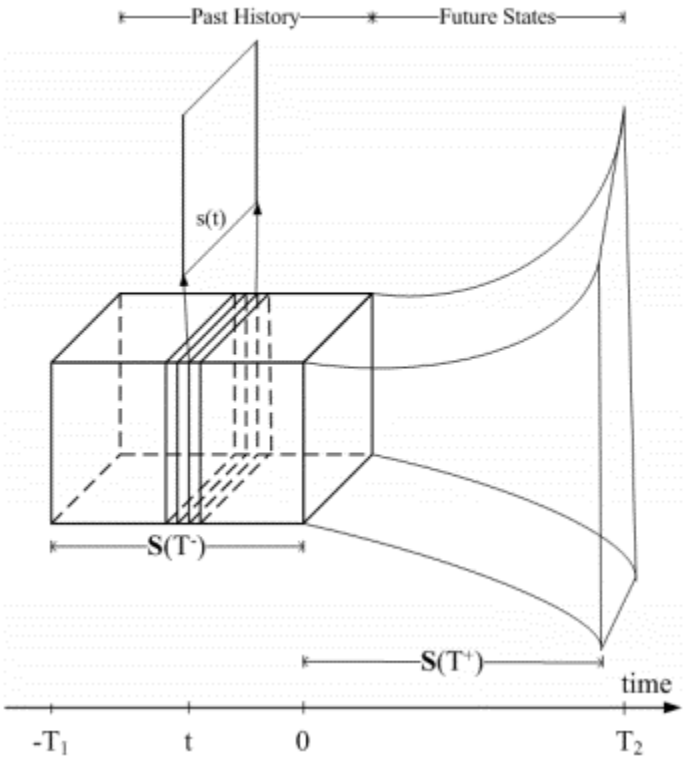


Figure 4.3: The EMM at past, current and future times [5].

It's important to recognize that this probabilistic perspective of the future states of the EMM is consistent with real-world systems behavior, where future states are generally not known. For instance, the outputs of programs remain uncertain until they receive inputs, which clearly can differ from one run to another. Furthermore,

various factors such as attacks, faults, etc., can come into play, with their occurrences not being predicted. Thus, even though the instruction set of the EMM follows deterministic principles in the sense that the state of $OR(i)$ is known given the state of $IR(i)$ for all $i \in I$, the actual states of $IR(i)$ and the execution times of i are generally not deterministically known outside of quite trivial systems. Hence, $S(T^+)$ is probabilistic and describes a stochastic process, which as per [33] has a dynamical system model.

4.3.2 Multiple Control Units

The computer model proposed by Maurer and extended in [260], focused on systems within a single control unit responsible for executing all instructions. Consequently, this model lacks the capability to support the simultaneous execution of multiple instruction sets. In order to enhance multiprocessor support capacity for modeling modern systems, Maurer's model can be extended such that the single control unit C previously defined in [260] is replaced with a collection of $N_C(t)$ control units, denoted as $C(t)$ and specified as follows:

$$C(t) = \{C_k | k = 1, 2, \ldots, N_C(t)\} \tag{4.16}$$

Where $N_C(t)$ denotes the number of control units within the system at time t, and each $C_k \in C(t)$ represents a separate control unit as defined in [260]. Furthermore, $C(t)$ is assumed to contain all processing components capable of affecting the states of $M(T)$. Consequently, CPUs, GPUs, DMA controllers, and similar elements are all included within the set of control units denoted by $C(t)$. Due to this diverse range of control units, each $C_k \in C(t)$ might have a distinct collection of instructions. As a result, the whole instruction set of the EMM is extended as well such that,

$$I(t) = \bigcup_{k=1}^{N_C(t)} I_k \tag{4.17}$$

Since it's assumed that all control units operate in parallel, simultaneous modifications to memory are possible. This is in contrast to the original Maurer model, in which only single, sequential instruction execution were allowed. Generally, various instructions are expected to possess distinct execution times, with the concept of

atomic execution time for instructions not being assumed. Consequently, the execution of an instruction might span multiple time slots t as discussed in [261].

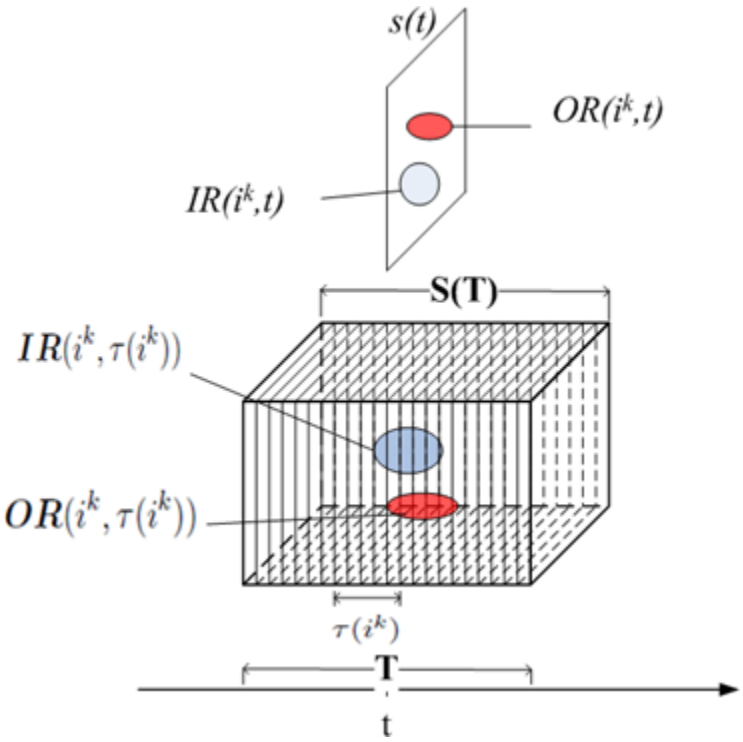


Figure 4.4: The execution of an instructioni^k and the spatial-temporal sub-spaces representing $IR(i^k)$ and $OR(i^k)$ [5].

In Figure 4.4, the system is depicted across the time span T (referred to as $S(T)$). Within this illustration, a snapshot is taken at a specific time instance $t \in \tau(i^k)$, denoted as $s(t)$. Here, i^k represents the execution of an instruction i on control unit $C_k \in C(t)$, and $\tau(i^k)$ denotes the time interval required for the execution of i^k on C_k. This snapshot portrays the immediate input and output regions of i^k as $IR(i^k, t)$ and $OR(i^k, t)$, respectively. Additionally, the image showcases the spatial-temporal regions denoting $IR(i^k, \tau(i^k)))$ and $OR(i^k, \tau(i^k))$.

To align with standard computer architectures, it is assumed that for all $t \in \tau(i^k)$, the intersection of $IR(i^k, t)$ and $OR(i^k, t)$ is the empty set. This restriction is necessary to enforce the defined causality properties of the EMM. Furthermore, the definition of Maurer instructions [259] ensures that this is consistent with standard models of instruction executions.

To illustrate, let's consider the simultaneous execution of two instructions, i^1 and i^2, taking place on control units C_1 and C_2 respectively, within the EMM. As

depicted in Figure 4.5, at a given time t during their execution, it's not necessary for $IR(i^1, t)$ and $IR(i^2, t)$ to be completely separate (i.e., their input regions can overlap). However, it's evident from the Figure 4.5 that $OR(i^1, t) \cap OR(i^2, t) = \phi$, indicating that the output regions $OR(i^1, t)$ and $OR(i^2, t)$ must not overlap. In general, the hardware of computer memory ensures that concurrent write operations cannot target the same memory location(s), i.e. via standard locking constructs.

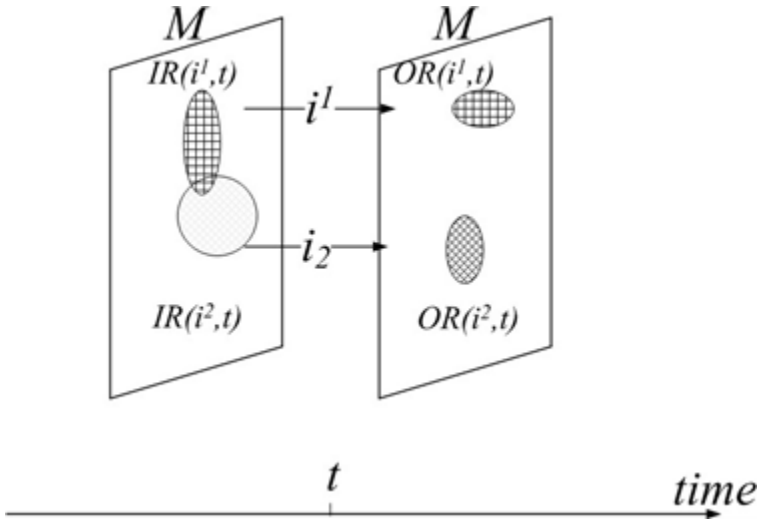


Figure 4.5: The concurrent execution of instructions [5].

4.3.3 EMM's Software Components or Composite Set of Instructions

To represent programs, the EMM employs the established software engineering notion of software components, which are callable collections of instructions with distinct functionalities [262]. Generally, a software component is characterized as:

> a block of instructions and internal data that exists as a black box that performs specific input-to-output mappings over given time frames [263].

While a simple program stands as a standalone software component, a complex program can manifest as a collection of components that may execute concurrently.

Let I_J be a sequence of instructions that is defined by,

$$I_J = (i_j | j = 1, 2, ..., J), \tag{4.18}$$

Here, J is an index set over I with a length $J > 2$. The instructions within I_J are executed sequentially in accordance with their indexed order. It's important to recognize that, just like in real-world systems, the instruction sequence can involve instructions that are executed across different control units. For example, a sequence of instructions could involve in reading data from the disc and sending it via email to a remote recipient. This sequence encompasses instructions handled by disk controllers as well as those managed by network adapter controllers, and so on. The execution of the instruction sequence I_J leads to a sequence of alterations in the system's state. Without any loss in generality, the overall state transformation can be viewed as a single transition from the initial system state, denoted as s, to the resulting system state, designated as s'. This transformation can be represented as:

$$s \equiv I_J(s) \equiv i_J(i_{(J-1)}(\ldots(i_2(i_1(s))))) \tag{4.19}$$

Let's now examine the execution of the previously defined instruction sequence I_J within a system featuring multiple control units, spanning a time period τ that is a subset of T ($\tau \subseteq T$). Naturally, there could be concurrent execution of the individual instructions from I_J across distinct control units. To encompass the temporal aspects, we can define the execution trace of the composition of I_J as follows:

$$trace(I_J, \tau) = \{< i_j, C_k, \tau(i_j) > | i_j \in I_J, C_k \in C, \tau(i_j) \subseteq \tau\} \tag{4.20}$$

Here, $\tau(i_j) \subseteq \tau$ represents the time interval during which instruction $i_j \in I_J$ was executed by control unit $C_k \in C$. Consequently, $trace(I_J, \tau)$ captures the dimensions associated with the simultaneous execution of the instructions within I_J throughout the system. This is achieved by specifying the control units responsible for executing the instructions, along with the corresponding time intervals over which these instructions were executed on each respective control unit. Consequently, $trace(I_J, \tau)$ accurately describes the control units that participated in the execution of the given instruction sequence.

Furthermore, it's important to note that there is no explicit requirement for the control units to be synchronized or sequential. Within the EMM, the computer's operating system itself can be modeled as a program. Consequently, the responsibility for ensuring the accuracy of parallel high-level instruction execution sequences lies within the OS, and is achieved through the implementation of standard semaphore

or lock constructs. It's worth noting that such concerns exist at significantly higher levels than the atomic one-at-a-time instruction execution processes originally defined by Maurer. Importantly though Maurer's original model does describe a dynamical system as it designed how instruction executions generate a mapping between sequences of event spaces. The input and output regions resulting from the execution of I_J within the time interval τ, denoted as $IR(I_J, \tau)$ and $OR(I_J, \tau)$ respectively, can be formally defined as follows:

$$\begin{aligned} IR(I_J, \tau) &= \bigcup_{\forall i_j \in trace(I_J,\tau), \forall \tau(i_j) \subseteq \tau} IR(i_j, \tau(i_j)) \\ OR(I_J, \tau) &= \bigcup_{\forall i_j \in trace(I_J,\tau), \forall \tau(i_j) \subseteq \tau} OR(i_j, \tau(i_j)) \end{aligned} \tag{4.21}$$

Where $IR(i_j, \tau(i_j))$ and $OR(i_j, \tau(i_j))$ represent the input and output regions respectively for $i_j \in I_J$. It's important to highlight that, based on its definition, $trace(I_J, \tau)$ is deterministic for past times and probabilistic for future times. Given that $t = 0$ represents the present time, the segments of $IR(I_J, \tau)$ and $OR(I_J, \tau)$ within the interval T^- (which encompasses past and current times) exist as subsets of $S(T^-)$. Conversely, the segments of $IR(I_J, \tau)$ and $OR(I_J, \tau)$ within the interval T^+ (which depicts to future times) exist within the probabilistic space $S(T^+)$.

Thus, the past history of the computer's execution is deterministic while its future states are probabilistic as is consistent with the behavior of real-world computer operations. Consequently, the structure $trace(I_J, \tau)$ defines the spatial-temporal progression of the subset of $S(T)$ linked to the execution of I_J, encompassing all feasible concurrency arrangements stemming from the utilization of multiple control units to execute segments (or the entirety) of the I_J instructions, i.e. it provides a scale-agnostic representation that can encompass modern computers and their operands.

Now, let γ represent a software component defined as:

$$\gamma = I_{J_q} | q = 1, \ldots, Q \tag{4.22}$$

I_{J_q} refers to an instruction sequence, as established in Equation 4.18 and 4.19, representing potential input-to-output mappings achievable through the execution of γ. Equation 4.22 outlines that γ embodies all the different control paths the software component can take during execution. This includes not only the ideal scenarios but

also accounts for possible errors, various inputs, and any other factors that might impact how the software operates during its execution.

Certainly, when γ is executed, it follows a specific sequence of instructions that dictates its behavior. However, within each individual execution, only one out of the many possible sequences of instructions will be chosen and executed. The specific sequence of instructions that is executed during any instance of γ depends on the inputs that are provided to γ at that run's time. Different inputs can lead to different paths through the set of possible sequences of instructions, determining which I_{J_q} sequence is followed in that specific execution.

4.3.4 Turing Reducibility

Maurer proved that its original Maurer model was Turing reducible [264] with Church's thesis [265], highlights that anything computable is also computable by a Turing machine. Therefore, the EMM must be shown to maintain Turing reducibility if the EMM is to be used as a general model for software systems. The fundamental distinction between the Maurer computer and the EMM lies in the mode of instruction execution. In the Maurer computer, instructions are restricted to sequential execution, occurring one after another within distinct time slots. As detailed in Section 4.3.2, the EMM permits concurrent execution of up to N_C instructions during each time slot.

In [5], it was shown that the EMM's parallel (or concurrent) instruction executions can always be reduced to an equivalent set of sequential instruction executions. Moreover, this notion of "sequential equivalence" is the common notion of "correctness" as applied to parallelization, i.e. this approach is standard to computing theory. Therefore, the EMM can always be trivially reduced to an equivalent Maurer model. Hence, Maurer's original Turing reducibility proof innately also extend to the EMM. As such, the EMM provides a general computer model. But, importantly, one that is i) scale-agnostic and ii) admits dynamical systems analysis approaches, i.e., the EMM provides the critical unified bridge between Turing-reducible computer models and DST-based performance analyses.

4.3.5 Modeling Virtual Machines and Containers

As the EMM is Turing reducible, it provides a foundation for modeling various programs and computations. Nevertheless, a particular subset of programs demands

extra attention and more thorough explanation of their EMM modeling. These programs are the popular virtual machines (VMs) and containerization. Generally, a virtual machine (VM) is a software application that simulates hardware to enable the execution of an unaltered guest OS in an isolated environment, on top of the already existing operating system, known as the host OS. Consequently, when a VM operates within the EMM, it can be viewed as a program is executing within the model. Therefore, based on [5], the most effective way to model VMs within the EMM is by employing a hierarchical EMM model. In this configuration, the primary computer hosting VMs is considered as the foundational EMM at the level 0 with the individual VMs then being modelled via separate nested EMMs.

Moreover, if the group of programs within a VM contains yet another VM or a container, an extra third layer of models would be required to simulate the program execution within that nested VM/container. Clearly, this approach can extend to arbitrary level of hardware abstraction. Within this structured setup, complex programs like VMs, containers and browsers can be accurately represented within the EMM, while their essential qualities remain intact. Specifically, VMs perceive virtualized hardware components as being under software control—a complexity that's best represented using nested hierarchical EMMs. It's important to note that this hierarchical EMM approach is also applicable for capturing the operations of multi-tab browsers, as well as other modern computing designs including function-as-a-service, etc.

4.3.6 Modeling Computer Networks and Cloud Computing

Lastly, let's explore the application of the EMM in computer network modeling. Consider a network that, at a given time t, consists of a finite number of computers ($1 < N_{net}(t) < \infty$). Clearly a EMM for each computer can be established. The network can then be designed as the union of all these individual EMMs. In this manner, the network model's memory is the combination of all the individual computers memory with the same principle applying to all the computer's components, instructions, etc. Hence, heterogeneous collection of computers can be trivially modelled, as for example, any computer in cloud-to-edge systems and solutions.

It's important to highlight that in EMM, as described by Elgamal [5], there's no presumption of a single system clock. Each computer's assumed to possess its own clock, where a clock synchronization may or may not exist. Even in the absence

of clock synchronization, as long as $N_{net}(t) < \infty$, there will always be an $\epsilon > 0$ that is smaller than the minimum time difference between any two clock ticks of any two arbitrary systems in $N_{net}(t)$. As a result, ϵ exists and provides the theoretical reference clock necessary for constructing the composite EMM network model. It should be noted that this global reference clock with separated clock ticks is a purely theoretical concept which can not be known (or knowable) in practice, as per [266]. That ϵ exists and $\epsilon > 0$ is though required if the EMM is to represents a σ-finite measure space over a countably infinite sequence of transformations.

In this composite EMM, every element capable of altering information within the network is regarded as a computer within the EMM framework. Consequently, routers and other networking devices are classified as specialized computers within the EMM paradigm, since, by definition, they incorporate EMM modellable control units and executable instructions whether done in hardware or software. Clearly, this extension of EMM for network-based systems is highly versatile and can be employed across a wide variety of networks, including both wired and wireless configurations, as well as mixed networks. As per real-world systems, the number of computers in the network must be finite for all instances t within the defined time interval T. Therefore, EMM can effectively model modern enterprise networks and critical infrastructure networks, encompassing wireless elements and intermittently present sensors (e.g., modern smartphones and tablets), as well as standalone embedded devices. Hence, the EMM provides a general and scale-agnostic approach to modelling of modern computers and network deployments.

It should be noted that this form of the EMM is restricted to modelling to digital components of modern computer systems. Hence, the physical gadget's analogue operational characteristics are outside of the EMM's intended scope. As such, wireless jamming attacks, broadcast capacity issues, etc. are outside of the EMM's scope to reason about, except with respect to what digitally observable information they may provide and/or generate. More details about the modeling of multiple control units, concurrent execution of programs, software components, virtual machines, networks and the proof of Turing reducibility can be found in [5].

4.4 Applying EMM to bridge QNM and DST

In Chapter 3, we demonstrated how we can model software systems in general and LDSS in particular as an QNM. By definition, all queues within a QNM exist in

memory and are executed on by instructions, with the processed events then being passed onto the next queue(s). As such, this can be trivially represented within the EMM as the subsets of the EMM's memory holding the queues and the instruction sets and control unit then need to process each queued event. Clearly any style on complexity of queues on queuing network can be modeled in this way via the EMM. Sections 4.4.1 and 4.4.2 highlight this EMM based QN modeling for basic queues and multi-server systems respectively.

4.4.1 EMM Analysis of Single Server Software Systems QNM

This section analyzes a software system which is modeled as a single server QNM in Chapter 3. By applying the EMM approach to gain insights into the system's behavior via DST.

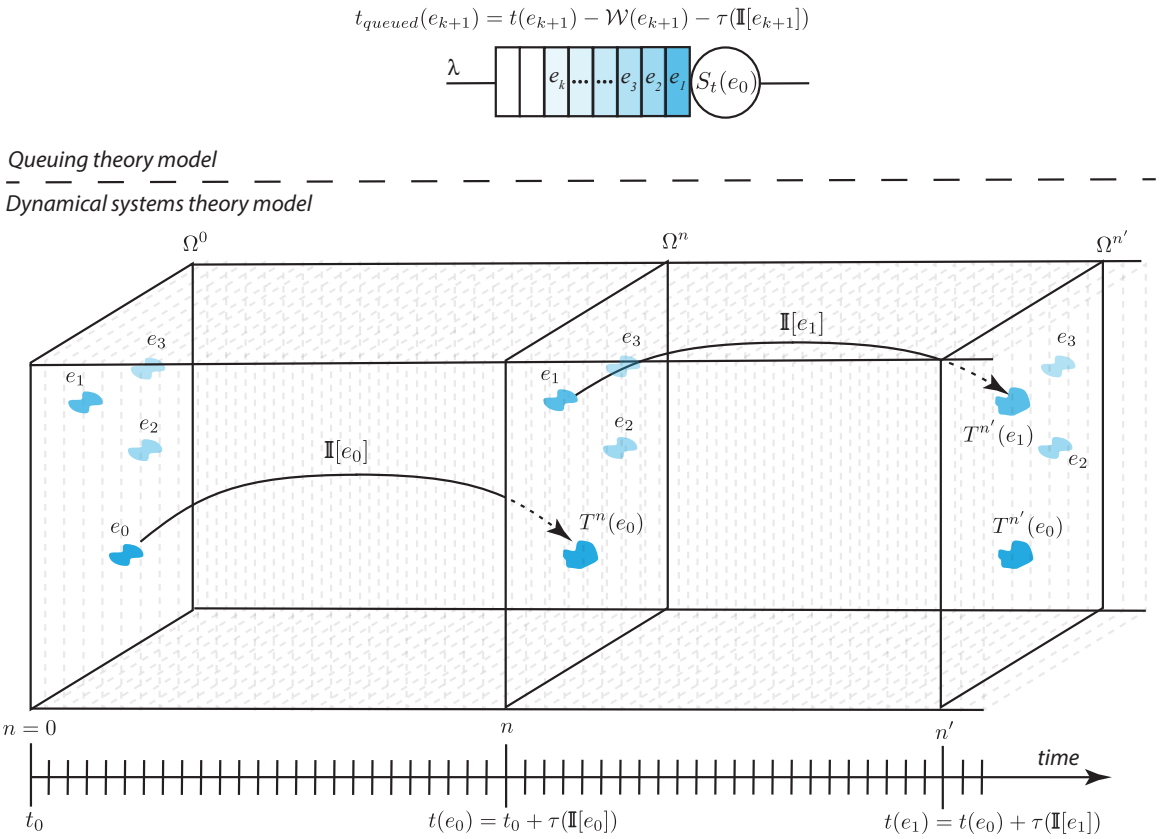


Figure 4.6: Modeling the evolution of a single-server queue over time applying EMM and DST

As shown in Figure 4.6, the single-server queue resides in memory. Each request in the queue represents an event e_k in the corresponding EMM and $\mathbb{I}[e_k]$ denotes the instructions associated with each e_k event. Transformation $\tau(.)$ denotes the time it takes for $\mathbb{I}[e_k]$ to complete where this is stochastic outside of trivial systems. Note that $\tau(.)$ corresponds to the service time $S_t(e_k)$ in a QNM as shown in Figure 4.6 with the difference that $\tau(.)$ includes any interruptions produced by OS schedules. As such, $\tau(.)$ potentially consists of several hidden middle states and transformations in EMM and DST, however, we are interested in the final state where the system has finished servicing the request by executing all the instructions in the $\mathbb{I}[e_k]$ i.e. the event e_k has been fully processed by the queue. The server in the QNM consists of two components to service the request, one is the set of required instructions and the second is the required time to execute those instructions. In the QNM analysis, typically these details are abstracted out as only the average service time is generally considered. Primary difference is that EMM must preserves all detail of all individual event executions (or processing) within each given queue as it describes a dynamical system and its full evolution over time. Within standard QNM analysis, such as provided via Markovian approaches, what is of interest are more general observable statistics or the QN's operation, e.g. average throughput, average response time, average wait time, etc. This is the same distinction that arises between the microscopic physical interactions of atoms and the macroscopic aggregate observables of temperature, pressure, etc. Similarly, within software systems or software-control systems, standard performance measures of interest exist as macroscopic observables e.g. delay, stutter, response time, etc. That exist and arise due to, generally, many billions of individual instruction executions; the microscopic world the EMM describes. We then can explain how this micro-level behaviors contribute to the averages and affect the performance predictability of the system.

In Maurer's original model, each instruction was assumed to execute sequentially allowing time to be inferred but not explicitly stated. Within the extended Maurer model, time must be now explicitly included to both dependent more complex and concurrent instruction executions and to allow for the bridging into DST. More specifically, by e_k, a specific instance of an event that executes on the EMM. The full set of instructions associated with e_k's execution is then given by $\mathbb{I}(e_k)$ with the time period over which e_j executes given by $\zeta(e_j)$.

For notational simplicity, the exact start time of e_j's execution within the EMM is given by $t_0(e_j)$ with $t(e_j)$ then giving the super-scripted absolute time associated

with e_j's execution such that,

$$R(e_j) = t(e_j) - t_0(e_j) \tag{4.23}$$

And

$$t_0(e_j) = Min[\tau(e_j)] \tag{4.24}$$

From a QN perspective, $\tau(e_j)$ is then composed of wait times, service times, and off times associated with all queues involved in e_j's execution. Respectively, denote these times by $t_{wait}[.] \geq 0$, $t_{service}[.] \geq 0$ and $t_{off}[.] \geq 0$ where these can then be super-scripted to denote the portions included by each specific queue such that,

$$\tau(e_j) = t_{wait}[e_j] + t_{service}[e_j] + t_{off}[e_j] = \sum_{k=1}^{K(e_j)} t^k_{wait}[e_j] + t^k_{service}[e_j] + t^k_{off}[e_j]$$

Assuming $K(e_j)$ queues were involved in that specific instance of e_j's execution.

Denoted by $\hat{e_k}$ a class of sufficiently similar events, such as a "web page load" event, or a "data base retrieve, process, and store" event. Then

$$\hat{e_k} = \{e_j | j = 1, ..., J\} \tag{4.25}$$

Importantly, DST and hence the EMM, deal with the specific execution instances of events e_j and not the general class label $\hat{e_k}$ for types of events, as the EMM focuses on how memory changes as instructions are executed.

4.4.2 Modeling Shared Servers in Cloud Environment (IaaS, CaaS and PaaS)

For a distributed system running on shared servers, the components are encapsulated within virtual machines (VMs) or containers and as discussed in Section 3.2.2 can be modeled applying QNM with additional ON/OFF switches associated with queues. The virtual machines and containers can be modeled by EMM as described in Section 4.3.5.

We can analyze the QNM depicted in Figure 3.12 applying EMM by modeling each VM/container as an individual EMM and then all the EMM analysis for tandem queues can be applied here. As shown in Figure 3.15, in the beginning of each time frame, schedulers in various levels decide about the set of active queues. Events

in running queues are processed and then trigger other events to be queued up to be processed in other queues. Also, new requests arrive to the queues according to a random arrival process $\Lambda(t)$ with time average rate λ. Being on a shared resources with ON/OFF switches, a delay $d(t)$ will be added to the service time which complicates the calculation of performance measures in both QNM and EMM. However, in EMM, this situation can be discussed through analyzing $\tau(\mathbb{I}[e])$ which includes the interruption(OFF) times occurring by schedulers. The effect of schedulers on performance predictability will be discussed in details in Chapter 5.

4.4.3 Modeling Multi-server Queues in Cloud Environments (Seamless FaaS)

Compared to other architectures, FaaS has a constraint that affects the performance predictability which will be analyzed through EMM in details in Chapter 5. In FaaS, the functions have a time limit for execution, for example, AWS has a 15 minutes time limit per execution and IBM has a 10 minutes time limit. Also, there is a fixed limit on the concurrent execution including ones waiting in queues to run for each function which is 1000 for IBM and AWS which means the number of servers plus items waiting in the queue should be less than 1000 [267]. Customers are able to increase these limits by individually requesting higher service levels. Therefore, we can model each function with its queue and all the servers running it as an individual EMM with multiple cores as explained in Section 4.3.2. The number of servers assigned to a specific function can vary dynamically based on the current load on the system. This flexibility allows the system to scale up or down to efficiently utilize available resources and meet the demand. The corresponding EMM depicted in Figure 4.7 shows this.

As shown in the Figure 4.7, each function can have several servers, and each server has different capacity and load. Therefore, it is possible to have different execution times for the same instruction set. It should be noted that the EMM depicted in Figure 4.7 is not only modeling loads but is also applicable to any multi-server queue associated with the specific functions running with their respective loads and capacities.

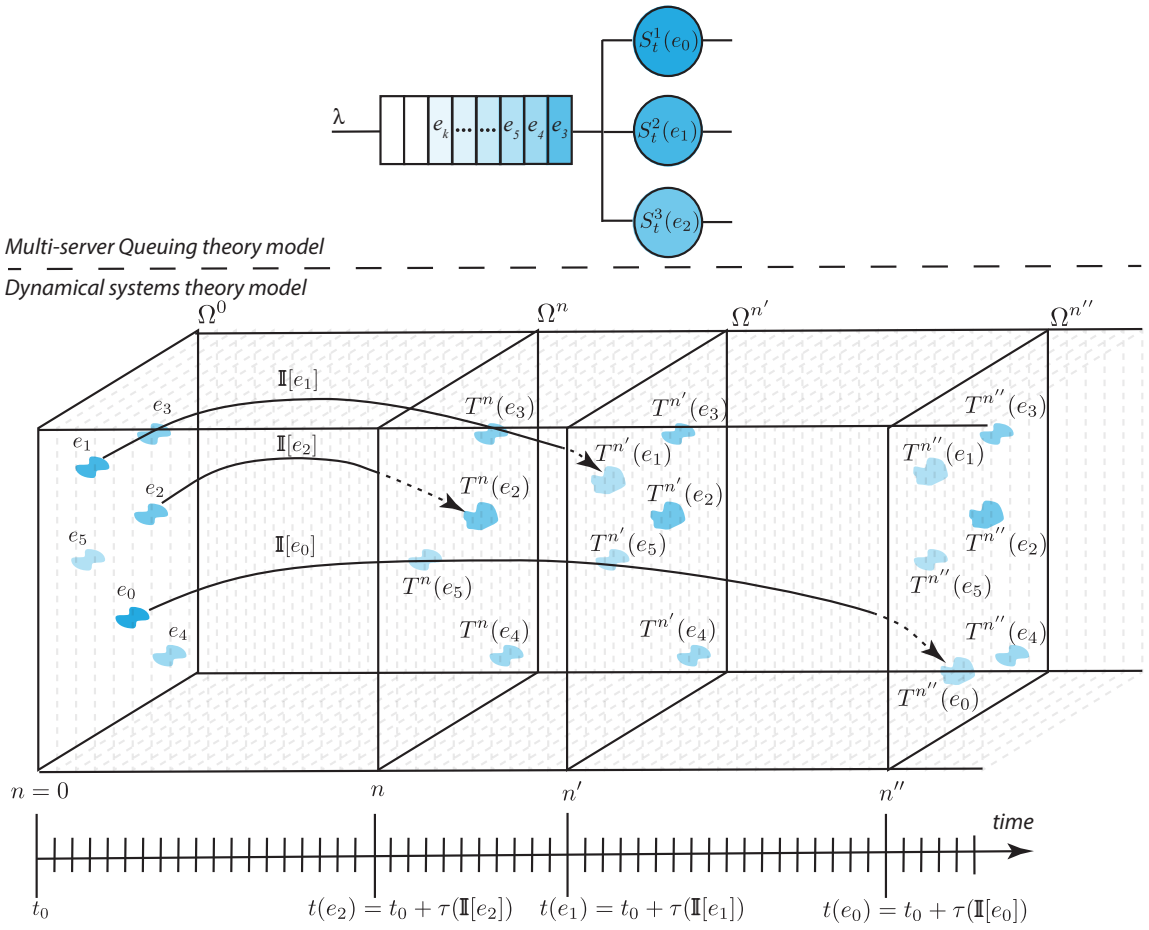


Figure 4.7: QNM evolution over time based on DST for a multiserver queue (e.g. FaaS)

4.5 Summary

In Chapter 4, we explored the utilization of the EMM to model QNMs and their performance based on the DST rules. This enables us to formally analyze software systems for BET-compliance in Chapter 5 by employing DST principles.

In the beginning of the Chapter 4, we cleared how for the run-time performance measurements, we focus on macro-level observables like response time and end-to-end delay. These observables denoted as a stochastic process, $x(t)$, encompass the execution of numerous processor-level instructions across collaborating processors and servers. Generally, in calculating performance measurements, statistics computed over time are typically used for software systems. The chosen window sizes depend

on factors such as event arrival rates, typically ranging from seconds to minutes to hours.

Software system performance measures, therefore, generally involve these $x(t)$ stochastic process macroscopic observables and run-time system features. Moreover, in general, exact time series $x(t)$ predictions are not required. Instead, based on a recording past history of $x(t)$ behaviors, the predictability goal is to accurately assess the probability that $x(t)$ will remain within a defined set of upper and lower bounds for all $t \in [0, T]$, with T denoting some reasonable time into the future. This weaker form of $x(t)$ prediction generally suggests intervals of meeting SLA and low SLO performance requirements, i.e., ensuring subsequent levels of client satisfaction are met.

The connection between the EMM and DST was then established and motivated via the need to assess the above $x(t)$ predictability concerns within a DST context. Chapter 5 will now use the carefully proposed approach to directly assess the necessary and sufficient conditions for BET to hold under QN models and their EMM representations.

Chapter 5

Analysing Software System Predictability and BET-compliance via DST

This chapter provides a detailed formal investigation of the extent to which it is possible to assuredly predict the performance of software-driven systems, specifically, via the EMM and DST approach to model software systems. Importantly, this is done in a scale-agnostic manner, making the results of analysis apply equally to complex large scale cloud-to-edge systems as they do to stand-alone small-scale embedded systems. In part, this scale-agnostic capability is gained via the formal application of QN and DST.

As discussed in Chapter 4, although both academic and industry research have sought to apply various techniques including control theory [63–68] and machine learning methods [69–71] to address QoS predictability, performance predictability remains an open industry problem particularly for modern large, complex and heterogeneous system deployments. Prior approaches generally assume that BET [33] holds, implying that the software systems as constructed and deployed are guaranteed to only produce ergodic measurement processes [268]. This presumption though has been not tested in the literature, outside of very early proof for only small scale systems in the earliest years of the IT industry [217]. To formally assess this presumption, we directly apply dynamical systems theory (DST) to test when and why BET may be violated in software systems and how this then leads to the generation of unpredictable performance behaviors within LDSSs, such as those observed in [4; 56].

Via this analysis, a clear and simple set of four software engineering design rules are produced. Following these four rules are shown to denote the necessary and sufficient conditions for BET to hold. Hence, these derived rules denote the "line in the sand" that differentiates run-time systems for which performance predictability exists, for example via traditional Markovian methods, from those systems where it doesn't, i.e. systems that exhibit non-stationary and non-ergodic statistical behaviours. The critical insight is also developed that software systems, particularly at large scale and complexity, are quite unique in that their run-time behaviours can take them back and forth across this predictability line. This denotes that software systems fall into a far more complex class of systems than are generally addressed in other engineering domains. This analysis therefore provides a direct formal explanation as to why real-world engineering of software systems has proven to be so challenging.

In summary, the core questions assessed in this Chapter are:

i) Are all software systems especially modern elastic cloud-deployed LDSSs assured to be BET-compliant?

ii) If not, then what conditions and scenarios cause BET violations?

iii) Subject to (ii), how must LDSSs then be designed such that BET-compliance and, therefore, performance predictability is guaranteed to hold?

Formal answers to (i)-(iii) will be developed via applying the EMM and DST approach of Chapter 4 to analyze the QN model of LDSSs developed in Chapter 3.

Chapter 5 begins by formally detailing how stochastic performance measures can be modelled as dynamical systems within the context of a software system's EMM and DST models. Formal definitions of statistical stationatrity and ergodic theory inclusive of BET and the equivalent definitions of ergodicity provided by Peter Walters in [254] are then provided and discussed. The QNM is then analyzed via the EMM-enabled DST approach to directly and formally assess (i) and (ii), where it is shown that: (a) queue drops and reliable protocols lead to the production of wandering sets of non-zero measures and that (b) fair OS process scheduling and high resource utilization levels lead to direct violations of measure invariance. Due to the differing DST impacts, (a) and (b) are addressed in details separately. The Chapter concludes by identifying the sources of performance predictability losses in software systems and applying the theory insights of (a) and (b) to develop a set of software engineering design rules required for software systems to maintain performance predictability,

as per Section 4.1 definition. Chapter 6 will then apply simulation to validate the Chapter 5 theory results.

5.1 Performance Measures as Dynamical Systems

As discussed in Chapter 4, the run-time performance measures of interest are typically macro-level observables obtained by calculating an aggregate function on a selected observable run-time performance feature over a specific time window. For example, average or maximum response times over 10 seconds or 1 minute periods are typical. Outside of trivial cases, these measurements are non-deterministic and must remodelled as a stochastic process $x(t)$ as shown in the top of the Figure. 5.1.

Run-time performance measures $x(t)$ are assumed to be produced by an executing software system deployed within a production execution environment. As a result, we have two interrelated dynamical system models. The first model encompasses the entire information space modelled by EMM, which represents the complete system along with its execution environment, be it a cloud regime, an embedded system, etc. The second model pertains to the subset of the information space of the first dynamical system which is responsible for mapping to the performance measure of interest through the application of $G_k[.]$.

In Chapter 4, we provided an in-depth description of the DST modeling of queuing network applying EMM as shown in the bottom of the Figure 5.1. It's important to note that while EMM deals with machine-level instructions, operating at a microscopic level of events, the performance measure $x(t)$'s dynamical system deals with macroscopic event measurements. These measurements are acquired by executing an aggregate function $F[.]$ on performance metrics over a specific time window.

As explained by Gray [33], any stochastic process, $x(t)$, has a dynamical system model defined by $(\Omega, \mathcal{B}, T, \mu)$ where $T : \Omega \rightarrow \Omega$ is a time-shift transform, μ is a measure such that $\mu : \Omega \rightarrow \Re^{+}$. If we normalize $\mu(.)$ such that it meets the axioms of probability, then $\mu(.)$ denotes a probability function. Within σ-finite measure spaces such a normalization is always possible, whereas it is not possible for infinite measure spaces.

Definition 5.1.1 (x(t)'s Dynamical System). A dynamical system on a σ-finite measure space is defined by quartet (Ω,$\mathcal{B}$,μ T) where,

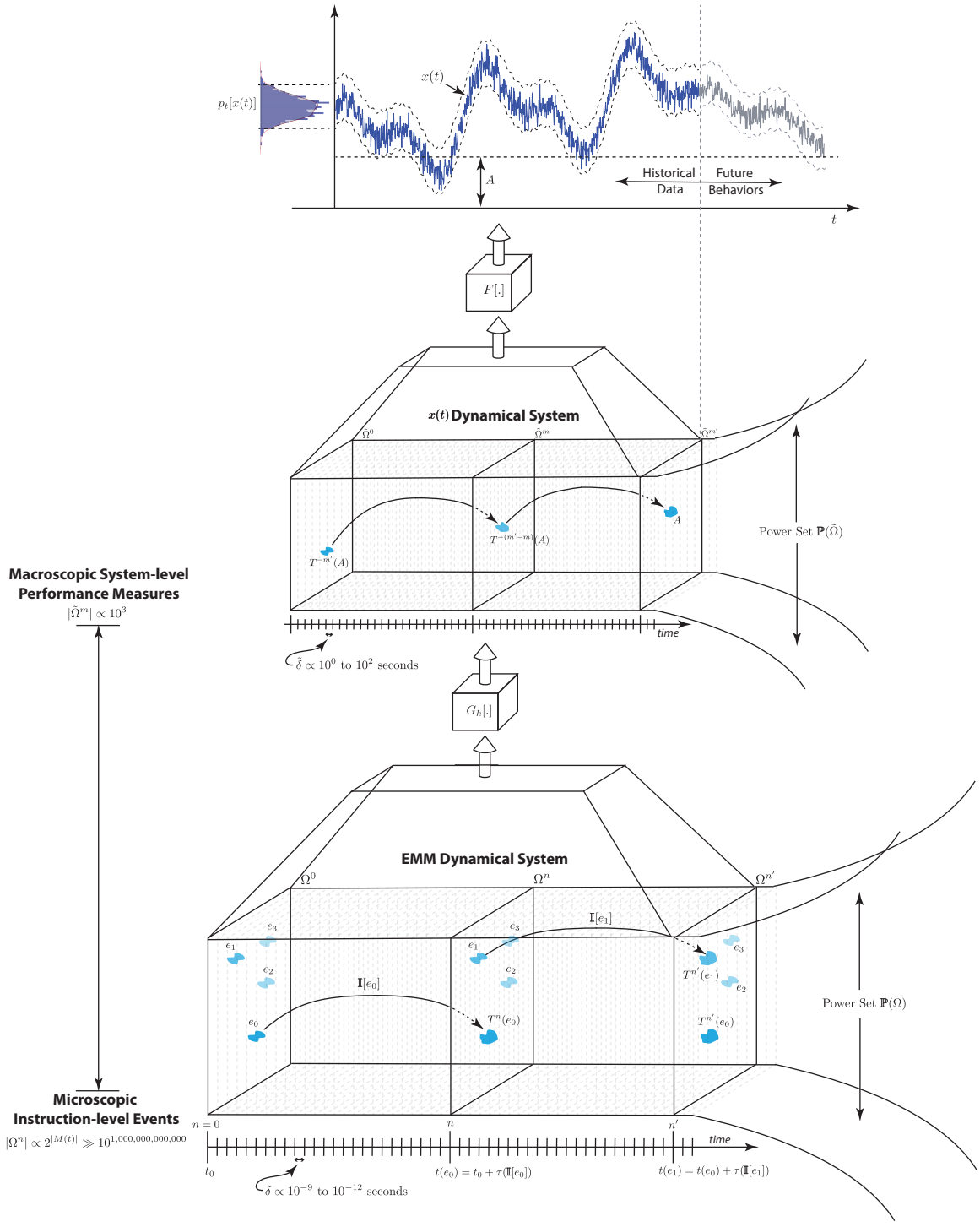


Figure 5.1: The relation between the performance measure $x(t)$'s dynamical system representation and the core underlying EMM-based dynamical system

- Ω is the universe of all possible events A_k such that for all k, $A_k \in \Omega$ and $|\Omega| < \infty$.
- $\mathcal{B}$ is the Borel Space over which Ω and $\mu(.)$ are defined.
- $\mu(.) : \Omega \to \Re^+$ is the mapping of the events onto the positive reals. If $\mu(.)$ can be normalized to meet the axioms of probability then $\mu(.) = p(.)$ and $\mu(.)$ can be considered to equivalently describes a pdf such that $\mu(\Omega) = p(\Omega) = 1$
- $T[.]$: $\Omega \to \Omega$ defines a mapping between event spaces, whether over time as per stationarity, or over an ensemble, as per ergodicity. For notational simplicity, Ω generally is assumed to denote a fully generic event space such that $T(\Omega) = \Omega$ is not assumed, i.e. $T(\Omega) \subset \Omega$ or $\Omega \subset T(\Omega)$.

Three very useful properties of a dynamical system are how $\mu(A)$ changes with the applications of $T[.]$, how multiple sequential application of $T[.]$ can be defined, and how A's pre-image $T^{-1}[.]$ is defined.

Definition 5.1.2 (**Measure Invariance**)**.** A measure $\mu(.)$ is invariant to $T[.]$ if and only if:

$$\mu[T^{-1}(A)] = \mu(A) \quad \text{for all } A \in \Omega \tag{5.1}$$

Where $T^{-1}(A)$ denotes A's pre-image. Equivalently, the transform $T[.]$ can be stated to be measure preserving.

Definition 5.1.3 (**Sequential Application of** $T[.]$)**.** Denote by $T^n[.]$ is sequential application of $T[.]$ such that:

$$T^n[A] = T[T[T[...T[A]]]] \tag{5.2}$$

If $T[.]$ is the time-shift transform then $T^n[.]$ denotes the forward progression of time, such that:

$$\forall j \quad t_j = T[t_{j-1}] \tag{5.3}$$

Definition 5.1.4 (**Pre-image** $T^{-1}[.]$ **of** $T[.]$)**.** Denote the pre-image of $T[.]$ by $T^{-1}[.]$ such that:

$$\forall A \in \Omega \quad A = T[T^{-1}[A]] \tag{5.4}$$

It should be noted that $T^{-1}[.]$ only denotes an inverse of $T[.]$ where $T[.]$ is a one-to-one and onto transform (or a bijective transform). If $T[.]$ is the time-shift transform then $T^{-n}[A]$ denotes the time reversal of A's mappings back into A's history.

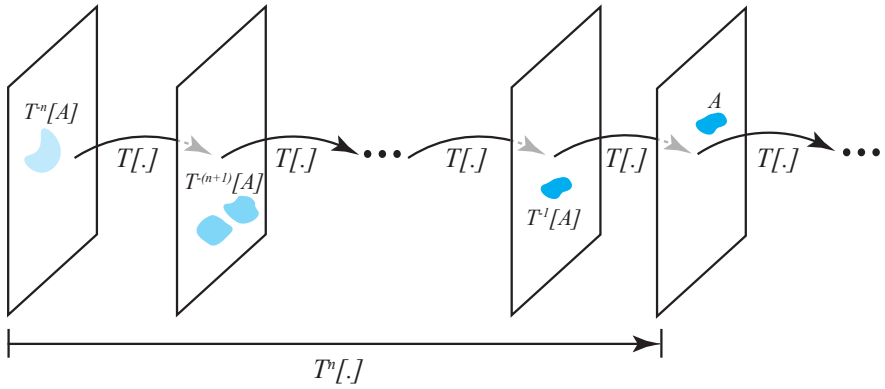


Figure 5.2: The relation between set A and its pre-image $T^{-1}(A)$

As will be discussed, most prior software predictability works and results rely on the presumption that BET holds. For example, Markovian solution approaches generally rely on recurrence theorems to hold, such as Poincarè recurrence [269], which generally rely on stronger forms of recurrence than the weaker recurrence of Theorem 3 Statement (3).

5.2 Statistical Stationarity and Ergodicity

Two critical concerns for modern software systems are: (i) how the system will behave over time, and (ii) whether multiple instances of the same (or a sufficiently similar) system, will perform similarly, i.e. do deployment regime sensitivities exist? Formally, within DST, issue (i) can be defined and assessed via statistical stationarity whereas issue (ii) denotes statistical ergodicity. As discussed in Chapter 4, stationarity and ergodicity can be defined through Birkhoff's Ergodic Theorem (BET) which states that:

Theorem 1 (Birkhoff Ergodic Theorem (BET))**.** *[258]*
If $\mu(.)$ is a probability measure on Ω and $T : \Omega \to \Omega$ is a measure preserving transformation, then $T[.]$ is ergodic if and only if:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} 1_A(T^n[x]) = \mu(A) \tag{5.5}$$

For each measurable set $A \subseteq \Omega$ and for almost every event $x \in \Omega$ of non-zero measure, where $T^n[.]$ denotes applying the transform $T[.]$, n times in succession and $1_A(x)$

denotes an index function where:

$$1_A(x) = \begin{cases} 1 \ \textit{if } x \in A \\ 0 \ \textit{otherwise} \end{cases} \tag{5.6}$$

The Left-Hand-Side (LHS) of BET assesses the average number of times x falls within A in the limit as the dynamical system's transform $T(.)$ is applied $N \to \infty$ times. If $T(.)$ is the time shift transform then this infinite sum only converges if statistical stationarity holds, i.e. if the dynamical system is stationary, the LHS of BET will converge to a scalar value.

BET provides the basis of Monte Carlo based simulation analyses as in general it is far easier and cost and time efficient to average across K experimental runs that it is to simulate very long runs. BET ensures that the averaged statistics computed across the former will provide identical information as if they were computed over the latter. Importantly, though applying BET requires proving Eq.5.5 holds, whereas most existing approaches presume BET holds without proof.

5.2.1 Formally Defining Statistical Stationarity

Assume $T : \Omega \to \Omega$ and that $T[.]$ is the time-shift transformation. Then if $T[.]$ is measure preserving then $\forall A \in \Omega$, $\mu(A) = \mu[T^{-1}(A)]$. If these A's define the cdf such that $P(x < A)$ then it is clear that the cdf cannot change over time, i.e. past histories of $x(t)$'s behaviour can be applied to accurately asses $x(t)$'s likely future behaviours. Formally, stationarity is defined as:

Theorem 2 (Statistical Stationarity)**.** *[33]*
The dynamical system $T : \Omega \to \Omega$ describing the stochastic process $x(t)$ is defined to be stationary if and only if $T[.]$ is:

- *One-to-one and onto, i.e., a bijective transform,*
- *The time-shift transform such that $x(t+1)) = T[x(t)]$ $\forall t$,*
- *$\mu(T^{-1}[A]) = \mu(A)$ almost surely $\forall A \in \Omega$.*

Where almost surely denotes that the only A for which $\mu(T^{-n}[A]) \neq \mu(A)$ must therefore be of zero measure in the limit as $n \to \infty$ such that $\mu(T^{-n}[A]) = 0$ $\forall A \in \Omega$ such that $\mu(T^{-n}[A]) \neq \mu(A)$ where $T^{-1}[A]$ denotes the pre-image of A.

It is important to note that the formal definition of stationarity permits the existence of a finite number of transient statistical behaviors within $x(t)$, provided that all of these eventually settle within a finite time frame. Hence, for a stationary stochastic process $x(t)$ described by its cdf $P_t[x(t)]$ then there must exist some finite time T_j such that:

$$\forall t > T_j \quad P_t[x(t)] = P[x(t)]. \tag{5.7}$$

However, stationarity requires that $x(t)$ always settles down to its steady-state behaviour in finite time post any disturbance to the system, with the formal requirement being "almost always" and, hence, only excluding events (or distributions) which themselves are of zero measure. Importantly, stationarity requires that $T_j < \infty$ exist but it does not seek to determine T_j's value for every (or any) disturbance, as these settling times are themselves system and disturbance dependent and, hence, cannot be generally derived or assessed.

Additionally, it should be noted that as DST addresses Ω in its entirety and against all possible disturbances, it cannot be verified experimentally, outside of extremely trivial systems. Theoretically derived insights, as developed via Chapter 5, though must be validated via experimental work, as provided in Chapter 6, to ensure the developed theory provides a useful model for the systems being analyzed, i.e. to avoid the problem of inaccurate theory models providing poor and inapplicable insights.

5.2.2 Formally Defining Statistical Ergodicity

Clearly, software system executables can be re-instantiated and/or multiple identical copies of those executables could be instantiated across multiple cloud computing facilities, i.e. to facilitate load balancing, etc. Hence, a reasonable question to ask is whether observations obtained from any given run-time instance can be used to predict the performance behaviours of any of the other run-time instances of the same (or sufficiently) similar system, assuming every instance exposed to the sufficiently similar incoming statistical workloads (as described in Chapter 3). Assume that $k = 1, \ldots, K$ run-time instances of a software system exist all deploying identical executable code bases. Then for any chosen stochastic performance measure $x(t)$, the above question can be formally posed as asking if:

$$P_k[x(t) < A] = P[x(t) < A] \quad \forall A \in \Omega. \tag{5.8}$$

Theorem 3 (Ergodic Equivalency). *[254]*
If $T : \Omega \to \Omega$ is a measure-preserving transformation of the probability space $(\Omega, \mathcal{B}, \mu)$ defined over an ensemble $k = 1, \ldots, K$ of experiments runs then in the limit as $K \to \infty$ the following statements are equivalent:

1. *$T[.]$ is ergodic.*
2. *The only members $A \in \mathcal{B}$ with $\mu(T^{-1}[A]\Delta A) = 0$ are those with $\mu(A) = 0$ or $\mu(A) = 1$ (where Δ is symmetric set difference: $A\Delta B = A/B \bigcup B/A$).*
3. *For every $A \in \mathcal{B}$ with $\mu(A) > 0$ it is the case that $\mu(\bigcup_{n=1}^{\infty} T^{-n}A) = 1$*
4. *For every $A, B \in \mathcal{B}$ with $\mu(A) > 0$ and $\mu(B) > 0$ there exists $n > 0$ with $\mu(T^{-n}A \cap B) > 0$*

This theorem defines three equivalent tests that can be used to prove whether or not ergodicity hold, where (2) denotes the formal definition of ergodicity, (3) denotes the existence of a recurrence relationship, and (4) denotes that only wandering sets of zero measure may exist. It should also be noted that for ergodicity, $T[.]$ is defined as a general transform whereas stationarity defines $T[.]$ specifically as the time-shift transform as presented in Figure 5.3.

Most prior works have pursued software system performance predictability either via assuming ergodicity hold (Theorem 3 Statement (1)) or via assuming as per Markovian approaches, that recurrence holds (Theorem 3 Statement (3)). Although applicable to smaller-scale it remains an open question with the current literature as to how to formally prove Statements (1) and (3) hold for model large-scale complex software systems and deployment regimes.

A core insight within this research and dissertation is that Theorem 3 Statements (2) and (4) can instead be applied to enable formal analyses of modern large-scale complex software systems. As will be discussed, these mathematically equivalent statements do not suffer the scale-induced challenges of Statement (3). Instead, when combined with the Chapter 4 developed QN-based DST-based EMM approach they provide a scale-agnostic approach for formally deriving the necessary and sufficient conditions for ergodicity to hold in software systems irrespective of the complexity or their run-time deployment regimes.

Proving recurrence for large-scale software systems has proven to be extremely challenging and difficult in practice, given the inherent complexity, scope, intricate

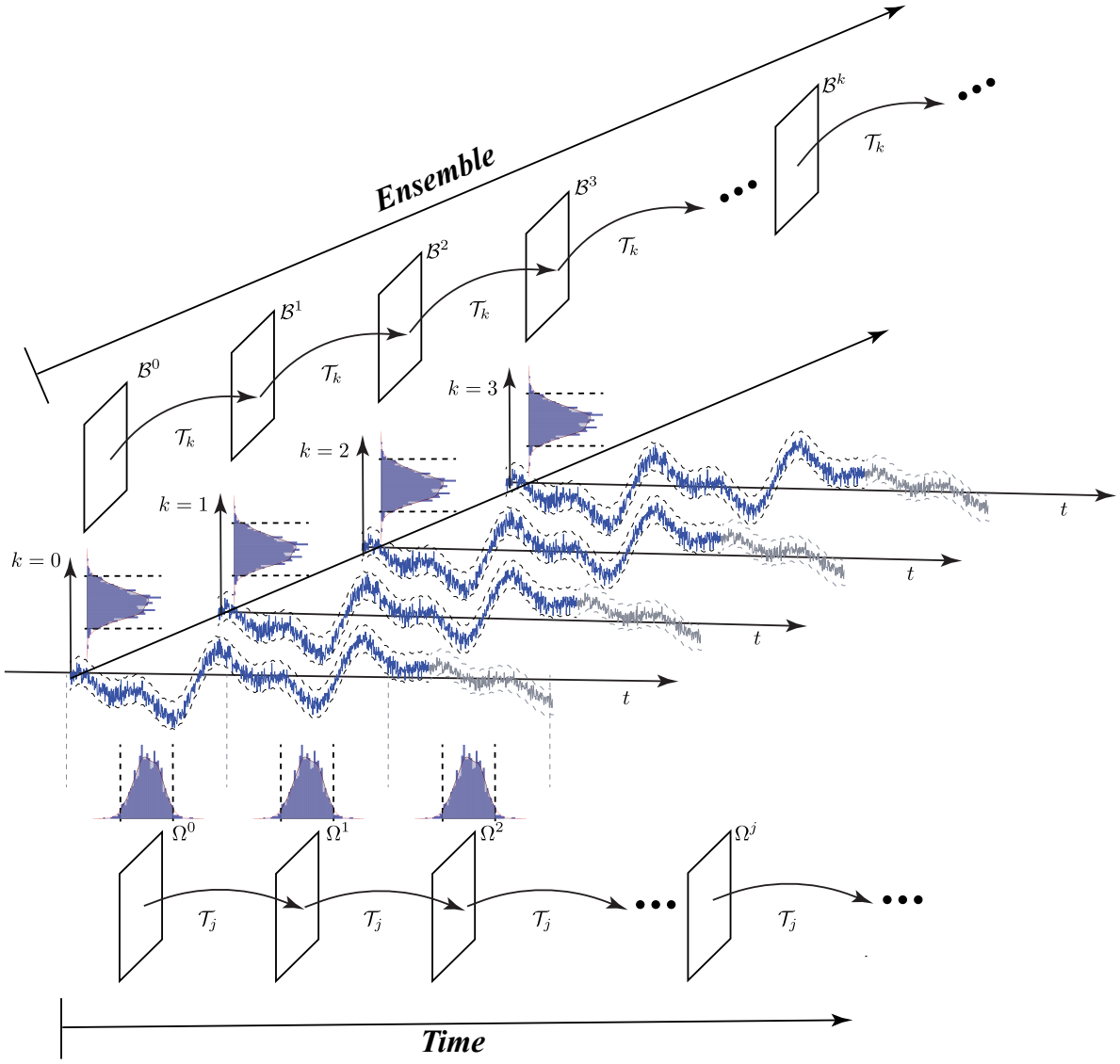


Figure 5.3: Stationarity and ergodicity relation

interactions, and time varying modern systems. Hence, current academic literature primarily models systems that are typically of orders of magnitude lower scales and complexities than those that are now commonplace within industry. Academic simulation and experimentation studies, while valuable, innately cannot assess the scales and complexities of modern global-scale cloud-deployed software systems. Formally assessing recurrence holds (Theorem 3 Statement 3) is hampered by modern real-world systems possessing many thousands of queues spanning the hardware and network layers, though the OS layers, and within each involved server's VM, container,

and application layer(s). Such realities have lead to existing Markovian-based literature generally focusing on simulating relatively small (to very small) scale QN-based systems when compared to the scale and complexities of, for example, the modern global software industry's commonplace SaaS systems and solutions..

Statement (2) defines ergodicity, by stating the only places where the symmetric set difference between any set $A \in \mathcal{B}$ and its pre-image $T^{-1}[A]$ can be of zero measure are when:

i) A itself has zero measure, which then requires $\mu(T^{-1}[A]) = 0$ if $\mu(T^{-1}[A]\Delta A) = 0$ is to hold, given that $\mu(.)$ must be consistently applied and A and $T^{-1}[A]$ must be identical almost surely everywhere.

ii) $\mu(A) = 1$ which requires that $A = \Omega$, i.e., it is required that $T[\Omega] = \Omega$, given the Theorem 3 is for σ-finite measure spaces.

It should be noted that Theorem 3 Statement(3) defines weak recurrence, whereas standard Markovian analysis approaches typically rely on strong recurrences as provided via Poincaré recurrence theorem [270; 271]. By comparison, Theorem 3 Statement (4) states that wandering sets of non-zero measure cannot exist in ergodic systems defined over σ-finite measure spaces. By contrast, for infinite measure spaces, wandering sets of non-zero measure leading to more complex definition of and consideration regarding ergodicity concept within infinite measure spaces than are covered by Theorem 3, which applied only to σ-finite measure spaces and not infinite spaces.

5.2.3 A Practical Illustrative Example

The Theorem 3, provides the conditions required for the RHS of BET to be a scalar and for the LHS BET to be a convergent scalar. To more clearly illustrate the BET concepts and the fundamental roles they play in the analysis and predictability of systems a simple coin-flip based system is considered. Assume $\Omega = \{\mathrm{H}, \mathrm{T}\}$ denotes the only allowable event outcomes of the coin flip. Denote by Ω^J the space of possible "Heads" (H) and "Tails"(T) observed, produced by J successive iterative coin flips, i.e. the time sequence corresponds to coin flip outcomes (e.g. for $J = 6$ one possible event is $e_{j=6} = \{\mathrm{H,H,T,T,T,H}\}$).

Now, extend the experiment by defining three separate coins. Coin 1 is a fair coin that is always flipped first and used to select whether Coin 2 or Coin 3 is then used for all subsequent flips within experiment run. All coins appear identical to the

observer and, hence, the observer can only "observe" (or measure) iterative sets of "Heads" or "Tails" outcomes of each experiment run. Consider the Scenario A and Scenario B experiments described below.

5.2.3.1 Scenario A:

In Scenario A, Coin 1, Coin 2, and Coin 3 are all fair coins as shown in Figure 5.4 such that:

$$\Omega_{\text{Coin}_1} = \Omega_{\text{Coin}_2} = \Omega_{\text{Coin}_3} = \Omega \quad \text{and} \quad \mu\{\text{H}\} = \mu\{\text{T}\} = 0.5. \tag{5.9}$$

In each run of experiment, we first flip Coin 1. If the outcome of Coin 1 is "Heads", we will use Coin 2 for J successive flips. Conversely, if the outcome of Coin 1 is "Tails", we will use Coin 3 for J successive flips. This setup ensures that the choice of the coin for the successive flips depends entirely on the result of the initial flip of Coin 1.

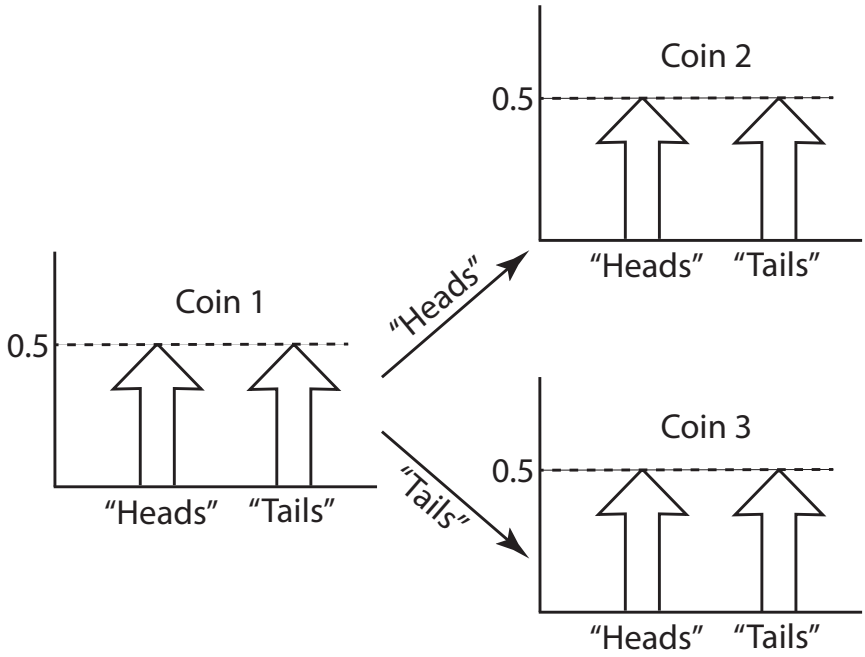


Figure 5.4: Coin Flip Scenario A in which Coin 2 and Coin 3 have the same probability $\{\text{H}, \text{T}\}$ distributions and all observed outcomes are independent and identically distributed (iid).

In the limit as $J \to \infty$, the outcome of the first coin flip (or the system's initial condition) becomes a zero-measure event. The only element $A \in \mathcal{B}$ is $A = \{$ H, T $\}$ and it is clear that $\mu(T_k^{-1}[A]\Delta A) = 0$ with $\mu(A) = 1$. Statement (2) clearly holds, and the system as described in Scenario A is ergodic.

5.2.3.2 Scenario B:

As shown in Figure 5.5, Coin 1 and Coin 2 are again fair coins, but Coin 3 is now an unfair coin such that:

$$\mu(\{\mathrm{H}\}_{\mathrm{Coin_3}}) = 0.7 \quad \text{and} \quad \mu(\{\mathrm{T}\}_{\mathrm{Coin_3}}) = 0.3 \tag{5.10}$$

This means that Coin 1 and Coin 2 are fair coins with equal probabilities of landing on "Heads" or "Tails" (0.5 each). Coin 3, however, is biased with a 0.7 probability of landing on "Heads" and a 0.3 probability of landing on "Tails". The rest of the experiment is similar to Scenario A.

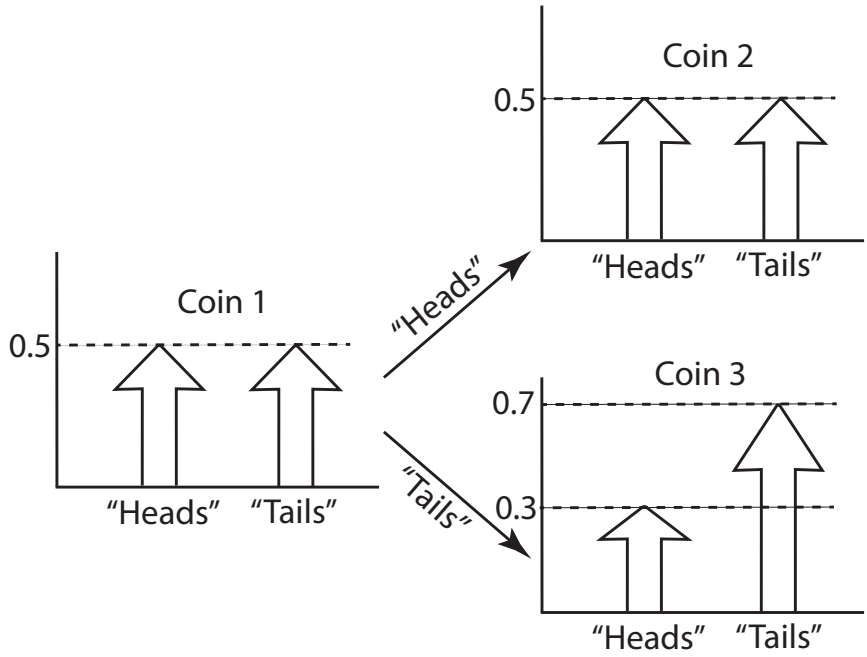


Figure 5.5: Coin Flip Scenario B in which Coin 2 and Coin 3 have unequal probability distributions.

The initial condition provided by Coin 1 remains a measure zero event in the limit as $J \to \infty$. Importantly, an observer can only see the "Heads" and "Tails"

outcomes which are indistinguishable between Coin 2 and Coin 3. Therefore, from the observer's perspective, $\Omega_{\text{observ.}} = \{H, T\}$. The actual (or true) experiment event space over the ensemble is composed of the composite of the fair and unfair coin event spaces, given by:

$$\Omega_{fair} = \{H_{fair}, T_{fair}\} \tag{5.11}$$

or

$$\Omega_{unfair}\{H_{unfair}, T_{unfair}\}. \tag{5.12}$$

Hence, the true event space of the experiment is $\Omega = \Omega_{fair} \times \Omega_{unfair}$ across the K runs of the experiment, wherein only the events H_{fair}, T_{fair}, H_{unfair}, and T_{unfair} can have non-zero measures. The action of the first coin flip, in any given experiment run, is to then set the measure $\mu(.)$ such that either $\mu(H_{fair}) = \mu(T_{fair}) = 0$ or $\mu(H_{unfair}) = \mu(T_{unfair}) = 0$ for all subsequent coin flips seen by the observer during that experiment run. As such, the measure $\mu : \Omega \rightarrow \Re^{+}$ depends on the per-experiment initial condition, as produced by the outcome of the first coin flip event. From the observer's perspective and relative to the observable event space Ω, the measure $\mu : \Omega \rightarrow \Re^{+}$ is given by,

$$\mu(X) = \begin{cases} 0.5 & \text{if } X_1 = H \\ \mu(H_{\text{Coin}_3}) & \text{if } X_1 = T \text{ and } X_j = H \\ 1 - \mu(H_{\text{Coin}_3}) & \text{if } X_1 = T \text{ and } X_j = T \end{cases} \tag{5.13}$$

Where X_1 denotes the outcome of the 1^{st} coin flip and X_j denotes the subsequent j^{th} coin flip within the same experiment run and $\mu(\Omega) = 1$ in all cases. As such, the measure $\mu(X)$ no longer denotes a scalar as required by BET's RHS as it instead describes a distribution. Averaging the results across any K experiment runs of each of J_k successive coin flips will produce a $\bar{\mu}(X)$ scalar but one, by definition, that exists somewhere in the continuum between the above "fair"/"unfair" coin boundary cases. Importantly, the boundary cases though exist as the only true possible models of the experiment. Hence, although averaging produces a scalar, it also will almost assuredly only produce a clearly wrong statistical model of the given experiment. More particularly, this averaged model is one that expressly can never exist within the real coin flip system. This highlights the necessity of ensuring BET applies (or provably holds) before such averaging is applied if meaningful and informative results are to be produced.

This arises as Monte Carlo simulation and ensemble average-based analyses are based on the underlying presumption that BET holds and, hence, that the ensemble averaged statistics equal time averaged statistics. As discussed above, under Scenario B, the right-hand side of BET becomes a distribution and therefore, it can not equal LHS scalar. Hence, the system of Scenario B is asymptotically stationary but not ergodic.

It should also be noted that the ergodicity of the above coin flip experiment relies only on Coin 2 and Coin 3, from the observer's perspective, having identical events with identical measures and not on whether both Coins are fair or not. Two equally unfair coins would also produce an ergodic system. Moreover, as Coin 1 only provides a measure zero initial condition to the system, in the limit as $J \to \infty$ the asymptotic ergodicity of the system arises independently of whether Coin 1 is fair or not or whether it is similar to Coin 2 or Coin 3. In all cases, the system remains asymptotically stationary as each Coin's respective possible event outcomes and their respective measures remain constant i.e. with iterations of J successive flips of the same coin are made.

5.3 Assessing BET for Software Systems

This Section combines the QN model of Chapter 4 with the EMM DST model to directly analyze when BET does and does not hold for run-time LDSSes and their performance measures. More particularly, traditional Markovian analysis approaches require that recurrence holds, typically for example via an application of Poincaré's Recurrence theorem (strong form) [269]. It is this BET-compliance assumption that then enables both Monte Carlo simulation and matrix-based Markovian analytical methods. From the perspective of Theorem 3, it is Statement 3 that formally describes recurrence (weak form) but it remains an on-going open research problem as to how to prove and ensure recurrence holds within modern large- to very-large scale software systems under modern deployment and management regimes, i.e., modern large-scale cloud-deployed software systems.

Theorem 3 though also denotes the equivalencies of Statements 1 through 4. More particularly, this Section shows that Statements 2 and 4 can be applied to provide a scale-agnostic approach to assess BET-compliance for run-time LDSS systems and, hence, to assess the predictability of the operational performance measures they produce. Scale-agnostic is an important property as any analysis methodol-

ogy constructed for LDSSes should continue to work and provide useful insights for smaller-scale systems, down to and including stand-alone embedded systems. All results of this Section apply only to systems whose behaviors can be described by dynamical systems defined over σ-finite measure spaces, i.e., the spaces over which probabilities and probability measures can be defined. The research does not address or seek to address infinite-measure spaces. As such, the provided analyses apply to all classical computing and may also apply to quantum computing, given the latter is typically formally expressed within probabilistic frameworks. Additionally, we show that Statements 2 and 4 of Theorem 3 relate to real-world easily measurable software engineering properties and, hence, produce a set of four easily followable software engineering design rules which are shown to be necessary and sufficient if a deployed LDSS is to produce predictable performance measures.

More particularly for simplicity but without a loss a generality, assume a specific time-domain run-time performance measure of interest, given by $x_k(t)$, is produced by the k^{th} deployment instance of a given LDSS. As discussed previously in Chapter 4, the focus is on assessing when it can be formally shown that:

$$P[x_k(t) \in [B_{lower}(t), B^{upper}(t)]] \geq 1 - \epsilon \; \forall t \in [0, T] \tag{5.14}$$

for some appropriately small $\epsilon \in [0, 1]$ over some defined time period $[0, T]$ where for convenience $t = 0 = t_0$ is the assumed start time and $0 < T < \infty$, i.e., that the performance of $x_k(t)$ falls within some prescribed upper and lower bounds. Clearly, assessing Eq. (5.14) requires knowledge of $x_k(t)$'s underlying cdf, defined by $P[x(t)]$. Most generally, this cdf will be both time and ensemble dependent. Hence, it is more accurately denoted as $P_{k,t}[x_k(t)]$, where t is the time index and k is the ensemble index. From the BET perspective, the LHS can be seen as asking if and when there exists a $t' \in [T_{\text{stat.}}, \infty)$ such that $\forall t' > T_{\text{stat.}}$,

$$P_{k,t}[x_k(t')] = P_k[x_k(t)] \tag{5.15}$$

Hence, ostensibly, the BET's LHS asks whether the given run-time instance of the LDSS assuredly settles down over time into a statistical steady-state behavioral model, as described by $P_k[x(t)]$ and ideally, it would be the case that $T_{\text{stat.}} < T$. More formally, the notion is that this convergence over time into a steady-state behavioral model must occur within a finite time post any disturbance to the system or any finite set of such disturbances, where all disturbances themselves can only span finite time.

Hence, BET does not in any way seek to address what may happen to a system while it is undergoing a disturbance, only that it must settle down to a known statistical behavioral model (or state) in finite time post any disturbance.

BET's RHS then asks whether this state is unique, i.e., do all possible instances of the LDSS under all possible disturbances always settle down to the same steady-state statistical behavioral model. For the purposes of this research, we are interested in the asymptotic behaviors of the LDSS in the limits as t and k go to infinity, given we are interested in how all possible instances of a deployed LDSS may behave. As such, BET's RHS can be seen as assessing in the limit as $K \to \infty$ whether,

$$P_k[x_k(t)] = P[x_k(t)] \; \forall \; k = [1, 2, \ldots, K] \tag{5.16}$$

Mathematically, BET requires that its LHS and RHS converge to the same scalar value, i.e., the LHS must be a convergent sum and the RHS cannot be a distribution. In both cases, for the conducted analyses it is assumed that the LDSS of interest is only processing BET-compliant in-coming statistical workloads. From the $x_k(t)$ performance measure perspective, the events $A \in \Omega$ of particular interest are those that inscribe the $P_{k,t}[x_k(t)]$ cdfs, i.e., the sets A for which $P_{k,t}[x_k(t) \leq A]$ for all $A \in [min(x_k(t)), max(x_k(t))]$ and for all t. It should be noted that $P_{k,t}[x_k(t)]$ can be both time and ensemble dependent. Hence, why $x_k(t)$ retains its k subscript, as $x_k(t) \neq x_{k'}(t)$ for all $k \neq k'$ and $t \in [0, T]$ almost always. The formal BET analysis will focus on applying Theorem 3 to assess the above issues in the context of:

(i) How wandering sets of non-zero measure can be and are produced within the QN model described EMM DST model (Theorem 3 Statement 4),

(ii) When event measures $\mu(A)$ can be shown to arise in BET consistent (or non-consistent) manners within the QN model described EMM DST model (Theorem3 Statement 2), and

(iii) Consequently, when BET's RHS and LHS can be shown to hold for the DST model of the chosen $x_k(t)$ run-time performance measure, i.e., when predictable run-time performance measures arise, as defined by Eq. (5.14).

Importantly, this distinction between the EMM-based DST model and the $x_k(t)$ DST model must be clearly understood. The EMM provides the lowest level description of the operation of the deployed LDSS. It specifically focuses on how each and

every memory change in the system occurs as the LDSS and its deployment environment run. As such, the QN model of the LDSS's operation is with respect to the EMM DST model. Performance measures, by definition, only focus on selected aspects of a LDSS's run-time operation, e.g., average response time, maximum latency, median jitter, etc. Hence, each given performance measure $x_k(t)$ exists within the context of its own DST model, formed as an information lossy mapping of the EMM's full DST model.

What is and is not mapped depends on the nature and characteristics of the specific $x_k(t)$ performance measure. This is captured as the $G_k[.]$ mapping denoted previously in Figure 5.1. For simplicity, Figure 5.1 also includes a mapping $F[.]$, which is intended to denote the specific statistical moments and characteristics reported for the performance measure $x_k(t)$, such as moving averages, moving medians, and moving quantiles, etc. Generally, in modern cloud-deployed LDSSes and associated cloud management systems, the instrumentation of these performance measures is separated out from the reporting shown on the user interface dashboard. This separation of the $G_k[.]$ and $F[.]$ mappings captures this reality. Mathematically, the operations in $F[.]$ can be integrated into $G_k[.]$ such that $F[.] = 1$ without a loss of generality.

With respect to the $x_k(t)$ performance measures, the conducted analyses will explore when:

1. *Ergodicity* does (or does not) hold with respect to the dynamical system defined over the ensemble of $x_k(t)$ observables (the RHS of BET), and

2. *Stationarity* does (or does not) hold with respect to each $x_k(t)$ time evolution (the LHS of BET).

It tends to be simpler to view the QN model in terms of its time domain operation. Hence, the issues of (2) will be explored first before discussing how these translate into the issues of (1). For notational simplicity, $T[.]$ and $\mu(.)$ will not be subscripted as it is clear by context whether (1) or (2) is being analyzed, i.e., whether $T[.]$ is defined over the ensemble, as per (1), or as the time-shift transform, as per (2). Clearly, to show BET-compliance does not hold it suffices to show that either the LHS or RHS do not produce scalar values. Within the conducted analyses though the LHS and RHS are both assessed. This more complete analysis is done as a number of emerging works are seeking to apply artificial intelligence (AI) or machine learning (ML) approaches to the problem of LDSS behavioral predictability (e.g.[69–71; 76; 12; 272; 72–75]). For such AI/ML techniques to work, the training data must contain sufficient Shannon

information to support accurate off-training data prediction. More directly, AI/ML results are based on knowledge they have been made aware of in training or via on-going training. Hence, from a Shannon information perspective, BET's LHS asks when and if past run-time behavioral histories can be used to provide sufficiently accurate predictions of a given LDSS deployment instance's likely future behaviors. BET's RHS, by comparison, asks when past knowledge gained from other run-time deployment instances of the same LDSS (or a sufficiently similar LDSS) can be used to accurately predict the given LDSS deployment instance's future behaviors. Through providing a complete analysis of BET's LHS and RHS, we also directly answer the questions as to when and why AI/ML techniques can or cannot be effectively used in LDSS performance prediction.

Importantly, the conducted analyses will also show that LDSSes do not exhibit static behaviors with respect to (i)-(iii) and (1)-(2) above. Instead, during run-time operations LDSSes can (and do) move back and forth over the BET denoted mathematical "lines in the sand" both within given LDSS instantiations and across sets of LDSS instantiations. Hence, LDSSes act in fundamentally different and distinctly more complex ways than do many classical DST analyzed systems, such as those arising within the more traditional domains of Physics, Mechanical Engineering, etc. This is further complicated by the operational reality that LDSSes evolve, are patched, are re-deployed, are enacted on by dynamic resource allocations and elastic services, etc. within deployment environments that themselves are patched, re-deployed, enacted on by management services, by cyber-security services, etc. Hence, from the QN and QN topology perspective, LDSSes inclusive of their deployment environments must also be considered as time-variant and, potentially, ensemble variant entities. As such, the conducted analysis must be with respect to this family of possible QNs and QN topologies that could arise from a given LDSS deployment instance or set of deployment instances, in terms of when and why the (i)-(iii) and (1) and(2) issues above arise.

As will be seen, the conducted analyses' focus on Statements 2 and 4 of Theorem 3 are inclusive of this family of potential QNs and QN topologies. Hence, the analysis produced insights and software engineering design rules innately are generalizable, as a consequence of generalizability of QN models and DST in both the EMM and performance measure contexts. Recognizing this generalizability across a LDSS's potential family of QN model and topologies though is important to the formal analysis of problem in the domains such as cloud resource allocation, elastic services, and the

applications of formal control theory based approaches[63–68]. The development and analyses of these more advanced areas are outside of the intended scope of this work.

Finally, as per Chapter 4, the applied QN model is idealized in that all inter-queue transitions are assumed to always occur successfully and in zero time. These idealizations are important as they allow the software engineering issues associated with any run-time performance measure to solely be restricted to those issues involve or arising within the modelled queues themselves, e.g., their behaviors, characteristics, topological interconnections, and execution regimes. This can be done without a loss of generality as transmission-oriented issues can simply be incorporated into per-queue characteristics, e.g., a wireless packet collision can be equivalently modelled as a queue drop event in the recipient queue. From this QN-focused software engineering perspective, we identified a complete set of four core issues that produce DST impacts relevant to BET analysis. These four core QN-related software engineering issues are:

1. Queue drop events and, particularly, on-going recurring queue drops within the modelled LDSS system and deployment.

2. Re-generation of queued events through the actions of reliable protocols, whether within the network, OS, or application layers.

3. Execution timing impacts produced by OS scheduling regimes, whether due to commonplace fair scheduling regimes or alternative regimes, such as Real-time Operating System (RTOS) scheduling, and

4. Execution timing impacts resulting from system resource utilization, loading, and contention issues.

Clearly, each of these issues is directly measurable in a software engineering sense within run-time LDSSes and their modern deployment regimes. Moreover, the above are a complete set of the issues that can occur relative to the QN model, under the interconnection idealization. A core insight of this research is how to map these software engineering relevant and easily run-time measurable issues to the foundation mathematics provided via DST that define when and why $x_k(t)$ performance measures are or are not predictable, as per Eq. (5.14). Again, it should be noted that the analysis follows the common traditional approach used within Physics, Mechanical Engineering, Chemistry, etc. of focusing the analysis on the macroscopic observables of a systems, in this case the $x_k(t)$ performance measures, wherein these arises as a

consequence of vast numbers of unobservable microscopic interactions and events, as defined via the EMM DST model. It is important in the analyses that follows that confusion does not arise between the macroscopic $x_k(t)$ DST model and microscopic EMM DST model. They are different and quite distinct from each other as highlighted previously in Figure 5.1.

For simplicity, Section 5.3.1 begins by exploring how wandering sets of non-zero measure (Theorem 3 Statement 4) are produced first across the time-series data produced by a given LDSS deployment instance and then across an ensemble of the LDSS's deployment instances. Section 5.3.2 then explores how variations in measure assignments can arise (Theorem 3 Statement 2) again first across the time-series data produced by a given LDSS deployment instance and then across an ensemble of the LDSS's deployment instances.

5.3.1 Production of Wandering Sets of Non-zero Measure (Theorem 3 Statement 4)

This analysis begins by looking at software engineering issues (1) and (2) above, namely QN queue drop events and the event regenerations produced via reliable protocol actions. Both are shown to produce wandering sets of non-zero measure with respect to the EMM DST model, although the production mechanisms are different and distinct across the two issues. Hence, for formal BET-compliance it is shown that neither (1) or (2) can be on-going issues within the LDSS's run-time operation. Pragmatically, even if BET-compliance does not formally hold, the levels at which (1) or (2) arise can be sufficiently low so as not to be observable under statistical goodness-of-fit tests.

In such cases, a given $x_k(t)$ performance measures can "appear" to be predictable via appropriately selected data analysis methods, while concurrently failing to formally BET's DST requirements. Such boundary cases arise due to the well-known issue of the innate uncertainties associated with data-driven statistics when compared to measure theory's more absolute analytical nature. Importantly, it is also shown that the $G_k[.]$ mapping existing between the EMM DST model and each $x_k(t)$ performance measure DST model entails that run-time LDSS's can concurrently support both predictable and non-predictable $x_k(t)$ performance measures, where this depends on whether or not the given performance measure is (or is not) impacted by the produced wandering sets.

More particularly, from the software engineering perspective and relative solely to issues (1) and (2) above, a given run-time $x_k(t)$ performance measure will be shown to be formally:

- *BET non-compliant* if the measure is impacted by queue drop events and/or reliable protocol event recovery actions, and
- *BET compliant* if the measure is independent of and unaffected by any and all queue drop events and/or reliable protocol event recovery actions.

Moreover, as queue drop and reliable protocol actions are themselves time dependent, it can also be the case that a given $x_k(t)$ performance measure to transition from being predictable to non-predictable (or vice versa) during the run-time operation of the LDSS, i.e., the LDSS can transition back and forth across the BET-compliance/non-compliance boundary as it runs. Traditionally, DST modeling with classical domains such as Physics, Chemistry, Mechanical Engineering, etc. do not consider such cases. Traditionally, BET-compliance arises largely as a consequence of Noether's Theorem [256]; Classical systems tend to be either BET-compliant or non-compliant, i.e., traditional DST systems are modelled in terms of how invariant measures lead to Hamiltonian or Laplacian system models. Modern LDSS therefore exhibit significantly more complicated DST behaviors than the classical DST systems found in more traditional domains.

It should be noted that data driven analyses cannot be used to formally assess BET compliance. This can only be done via DST-based analytical analyses. Again, this is due to the well-known uncertainty issues associated with statistical methods, as compared to analytical probabilistic or measure theory analyses. As will be seen in Chapter 6, the cases of interest in this work clearly fall well outside of the typically relatively small neighborhood of statistically non-observable BET non-compliant systems. Instead, as the Chapter 6 simulation results will show, the BET non-compliant LDSS scenarios produce fairly broad families of $x_k(t)$ cdfs even for the pragmatics but fairly limited cases of ensembles of 100 experiment runs.

As will be seen, the developed DST mathematical analyses directly guide software engineering measurable observables and insights into the degree to which a given LDSS run-time behaviors will (or will not) be statistically predictable (or well-behaved). Hence, the above formal distinction between when BET-compliance does or does not hold versus when it can be statistically measured is primarily of theoretical interest. The theoretical analysis does have important implications in emerging

areas such as the use of formal control theory approaches within cloud resource allocation and elastic services, as BET-compliance directly related to the formal issue of controllability. Such issues though are outside the intended scope of this work and left as future research areas. A core insight of the conducted research though is that real-world easily measurable software engineering metrics can be directly connected to the underlying BET mathematics governing performance measure predictability, which highlight the value of taking a DST-centred analysis approach.

Within all the analyses, the LDSS is assumed to be processing typical workloads such that the LDSS is neither a highly overloaded state or a starvation state. Clearly, run-time performance prediction is trivial in these boundary cases. Hence, the interesting cases are those in this middle ground. As per BET theory, it may be the case that the LDSS enters starvation or overload for limited time periods, such as may occur with bursty workloads. As per the formal DST stationarity and ergodicity definitions, such situations do not impact the resulting analyses as the only excluded cases are permanent and complete system starvation or overload. Moreover, the workloads arriving at the LDSS to be serviced are presumed to themselves be BET-compliant. This is necessary as it makes no sense to seek to assess LDSS performance predictability for workloads that themselves are not predictable.

Finally, of interest in analyses is assessing the limits of an LDSS's observable performance predictability across the full extent of time and over the full set of potentially instantiable systems. Hence, the conducted analyses do not seek to explore (or assess) the time period that may be required for a given LDSS to settle down post-disturbance into a well-behaved (or BET-compliant) statistical state. Again, such issues are important in terms of the design and analysis of dynamic resource allocation and elastic service approaches, but they are also innately LDSS and disturbance dependent. Hence, these issues have been left as areas of future research.

More formally, the conducted assessments focus on asymptotic stationarity and asymptotic ergodicity in the limit, respectively, as time goes to infinity and the ensemble cardinality goes to infinity, i.e., the system's behaviors in $\lim_{j\to\infty} T^j[.]$ and $\lim_{k\to\infty} T^k[.]$. In this sense, the analyses ask the question as to "Can the LDSS produce predictable performance measures?", as opposed to whether a given LDSS deployment instance does or does not behave predictably. It will be shown that answering the former question leads to software engineering insights and easily run-time measurable quantities that enable the latter to also be answered. In general, particularly for cloud-based LDSS deployments, there is limited to no control over

the detailed characteristics of any given deployment instances. Hence, "luck of the draw" performance predictability is of limited value given that data drawn at different times or from different execution instances would no longer exhibit the claimed run-time statistical behaviors, as per Eq. 5.14. As such, pragmatically it is asymptotic BET-compliance that is of more interest and the solely focus of the analyses.

5.3.1.1 Impacts of Queue Drop Events

By definition, when an event is dropped out of a queue within the QN model, that event is then "lost" from the QN model in that it assuredly can no longer trigger (or produce) any future events (or memory changes) within the QN. Within Markovian analyses queue drop rates are computable but, via commonly applied Markovian mathematics, any information contained within any and all dropped events is then also "dropped" out of the subsequent analyses. In general, this leads to the common practice of reporting Markovian analyses in terms of the average values of measure performance measures of interest, as opposed to the full characteristics of the performance measures' complete pdfs (or cdfs).

The commonplace Markovian analyses approach of reporting on only a relatively small set of low order statistical moments, e.g., average, median, etc., can also arise via the presumption that the Law of Large Numbers (LLN) applies and, hence, that these denote informative statistics. The LLN though also presumes BET-holds [33]. Hence, it is not appropriate to non-BET compliant regimes. Moreover, as per Eq. (5.14), the performance predictability of interest in this research innately requires knowledge of the tail behaviors of the cdfs. Such behaviors are typically not captured by low order statistical moments for anything but quite simplistic distributions or, more formally, cdfs completely describable via a small set of sufficient statistics, e.g., exponential family distributions.

As per Figure 5.6, denote by $W_0 \in \Omega$ the composite collection of events that are dropped out of any queue within a given LDSS's QN at the defined but arbitrary starting time of t_0. Defining $T : \Omega \to \Omega$ as the dynamical system of interest, where $T[.]$ is the time-shift operator and $\mu : \Omega \to \Re^+$. Then it is obvious that $\mu(W_0) > 0$ given that actual events were dropped out of their respective queues at t_0, as per the definition of W_0's construction. By applying standard DST it is then the case that,

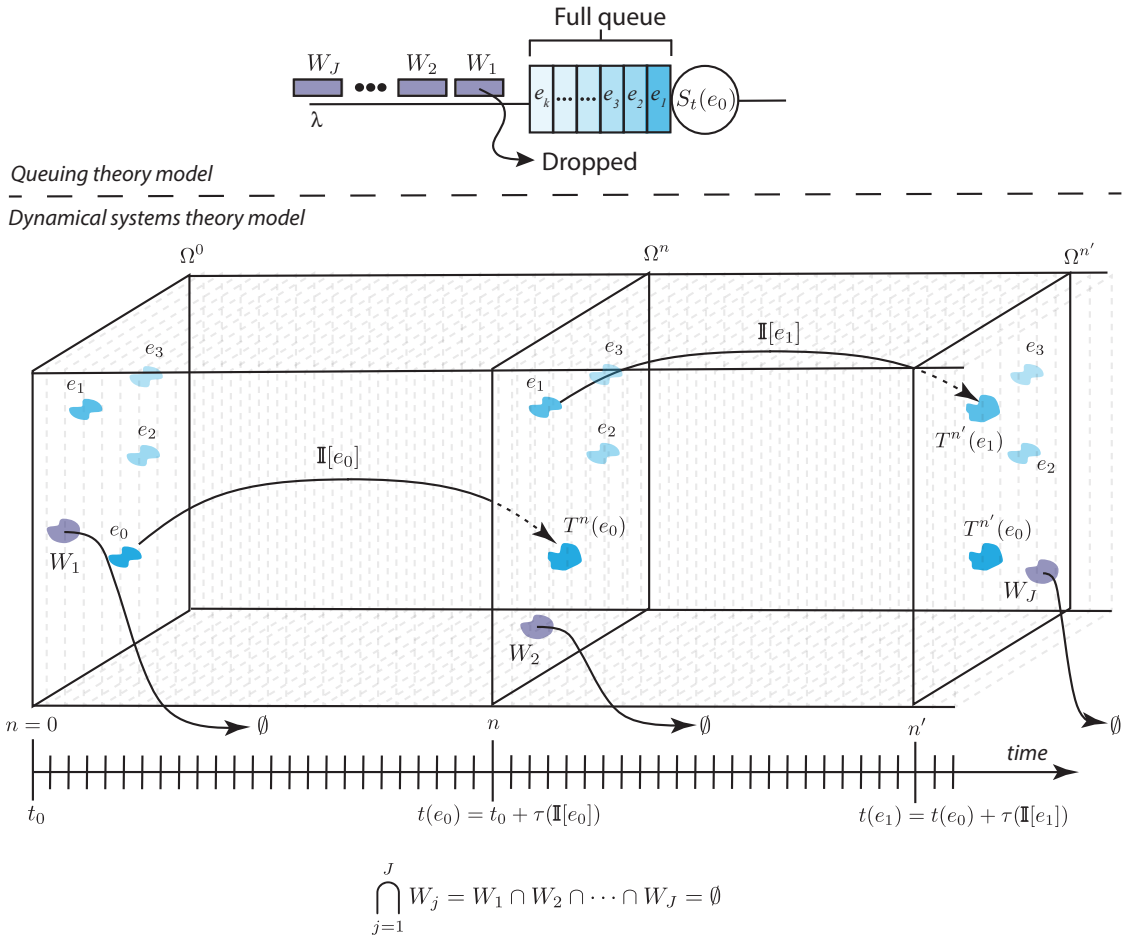


Figure 5.6: Dropped events in a single queue

$$\begin{cases} W_1 = T[W_0] \\ W_j = T[W_{j-1}] = T^j[W_0] \end{cases} \tag{5.17}$$

These W_j then denote the iterative set of queue drop events occurring at each subsequent t_j time step. Importantly, no presumption exists that the W_j are related to or dependent on each other. The W_j queue drops arise as independent events. But, these time domain queue drop events can be collected in the composite set $\mathcal{W} = \{W_j | j = 1, \ldots, J\}$, denoting the iterative collection of all queue drop events occurring within the given QN for $t_j \in [0, T_j]$. If no queue drop event occur within any queue within the LDSS's defined QN model for some $t_j \in [0, T_j]$ then $W_j = \emptyset$ where for simplicity but without a loss of generality t_j is assumed discrete.

As each $W_j \in \mathcal{W}$ denotes events that have "dropped" out of the QN, i.e., events that can no longer produce any effects on the QN, it is clear that $W_0 \cap W_j = \emptyset$ for all j. Moreover, it is equally clear that $W_j \cap W_{j'} = \emptyset$ for all $j \neq j'$. Additionally, as the W_j denote actual events that were within the QN until they were "dropped", it must also be the case that $\mu(W_j) > 0$ for all t_j unless $W_j = \emptyset$ given that $|\Omega| < \infty$, as per the EMM's σ-finite measure space description of the QN.

For simplicity and without a loss of generality, the new set $\mathcal{W}^* \subset \mathcal{W}$ can be produced simply by removing all of the $W_j \in \mathcal{W}$ for which $W_j = \emptyset$. As queue drops are assumed to have occurred in the system, it must be the case that $|\mathcal{W}^*| > 0$ and, if queue drops are on-going, that $|\mathcal{W}^*| \to \infty$ in the limit as $j \to \infty$. Without a loss of generality, $\mathcal{W}^*$ can be considered in the subsequent analysis.

Definition 5.3.1 (**Wandering Set of non-zero Measure [254]**)**.** Define a dynamical system $T : \Omega \to \Omega$ with measure $\mu : \Omega \to \Re+$ over a σ-finite measure space. Define a set $\mathcal{W}$ such that $\mathcal{W} = \{W_j = T^j[W_0] \mid j = 1, \ldots, J\}$ and $W_0 \in \Omega$. Then $\mathcal{W}$ is a:

a. *Wandering set* if for all $W \in \mathcal{W}$ it is the case that $W \cap T^j(W) = \emptyset$ for all j where $T^j(W) \neq T^{j'}(W)$ for all $j \neq j'$ and $\mu(W) > 0$ [270; 271].

b. *Weakly wandering set* if for all $W_j \in \mathcal{W}$ where $W_{j+1} = T[W_j]$, it is the case that $W_j \cap W_{j+1} = \emptyset$ and $\mu(W_j) > 0$ and $\mu(W_{j+1}) > 0$ [270].

Clearly, if $\mathcal{W}$ a wandering set then it will also be weakly wandering, although weakly wandering sets need not be wandering sets. Moreover, Theorem 3 Statement 4 requires that there be no weakly wandering sets of non-zero measure as per (b), whereas traditional Markovian analyses [223] via its reliance on theorems such as Poincaré Recurrence Theorem [273] typically requires that there be no wandering sets of non-zero measure, as per (a). The distinction between (a) and (b) is important as it denotes whether the defined recurrence requires that any $A \in \Omega$ can be passed through either an infinite number of times, as per (a), or at least once, as per (b), in the limit of $T^j[A]$ as j goes to infinity. Matrix-based Markovian solution approaches generally require that (a) holds. With respect to assessing performance predictability, we are able to show when the more restrictive (a) does (or does not) hold and, hence, consequently when (b) also does (or does not) hold.

Theorem 4. *(Claim 1).*
Within QN modelled LDSSes, the on-going occurrence of queue drop events within the QN creates wandering sets of non-zero measure with respect to the executing run-time LDSS instances for:

(i) The dynamical system defined over the time-shift transform for a given LDSS deployment instance, and

(ii) The dynamical system defined over the ensemble of possible run-time deployment instances of the same (or sufficient similar) LDSS.

As such a LDSS experiencing on-going queue drop events will not be BET-complaint and, therefore, will not exhibit predictable run-time behaviors for any performance measure $x_k(t)$ impacted by the produced wandering sets under the definition of Eq. (5.14)

Proof. The proof of the above theorem is as follows:

(i) Time-shift Dynamical System:

Clearly, it must be shown that the set $\mathcal{W}$ defined above for the run-time queue drop events meets the above (a) and (b) conditions that define wandering sets of non-zero measure. As discussed above, the composite set of specific run-time event instances dropped from any queue in the QN at any time t_j can be denoted by $W_j \in \Omega$. For all t_j for which no queue drop occurs anywhere in the QN then $W_j = \emptyset$. By DST, if $T[.]$ is the time-shift transform then $W_{j+1} = T[W_j]$ with $T[.]$ for all $j > 0$, assuming t_0 is chosen as the arbitrary start time of the analysis.

By definition, $\forall t_{j'}, t_j$ for which $t_{j'} > t_j$, it must be the case that $W_j \cap W_{j'} = \emptyset$. Moreover, as each W_j represent real events dropped out of the system and $|\Omega| < \infty$, then for all j it must be the case that $\mu(W_j) > 0$. Hence, the set $\mathcal{W} = \{W_j | \forall j = 0, 1, 2, \ldots\}$ is a wandering set in limit as j goes to infinity provided queue drop events are on-going within the system. As $\mathcal{W}$ is a wandering set it is also a weakly wandering set. Moreover, it is clear that $W_j \cap W_{j+1} = \emptyset$ for all $j > 0$.

Hence due to the existence of a wandering set of non-zero measure the LHD of BET cannot converge to a singular scalar value as which value arises depends on how the denoted wandering set evolves during the given deployment instance of

the LDSS's run-time execution. More particularly, the size of the power set of $\mathcal{W}$ describes the cardinality of set of possible $x_k(t)$ cdfs that can be produced.

(ii) Ensemble Dynamical System

Based on the above time-shift transform DST analysis, define a new set $\mathcal{W} = \{\mathcal{W}^k | k = 1, 2, \ldots\}$ as the collection of wandering sets of non-zero measure that occur for across all of the k run-time possible deployment instances of the LDSS under analysis. By definition, the collection of such $\mathcal{W}^k = \emptyset$ occurrences will themselves be of measure zero in the limit as $k \to \infty$, given the above presumption that queue drops are both occurring and on-going.

The dynamical system of interest is now the one defined across the ensemble of k run-time deployment instances of the LDSS such that $T[.]$ spans the ensemble. For any time t_j within any k instance there must exist a $W_j^k \in \mathcal{W}^k$ as per the above definitions, where t_j references the absolute run-time within each LDSS's own instantiation. More specifically, each such t_j can be viewed as a count of the system-wide clock ticks that have occurred since the k^{th} instance of the LDSS began its execution. It should be noted that this definition of the t_j's is different and distinct from the notion of there being a global absolute reference time clock accessible to and across all LDSS instances, as such a definition would be more restrictive and not allow different LDSS instances within the ensemble to be instantiated at different absolute start times.

As such, $\forall\ k, k',$ and t_j, if $W_j^k \cap W_j^{k'} \neq \emptyset$ it must be the case that the exact same specific event instances have been dropped by the exact same queues at the exact same point in each of the instantiated LDSS's run-time executions. In the case of weakly wandering sets, this would need to occur at least for some of the k run-time instances, whereas for wandering sets, it would need to occur across all instances. Clearly, this could occur if the system always entered into an overload condition relatively quickly after instantiation. But this would then simply be a "broken system" for which run-time software system performance predictability itself is no longer a meaningful objective, i.e., as discussed previously, this system would be in the class of "failed systems" that are within the uninteresting trivial boundary cases within conducted analysis.

Clearly, if all of the W_j^k can exactly match for all k and j then no wandering sets can exist across the ensemble of the LDSS's possible run-time executions.

Define this specific perfectly matching $\mathcal{W}$ set by $\mathcal{W}^*$. In the limit as k and j go to infinity the number of possible $\mathcal{W}$ sets itself is countably infinite. Hence, $\mu(\mathcal{W}^*) = 0$ across all possible ways queue drops can occur within the LDSS across all time and across possible instantiations of the LDSS. As such, for all other cases there must exist $W_j^k \cap W_j^{k'} = \emptyset$ for $k \neq k'$ for every fixed t_j. As such these sets form wandering sets of non-zero measure over the ensemble and, hence, also form weakly wandering sets of non-zero measure.

Clearly, the above discussion is not impacted for any k and t_j for which no queue drops occur, as $W_j^k = \emptyset$ in this case, as per the definition of the W_j^k. Hence, $W_j^k \cap W_{j'}^{k'} = \emptyset$ for all k' and $t_{j'}$. Therefore, the only case where wandering sets of non-zero measure no longer arise are when queue drop event no longer ever occur. In which case the LDSS will transition into being a BET-compliant system possessing predictable performance measures. The case of interest though is when queue drops are occurring and are on-going.

Hence, when a LDSS has occurring and on-going queue drop events the RHS of BET is also non-convergent, as it then denotes a distribution and not a scalar. Therefore, as per (i) and (ii), neither the LHS nor RHS of BET converge to scalar values. Hence, a LDSS experiencing on-going queue drop events cannot be formally BET-compliant. Moreover, the cardinality of the family of possible $x_k(t)$ cdfs produced by the system is now given by the cardinality of the power set of $\mathcal{W}$ defined across the ensemble.

□

Observation 1:

It should be noted that Theorem 4 only states that the $x_k(t)$ performance measures impacted by the wandering sets will not exhibit predictable behaviors. It does not state that other $x_{k'}(t)$ $k \neq k'$ performance measures cannot independently arise that are unaffected by the generated wandering sets. Such $x_{k'}(t)$ would exhibit statistically predictable run-time behaviors whereas, concurrently, the $x_k(t)$ measures would not. This observation highlights the behavioral complexity inherent in modern LDSS deployments.

Observation 2:

Theorem 4 also highlights that it is possible for a LDSS to transition to and from being BET-complaint for a given $x_k(t)$ performance measure. A LDSS's run-time behavior can cause a given $x_k(t)$ measure to cross back and forth over the mathematical "line-in-the-sand" denoting BET-compliance due to the specific nature and characteristics of the produced queue drop wandering sets. From a software engineering perspective, the nature of when and where queue drop events occur though is generally easily measurable in real-world operational systems.

Observation 3:

Finally, as per Theorem 4, it is the production of wandering sets of non-zero measure that give rise to a loss of LDSS behavioral predictability. As such the number of behavioral regimes a given LDSS could exhibit for a given $x_k(t)$ performance measure is given by the power set over all possible W_j^k occurrences for all k and all j. For any reasonably large-scale LDSS, even presuming a finite ensemble and finite time, the cardinality of this power set, is finite but exceedingly large. The pragmatic effect of this is that a continuum of possible cdfs are producible by any given LDSS for any given $x_k(t)$ measure. Hence, developing a bound on this cdf continuum would generally not be possible.

The above discussions leads to the following software engineering design rule which is required if run-time LDSSes are to support predictable performance measures relative to issue (1) above.

Definition 5.3.2. (**Software Engineering Design Rule 1**)
Within a deployed LDSSes, queue drop events must be actively managed and mitigated if the resulting LDSS's run-time performance measures are to be statically predictable in accordance with Eq (5.14).

Moreover, it is possible that observably predictable performance measures occur in LDSSes experiencing occurring and on-going queue drop events, but only when at least one of the following condition occures.

(i) The given $x_k(t)$ performance measure arises independently of and is unaffected by the queue drop events.

(ii) The rate at which queue drop events are occurring is sufficiently low that the formal loss of BET-compliance within $x_k(t)$ is not detectable via appropriately chosen data-driven statistical goodness-of-fit tests.

(iii) The nature and characteristics of differences between the produced wandering sets are such that these differences are not detectable via appropriately chosen data-driven statistical goodness-of-fit tests.

Of these cases, (i) can commonly occur within real-world LDSSes for a number of common performance measures, such as round-trip delay, that by definition require that all the underlying events giving rise to the measure have successfully transited through all the relevant LDSS queues. Additionally, (ii) may occur in real-world LDSSes but it would be largely restricted to appearing within quiescent or near-quiescent LDSSes and deployment environments, where these are well-known to generally produce predictable performance measures. Hence (i) and (ii) are largely trivial boundary cases of the above DST analysis. Finally, although (iii) is theoretically possible within real-world LDSSes, it is highly unlikely as it would exist as a measure zero outcome within any sufficiently large-scale LDSS run-time deployment.

Within modern container-based cloud deployment environments, Software Engineering Design Rule 1 can be easily and relatively trivially run-time implemented by simply viewing each container as servicing a queue and instrumenting the container to count the items (or events) waiting to be processed. As the count of waiting (or queued) events increases towards a selected performance threshold, conditioned on expected workload volumes, then the container can serve off a duplicate container and establish the required load balancing. This type of active container management can include a "self-terminate" once duplicate containers have detected themselves to be in starvation for a selected time period. In this manner, a container-deployed software system could dynamically increase (or decrease) its required resource footprint in a fully distributed manner so as to ensure the overall LDSS remained within a BET-compliant behavioral regime, at least with respect to the impacts of queue drop events.

As such, Theorem 4 and Software Engineering Design Rule 1 provide the mathematical insights and software engineering requirements underlying increasingly common in-industry practices such as implemented in whole or in part within modern dynamic resource management frameworks such as Apache Storm, Apache Spark, Kubernetes and Docker Swarm. To our knowledge, the DST mathematical under-

pinnings of these types of solutions and their necessity for achieving run-time LDSS performance measure predictability has not been previously formalized or highlighted.

5.3.1.2 Impacts of Reliable Protocols

Clearly, as queue drop events occur then, when reliable protocols are used, these protocols whether within the network layers or application layers actively act to re-instantiate the lost (or dropped) events. Importantly, under formal DST contexts, these recreated events are not identical to the dropped events but exist as new events created into the LDSS at a later time and then posses their own time periods under which they are serviced by the LDSS. Moreover, for simplicity with respect to the QN model, it will be assumed that all methodologies by which dropped events are regenerated or otherwise responded to fall into the general class of the response actions created or initiated by reliable protocols, i.e., a very broad definition of *reliable protocols* is used. This broad definition though is appropriate to the QN model approach in that what is of interest is any and all activities that cause dropped events to be recreated into the QN at some later time.

In a similar manner to the above Section 5.3.1.1 discussion, a set $\mathcal{W}_{\text{rel.}}$ such that $\mathcal{W}_{\text{rel.}} = \left\{\mathcal{W}^k | k = 1, 2, \ldots\right\}$ can be defined over each of the k ensemble instantiations of the LDSS such that $W_j^k \in \mathcal{W}_{\text{rel.}}^k$ represent the events reproduced at each time step t_j by the reliable protocol actions, i.e., in response to the need to recover a previously dropped event(s) that occurred at some time $t_{j'} < t_j$. If no events are reproduced for some t_j and k then $W_j^k = \emptyset$. Importantly, the events within the so defined W_j^k exist as real events that are re-queued in to the QN to then be processed. Within the DST context, the time evolution of these W_j^k events will arise independently of what may have occurred for the dropped events had they not dropped. Hence, these W_j^k sets are different and distinct from the W_j^k queue drop sets described in Section 5.3.1.1. These new events themselves though may end up being dropped by the QN while being processed, triggering an addition $W_{j'}^k$ set at the future time $t_{j'}$. Such issues are reasonably likely to occur in practice as queue drop events tend to arise as a LDSS deployment moves towards overload conditions. Hence, the actions of reliable protocols can be viewed as creating additional workloads via re-generated events into an LDSS already experiencing overload conditions. Clearly, identical arguments to those of Section 5.3.1.1 can now be applied to show that this new $\mathcal{W}_{\text{rel.}} = \left\{\mathcal{W}_{\text{rel.}}^k | k = 1, 2, \ldots\right\}$ set produced by the reliable protocol actions is itself a

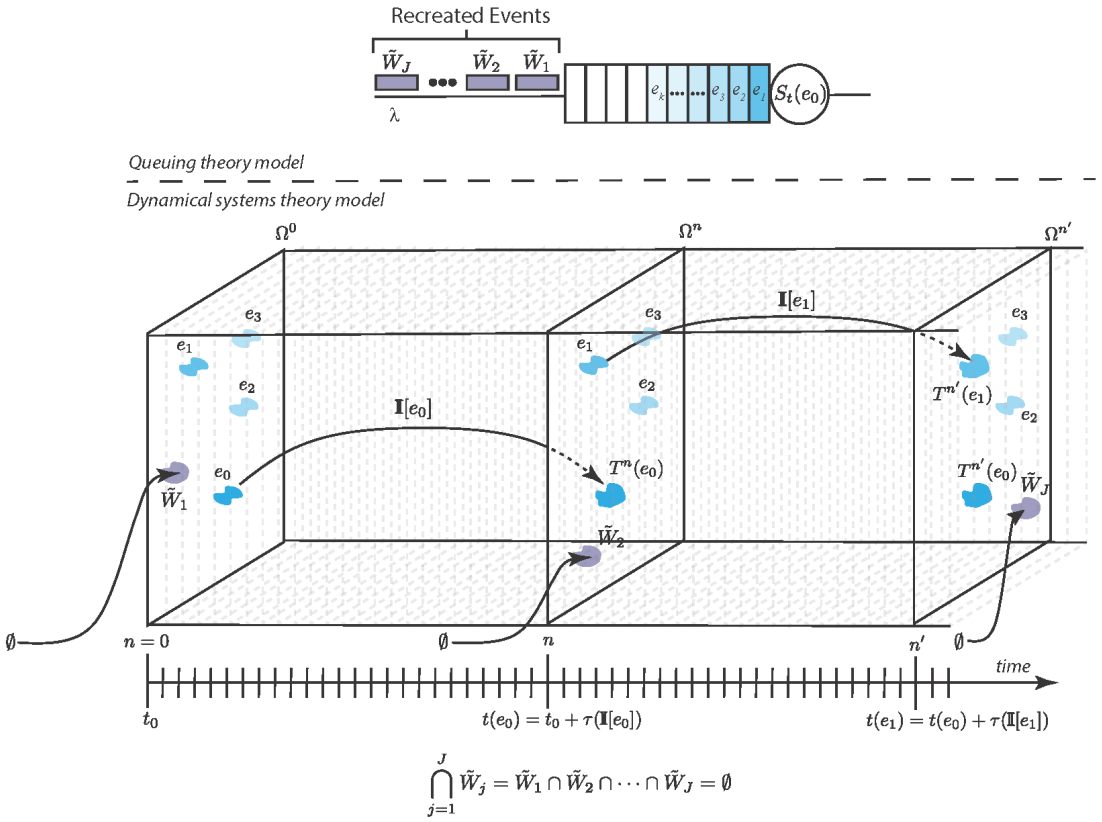


Figure 5.7: Re-created events in a single queue

wandering set of non-zero measure across both the time-shift dynamical system and the ensemble dynamical system. Hence, Theorem 5 can be stated as follows:

Theorem 5. *(Claim 2).*
Within QN modelled LDSSes, the actions of reliable protocols to recover events lost due to the on-going occurrence of queue drop events within the QN creates wandering sets of non-zero measure with respect to the executing run-time LDSS instances for:

(i) The dynamical system defined over the time-shift transform for a given LDSS deployment instance, and

(ii) The dynamical system defined over the ensemble of possible run-time deployment instances of the same (or sufficient similar) LDSS.

As such a LDSS using reliable protocol actions to recover from on-going queue drop events will not be BET-complaint and, therefore, will not exhibit predictable run-time

behaviors for any performance measure $x_k(t)$ impacted by the produced wandering sets under the definition of Eq. (5.14).

Proof. Assume that the wandering sets of non-zero measure of Theorem 5 exist within the QN model of the LDSS. Define this wandering set of QN dropped events by $\mathcal{W}_{\text{dropped}} = \{\mathcal{W}^k | k = 1, 2, \ldots\}$, as per Section 5.3.1.1. Again, as per above, each W_j^k set occurred at time t_j within the k^{th} run-time instance of the given deployed LDSS. Hence, each such W_j^k is composed of a set of events $\{e_n | n = 1, \ldots, N_j^k\} \in W_j^k$ that have been "lost" from the LDSS's executing QN model at times $t_{j'} \leq t_j$. These e_n events can no longer generated any future events with in the QN model's execution. Hence, the reliable protocol(s) must regenerate them if they are not to be permanently lost from the system.

Denote by $\tilde{e}_{j''}^n$, the reliable protocol reproduction of any such "lost" events at some time $t_{j''} > t_j$, later in the QN's execution, where for simplicity the k subscript is dropped. Clearly, the time between an event being "lost" and it being reproduced will be a stochastic process, denoted by $t_\Delta = t_{j''} - t_{j'} > 0$. But, for each t_j time step in the LDSS's execution the set of reliable protocol reproduced events can be collected and assigned into a set $\tilde{W}_j^k$, where $\tilde{W}_j^k = \emptyset$ if no events were reproduced for a given time t_j. Clearly, $\mu(\tilde{W}_j^k) > 0$ for all times t_j and run-time instances k where such reproduced events do occur given that real events are reproduced and re-queued.

Clearly, these $\tilde{W}_j^k$ sets themselves form wandering sets of non-zero measure with respect to both the time-shift and ensemble dynamical systems. More particularly, the formal proof follow identically from the formal proof of Theorem 4 above, the only difference being the nature and characteristics of how the respective wandering sets $\mathcal{W}_{\text{dropped}}$ and $\mathcal{W}_{\text{rel.}}$ are produced. □

Observation 1-3 of Theorem 4 also continue to hold and apply for Theorem 5, where the actions of the reliable protocols replace those of the queue drop events. It should be noted that within the context of the QN model as applied in this work, reliable protocol actions only occur as a consequence of queue drop events having occurred. Hence, this Section highlights that the $\mathcal{W}_{\text{rel.}}$ wandering sets a secondary type of wandering set across both the time-shift and ensemble dynamical system and, therefore, will act to decrease the overall statistical predictability of any performance measures $x_k(t)$ also impacted by $\mathcal{W}_{\text{rel.}}$. Importantly, the reliable protocol produced wandering sets can be seen to further increase the cardinality of the family of possible $x_k(t)$ cdfs by the additive impacts of the power set of $\mathcal{W}_m boxrel.$.

Reliable protocols are clearly a critical necessity within modern LDSS operations. The above discussion does not seek to negate their usefulness, but only to assess the degree to which their use impacts performance predictability as per the Eq. (5.14) definition. More specifically, a likely general presumption would be that reliable protocols add to or improve run-time performance predictability. The DST analysis though produced the initial counter-intuitive result that reliable protocols produce a secondary form of wandering sets of non-zero measure, which the acts to decrease performance predictability. This result can be explained by observing that reliable protocols add regenerated workloads into LDSSes already experiencing queue drop events, i.e., already entering into overload conditions. Hence, from the performance predictability perspective the net impact of reliable protocols is to increase the workload within a LDSS already experiencing overload, which will then act to further reduce its performance predictability as per the above DST analysis.

Definition 5.3.3. (**Software Engineering Design Rule 2:**)
Within a deployed LDSSes, the use of reliable protocols to recover from queue drop events will generally result in the LDSS's run-time performance measures having decreased statistical predictable. Hence, reliable protocols should be used where they are required and necessary, but they should not be overused where and when they are not required.

5.3.2 Production of Variations in the Event Spaces and Non-invariant Measures (Theorem 3 Statement 2)

Section 5.3.1 addressed the issues inherent in the production of wandering sets within dynamical system event spaces. By contrast, this Section explores the issues inherent of how the $T[.]$ mapping and $\mu(.)$ measures associated with each dynamical system are impacted by OS scheduling and resource utilization timing issue within LDSSes and their deployments. More specifically, it is the sets A inscribing $x_k(t)$'s cdf given by $P_{k,j}[x(t) < A]$ that are of interest, where for completeness $A \in \Omega_{j,k}$ as the events spaces themselves can change both over time and ensemble instances. Given that $x_k(t)$'s cdf exists for all k and times t_j, the Axioms of Probability must also hold and, therefore $\mu(\Omega_{j,k}) = 1$ for all k and t_j. As was highlighted in Section 5.3 above, the QN-related software engineering issues that can give rise to such issues are due to:

i The execution timing impacts of OS scheduling regimes and, particularly those of commonplace fair scheduling regimes, and

ii The execution timing impacts of system resource utilization, loading, and contention issues.

The (i) and (ii) issues are different and distinct, with each impacting the timing characteristics separately and, potentially, concurrently affecting how events are processed and executed. The sub-Sections below discuss (i) and (ii) in turn. To simplify the discussions and analyses, it is useful to consider the impacts of (i) and (ii) on a particular or specific form of $x_k(t)$ performance measure. In assessing BET-compliance, it suffices to show that non-compliant $x_k(t)$ measures can exist, which then allows for the development of the software engineering design rules required if BET is to hold for any given $x_k(t)$ run-time performance measure.

More directly, assume that $x_k(t)$ denotes the average response time associated with a given class of events $E = \{e_p | p = 1, \ldots, P\}$, such as web page load events, database write or read events, etc. As $x_k(t)$ is an average it must be computed over some set of time intervals (or windows) $W = \{W_0, W_1, W_2, \ldots, W_J\}$ where $W_j \cap W_{j'} = \emptyset$ and $|W_j| = |W_{j'}|$ for all $j \neq j'$. Define the start time of an event (or window) by $t_0(.)$ and its duration by $\Delta(.)$, such that each event e_p's completion time is given by $t_0(e_p) + \Delta(e_p)$ and the end time of each W_j window is given by $t_0(W_j) + \Delta(W_j)$. The average response time $x_k(t)$ performance measure is then formally given by,

$$x_k(t) = \begin{cases} \frac{\sum_{\forall e_p} 1_{W_j}(e_p) \cdot \Delta(e_p)}{\sum_{\forall e_p} 1_{W_j}(e_p)} \\ \text{undefined} \qquad\qquad \text{otherwise} \end{cases} \tag{5.18}$$

where $1_{W_j}(.)$ is an index function that equals 1 if $[t_0(e_p), t_0(e_p) + \Delta(e_p)] \subseteq W_j$ and 0 otherwise. Hence, $x_k(t)$ simply counts the number of events that have completed their executions within each defined time window and reports the average of the completion times recorded for each window. Typically, the time windows W_j will be fixed and set within the given LDSS run-time deployment and management system, i.e., as per second response time averages, per hour averages, etc. The complexity comes in that the $\Delta(e_p)$ event durations will most generally exists as stochastic processes that are both time and ensemble dependent. Hence, this is more accurately denoted as $\Delta_{k,j}(e_p)$, whereas k denotes ensemble but j now denotes any time $t_j \in [t_{k,0}(e_p), t_{k,0}(e_p) + \Delta_k(e_p)]$ given that how event's evolve over time can be impacted by system and deployment

environment issues arising throughout their execution durations. Importantly, events are not "recorded" within the $x_k(t)$ average until they have completed their executions. As per common real-world measurement regimes, partially completed events do not give rise to changes in the reported average response times.

The issue of interest in the subsequent sub-Sections is that how stochastic variations in $\Delta_{k,j}(e_p)$ arise and the impacts these have on the $P_{k,j}[x_k(t) < A]$ sets that define $x_k(t)$'s cdf. More particularly, Statement 2 of Theorem 3 requires that the only members $A \in \Omega$ and $T^{-1}[A] \in \Omega$ with $\mu(A\Delta T^{-1}[A]) = 0$ must be such that $\mu(A) = 0$ or $A = \Omega$ or, equivalently, $\mu(A) = 1$. It should be noted that Theorem 3 requires that $T[.]$ be measure preserving but, unlike the formal stationarity definition one-to-one and onto is not required. To simplify but without a loss of generality, the event spaces associated with each of the W_j windows will be denoted as Ω_j wherein each of these windows will be assumed to denote the same time period across all of the ensemble runs, e.g., a sequence of one second windows beginning at the start of execution for the LDSS's k^{th} deployment instance.

5.3.2.1 OS Scheduling Impacts

In general, with respect to the QN model, OS scheduling regimes can be modelled in terms of enacting stochastic On/Off switches for each impacted queue. Within commonplace fair OS scheduling, the durations of each On (or Off) instance will depend on the number of active executing processes currently executing within the given OS layer, which is time-varying, stochastic and load-dependent. By contrast, real-time OS scheduling, would place an upper bound on when each given process will have received its allocation of processing resources. From a QN perspective, under real-time scheduling, queues continue to exhibit On/Off sequences, but these now arrive systematically and consistently as opposed to stochastically under fair OS scheduling. Importantly, within the real-time OS scheduling periods, i.e., at finer time resolutions, the QN On/Off processes remain time variant, stochastic, and load dependent. It is only at time resolutions at or above the real-time OS's enforced time constraints where these behaviors change.

As with Section 5.3.1 above, the analysis will begin with the time-shift dynamical systems before proceeding to show how the relevant issues persist across the ensemble defined dynamical system. Additionally, the focus will be on assessing the

average response time $x_k(t)$ performance measure described above, although this can be replaced by any other desired $x_k(t)$ performance measure.

Theorem 6. *(Claim 3):*
Within QN modelled LDSSes, the actions of OS fair scheduling within non-quiescent processing regimes produces stochastic per-event driven DST transforms leading to variation in the observed event spaces for:

(i) The dynamical system defined over the time-shift transform for a given LDSS deployment instance, and

(ii) The dynamical system defined over the ensemble of possible run-time deployment instances of the same (or sufficient similar) LDSS.

As such a LDSS that relies on fair OS scheduling within its deployment regime(s) will not be BET-complaint for any $x_k(t)$ performance measures impacted by these stochastic variations. Therefore, such $x_k(t)$ measures will not exhibit predictable run-time behaviors under the definition of Eq. (5.14)

Proof. **i)Time-shift Dynamical System:** As per the proof of Theorem 4, assume that $T[.]$ denotes the time-shift transform associated with an LDSS's k^{th} deployment instance and that a measure $\mu : \Omega \rightarrow \Re^+$ is defined for this dynamical system. As per the above description, associated with each W_j window of time is an event space Ω_j denoting the composite of events A that inscribe $x_k(t)$'s cdf over each W_j window. Clearly, for all j it must be the case that $\mu(\Omega_j) = 1$ and, moreover, the Axioms of Probability must be met for $\mu(.)$ across all Ω_j such that $\mu(\Omega_j) = 1$. The A's that inscribe $x_k(t)$'s cdf, as per Eq. 5.18, denote the distribution of the number of events completing within the W_j window. Under fair OS scheduling, the stochastic, time-varying, and load dependent nature of the queues On/Off behaviors will cause subsets of event completions to be shifted stochastically forward in time. This entails that there must be events $A' \in \Omega_{j'}$ for $j' > j$ which now complete in windows $W_{j'}$ where in a quiescent system would have completed in W_j such it would have been the case that $A' \in \Omega_j$. As these are real events being executed $\mu(A') > 0$. Hence, if event completions are being stochastically shifted forward in time, then there must be cases where $\Omega_j \neq \Omega_{j'}$ for $j' \neq j$. Moreover, these cases themselves cannot be measure zero. Hence, $T[.]$ cannot be a bijective transform nor will it be a

measure preserving transform for all $A \in \Omega_j$ for all j. Hence, the LHS of BET cannot be a convergent sum.

If the fair OS scheduling is replaced by real-time OS (RTOS) scheduling, then all $\Delta(e_p)$ executions times would be guaranteed to complete within a defined time period, denoted as T_{RTOS}, such that $\Delta(e_p) \leq T_{RTOS}$ for all successfully completed e_p. If the W_j measurement windows are then set to cover T_{RTOS} or greater intervals, then event completions can no longer shift forward in time. Hence, at time resolutions of T_{RTOS} or greater it will be the case that $\Omega_j = \Omega_{j'}$ for $j' \neq j$. Moreover, if the incoming workload is stationary and ergodic then statistically similar sets of events will be completed during each W_j. If resourcing utilization issues are set aside, then all sufficiently similar events will complete in sufficiently similar times and, hence, $T[.]$ will necessarily be bijective and measure preserving at (or above) the selected real-time OS time scale (or resolution). Below this time resolution, such arguments cannot be applied and the system will appear to be non-stationary as per the fair OS scheduling case.

ii) Ensemble Dynamical System: Now consider the dynamical system defined over the k ensemble instances, but with the same W_j, Ω_j, and A definitions as per (i) above. It is now assumed that the LHS of BET always converges to a scalar with the question being to assess whether the RHS of BET also converges to a scalar and always to the same scalar, i.e., $x_k(t)$ is now presumed to be a statistically stationary process for all k albeit potentially a different process for some (or all) k. To clarify this issue, the notation Ω_k is now used to highlight that the potential of the event space changing over the k ensemble runs, i.e., time has been removed from the analysis as result of presuming the LHS of BET is convergent. Again, the A's of interest are the ones that inscribe the cdfs although in this case this is for $P_k[x_k(t) < A]$ given the assessment is across the dynamical system defined over the ensemble. More specifically, $T[.]$ defines this mapping across ensemble cdfs with $\mu(.)$ being the measure over the Ω_k event spaces. As per above, under fair OS scheduling the $\Delta_k(e_p)$ will be a stochastic process drawn from unknown and unbounded underlying distributions. More particularly, as per the k-subscript, these underlying distributions are likely to change considerably from run to run, i.e., with k. Hence, again, it will be the case that there exist $\Omega_k \Delta \Omega_{k'} \neq \emptyset$ for at least some $k \neq k'$. More particularly, as per DST define $\Omega_{k'} = T^{-1}[\Omega_k]$. Hence, it will be the case that there exist

$A \in \Omega_k$ for which $T^{-1}[A] \in \Omega_{k'}$ and $A' \in \Omega_k$ for which $T^{-1}[A'] \notin \Omega_{k'}$. Clearly, as such it cannot be the case that for all $A \in \Omega_k$ such that $\mu(A\Delta T^{-1}[A]) = 0$ that $\mu(A) = 0$ or $\mu(A) = 1$ given that there must also be $A' \in \Omega_k$ for which $\mu(A') > 0$ as a consequence of the Axioms of Probability. As such the RHS of BET cannot converge to a scalar for all A when $\Delta_k(e_p)$ is itself a stochastic process, as must be the case for fair OS scheduling within non-trivial operational regimes.

As per (i) above, the same arguments can be used if fair OS scheduling is replace with RTOS scheduling in that $\Delta_k(e_p) \leq \Delta_{max(RTOS)}$ for all e_p and all k. Hence, when suitable windows are used the RHS of BET can then be convergent and BET-compliance can be achieved, but at the observation time frames of the applied RTOS scheduling. If soft RTOS scheduling is used, then BET-compliance will again be broken for any performance measures involving e_p events that are late to complete, as defined within the given RTOS implementation.

□

As per the above discussion, in the DST sense LDSSes present extremely complex run-time behaviors within their commonplace fair OS scheduling deployment regimes. This is further complicated by multiple OS scheduling layers being present and active within modern cloud computing regimes in which containers are deployed into leased VMs which then are task swapped by the server OS, given generally more active VMs exist with server than its count of cores required to service them, i.e., time-slicing of the VMs is a well-known cloud computing behavior.

Observation 4:

Clearly, the above fair OS scheduling issues impact all events being executed by the LDSS, although, potentially, to quite different degrees. Hence, these fair scheduling issues can lead to performance predictability issues within $x_k(t)$ performance measures that are *not* impacted by the Section 5.3.1 discussed wandering sets. Hence, the each of the Section 5.3.1 and 5.3.2 issues arise independently and, potentially, concurrently within LDSS run-time operations. As such, co-occurrences of the issues can significantly magnify the degree to which run-time performance predictability is lost for given $x_k(t)$'s, as will be seen within the Chapter 6 simulation results.

Observation 5:

Within small-scale systems or systems near their quiescent regimes, although $\Delta(e_p)$ will formally retain its stochastic nature under fair OS scheduling large variations in $\Delta(e_p)$ observed run-times for a given e_p or class of sufficiently similar e_p events are relatively unlikely. Moreover, in any LDSS where with suitably high probability given e_p's required processing time can be delivered by the relevant OS layers in sufficiently timely manners $x_k(t)$ performance predictability will not appear to have been lost, even though the system will formally remain asymptotically non-BET compliant. More formally, if an LDSS and its deployment environment are such that for all e_p required to be processes it is the case that almost always $\Delta(e_p) < \Delta_{min}(e_p) + \zeta$ then the LDSS's run-time performance measures will for practical purposes appear BET compliant, where $\Delta_{min}(e_p)$ is the minimum possible execution time required for event e_p and $\zeta > 0$ is a sufficient small positive constant.

Observation 6:

It should be noted that utilizing RTOS OS scheduling does not eliminate the run-time performance predictability issues. Instead, a ROTS merely enforced absolute time periods by which point each e_p event must have completed. Hence, if all e_p events are subject to RTOS scheduling, then it is only for time resolutions above that of the RTOS scheduling periods where BET-compliant $x_k(t)$ behaviors will arise. For narrower time period, the same $x_k(t)$ performance measure may exhibit non-BET compliant behaviors, as these finer time resolutions are outside of the RTOS's intended regime of control. Hence, the introduction of RTOS scheduling has the effect of also introducing the need for multi-resolution time series analysis in that the LDSS performance measures may exhibit quite different fine and coarse grain time resolution behaviors.

Observation 7:

Finally, if a LDSS is primarily processing very small duration e_p events, then issues similar to those of Observation 5 arise in that the LDSS performance measures could then appear to be BET-compliant. Moreover, if the LDSS has a complex mixture of small and long duration e_p events then this is likely to produce very poor $x_k(t)$ predictability behaviors. Mixing traffic types is well-understood to lead to poor performance regimes within Markovian analysis. Hence, the observation that mixed QN

traffic workloads should be avoided is not in itself new. The new insight relative to Theorem 6 above is that such mixing is also likely to also lead to further decreases in the $x_k(t)$ performance predictability, again as defined per Eq. (5.14).

This set of Theorem 6 generated observations lead to the following software engineering design rule.

Definition 5.3.4. (**Software Engineering Design Rule 3**)
Within a deployed LDSS, where possible, RTOS OS scheduling with hard scheduling regimes should be employed, particularly within LDSSes requiring run-time performance predictability, as fair OS scheduling regimes under typical non-quiescent operational regimes will innately give rise to non-predictable LDSS performance measures for all but quite trivial operations and workload scenarios.

5.3.2.2 Resource Utilization and Contention Impacts

As with OS scheduling issues, resource utilization and contention issues also impact the nature of the QN model's queues On/Off processes. More particularly, from the perspective of contention, assume a given process holds a lock (or semaphore) on a given resource. All other processes requiring that resource for their execution then cannot run until the lock is released, irrespective of the processing time allocated by the associated OSes for those queues. Hence, the impact of contention is to produce another stochastic process that overlays onto the per-queue level On/Off processes engendered through OS scheduling. In essence, resource contention leads to stochastic increases in the duration of the Off cycle periods, where such impacts are themselves time-dependent and arise at the per-queue level. These issues can continue to arise even if RTOS scheduling is enabled if careful consideration is also not given to resource utilization and contention issues. Hence, these issues exist as separate and distinct concerns to the Section 5.3.2.1 OS scheduling issues.

Within standard Markovian QN analysis under Little's Law [246] and, presuming, recurrence holds (Theorem 3 Statement 3), then the response time R_n of some queue n that uses resource m for its execution in its simplest case is given by:

$$R_n(e_p) = \begin{cases} \frac{D(e_p)}{1-U_m} & \text{if } U_m \in [0,1) \\ \infty & \text{if } U_m \geq 1 \end{cases} \tag{5.19}$$

where $D(e_p)$ is the service demand required to process workload e_p on queue n when resource m is at quiescent (or near quiescent) load and $U_m \geq 0$ is the normalized

utilization of resource m. Within real-world LDSSes, as $U_m \to 1$ then $R_n(e_p) \to \infty$ in the limit. But, within the theoretical analyses conducted in LDSS system design stage $U_m \geq 1$ can commonly arise, with this denoting a failed system in which extreme overloading will have rendered the system non-operational. Typically, operational utilization rates in the $U \in [0.2, 0.7]$ are deemed as providing a reasonable trade-off between response times and per-resource costs.

Within real-world LDSSes and deployment environments, per-resource utilization is itself a time varying stochastic process, more accurately denoted as $U_m(t)$. More particularly, within modern cloud environments physical servers may host 100+ concurrently active VMs, where this generally exceeds the count of their available processing cores. As such, VMs themselves are swapped in and out of being active by the base server OS, i.e., they ostensibly exist as heavy-weight processes under the server's base OS scheduling regime. Moreover, the cyclical activating and deactivating of these heavy-weight processes lead to contention issues within the server's hardware caches, physical drives, network services, etc. In particularly bad cases it can lead to drive thrashing, an issue that led to major cloud providers moving away from allowing database servers to be created using their lowest cost VM offerings.

As such, Eq. (5.19) is more accurately written for real-world LDSS operations as,

$$R_n(t|e_p) = \begin{cases} \frac{D(e_p)}{1-U_m(t)} & \text{if } U_m(t) \in [0,1) \\ \infty & \text{if } U_m(t) \geq 1 \end{cases} \tag{5.20}$$

wherein the assumption of Theorem 3 Statement 2 holding is now also removed. Hence, $R_n(t|e_p) \in (0, \infty)$ is itself now an unbounded stochastic process even when a real-world relevant utilization is presumed of $U_m(t) \in [0, 1)$. More particularly, the rate of change of the response time is then given by,

$$\frac{d}{dt}[R_n(t|e_p)] = \begin{cases} \frac{D(e_p) \cdot \frac{d}{dt}[U_m(t)]}{[1-U_m(t)]^2} & \text{if } U_m(t) \in [0,1) \\ \infty & \text{if } U_m(t) \geq 1 \end{cases} \tag{5.21}$$

This highlights the core non-linear nature of the LDSS performance predictability problem in that even relatively small changes in utilization past the approximate 0.7 threshold can quickly lead to the occurrence of effectively unbounded response times for all impacted events. When reliable protocols are in use, this is likely to generate even more events to be processed leading to yet higher resource utilization levels as the unbounded response times surpass reliable protocol time-out thresholds.

The overall impacts of resource contention and utilization issues is therefore to create an additive stochastic process effect that adversely impacts the e_p execution durations. Denoted this resource contention and utilization impacts as $\Delta_2(e_p)$ with the prior OS scheduling stochastic impacts now denoted as $\Delta_1(e_p)$. The overall timing impacts to the event e_p's execution can then simply be given by $\Delta(e_p) = \Delta_1(e_p)+\Delta_2(e_p)$, where both $\Delta_1(e_p)$ and $\Delta_2(e_p)$ are generally unknown and unbounded stochastic processes. Hence, the following theorem can be stated with respect to resource utilization and contention issues:

Theorem 7. *(Claim 4):*
Within QN modelled LDSSes, the impacts of resource contention and utilization issues within non-quiescent processing regimes produces stochastic per-event driven DST transforms leading to variations in the observed event spaces for:

i) The dynamical system defined over the time-shift transform for a given LDSS deployment instance.

ii) The dynamical system defined over the ensemble of possible run-time deployment instances of the same (or sufficient similar) LDSS.

As such a LDSS that experiences sufficient resource contention and utilization issues will not be BET-complaint for any $x_k(t)$ performance measures impacted by these stochastic variations. Therefore, such $x_k(t)$ measures will not exhibit predictable run-time behaviors under the definition of Eq. (5.14)

Proof. As per the above discussion, the impacts of the resource contention and utilization issues are to produce an unknown and unbounded stochastic process that impacts the execution duration of all effected e_p events, denoted as $\Delta_2(e_p)$. This combines with the OS scheduling issues to produce an overall unknown and unbounded stochastic process impacting the execution durations of all effected events defined by $\Delta(e_p) = \Delta_1(e_p) + \Delta_2(e_p)$, where $\Delta_1(e_p)$ denotes the OS scheduling impacts.

As the proof of Theorem 6 (Claim 3) is simply based on $\Delta(e_p)$ being an unknown unbounded stochastic process the identical proof applied for this theorem. Moreover, it is clear that the proof continues to hold even in the case where $\Delta_1(e_p)$ is itself deterministic, as effectively occurs under RTOS-based OS scheduling. □

Observations 4 through 7 continue to hold for resource utilization and contention based stochastic impacts on event execution durations. But these impacts can arise

independently of the observed OS scheduling based issues. This leads to the final software engineering design rule, namely,

Definition 5.3.5. (**Software Engineering Design Rule 4:**)
Within a deployed LDSSes, where possible resource contention issues should be minimized and resource utilization levels carefully monitored so as to ensure adequately bounded event execution times are occurring. Failing to manage such issues will tend to give rise to non-predictable LDSS performance measure for all but quite trivial operation and workload scenarios.

Importantly, the above software engineering design rules do not differ for what is pragmatically well-known in industry to give rise to better behaving LDSSes. The insight and contribution of this research though is that these software engineering design rules have now been placed onto and source out of formal DST mathematical foundations, i.e., they arise as necessities and not just "nice-to-have" LDSS operational features. Moreover, as the above issues exist as the complete set of issues that can arise with respect to the denoted QN model and each of the issues are shown to arise independently within the DST analysis, the conditions and concerns constitutes the necessary and sufficient set of issues that must be considered if formally BET-compliant $x_k(t)$ performance measures are to be ensured with run-time LDSS operations, as per the Eq. (5.14) definition of predictability.

5.4 Summary

In Chapter 5, our focus was on the formal investigation of software-centric systems ergodicity, spanning from simple embedded systems to complex distributed software systems in commercial clouds. Many modern software-centric technologies rely on using control theory and machine learning to solve challenges from predicting performance in large-scale distributed systems to analyzing the reliability of autonomous systems. The underlying assumption for all these methods is that the deployed software systems are ergodic and stationary, allowing them to predict future behavior based on past data.

We explained the modeling of stochastic runtime performance measures as dynamical systems, particularly macro-level observables obtained through averaging over specific time intervals, for example, average response times over 10-second or 1-minute periods. Then, we explored formal definitions of statistical stationarity and

ergodic theory, including Birkhoff's Ergodic Theorem (BET) and equivalent definitions of ergodicity. One of the most important theories of ergodicity (Theorem 3) is proposed by Peter Walters [254] in which he proposes four equivalent conditions for testing ergodicity and if a system fails to satisfy just one of these conditions, it loses its ergodicity. The core developed insight is that Statements 2 and 4 can be relatively easily applied to modern large-scale LDSS, whereas this has proven to be problematic within the literature for Statement 3. Hence, Statements 2 and 4 were shown to lead to a scale-agnostic DST and EMM-based QN analysis methodology.

The investigation of these conditions revealed the sources of unpredictable performance behavior, especially in large-scale distributed systems (LDSS) deployed in commercial cloud environments. Leveraging EMM, we applied DST rules to study the behavior of software systems under fair scheduling, server overloading, queue drops and reliable protocols with more focus on shared environments. We proved that all these factors led to challenges in maintaining ergodicity in deployed software systems.

As a result, we discussed how the application of RTOS and careful management of server utilization are essential for addressing the measure invariance challenges due to fair scheduling and server overloading. Furthermore, we need to make sure that queues are not becoming overly full to prevent the queue drops result in wandering sets of non-zero measure render the system non-BET compliant. The DST analysis led to the development of the four software engineering design rules required to ensure BET-compliance is maintained. These are summarized in Table 5.1 below.

Table 5.1: Software Engineering Design Rules

SEng Design Rule	**DST Context**	**Theory Issue**	**Pragmatic Action**
No. 1	Theorem 3 Statement 4	Queue drop events create wandering sets of non-zero measure.	Measure queue drop events and duplicate resources to minimize queue drops.
No. 2	Theorem 3 Statement 4	Reliable protocols create wandering sets of non-zero measure.	Measure rate at which reliable protocol actions are enacted and add resources to minimize occurrence rates.
No. 3	Theorem 3 Statement 2	Fair OS scheduling shifts event completions stochastically forward in time.	Where possible replace fair OS scheduling with RTOS scheduling.
No. 4	Theorem 3 Statement 2	Resource contention and high utilization rates shift event completions stochastically forward in time.	Where possible minimize resource contention and overly high utilization rates.

Chapter 6

Simulation-based Validation

In this Chapter a packet-level cloud-deployed LDSS simulation framework, based on OMNet++ and INet, is developed and used to experimentally validate the Chapter 5 theory results and insights.. To retain a real-world context the simulated exemplar LDSS is taken from an operational industry-held cloud deployed SaaS system. The Chapter begins by introducing this industry-held system. The details of the developed simulator platform and framework are then introduced. This is then followed by a relatively comprehensive set of experiments scenarios and results designed to test the exemplar LDSS under the composite set of the Chapter 5 Software Engineering Design Rules 1-4 and their combinations. This more thorough validation-based analysis is provided due to the quite strong claims of Chapter 5 i terms of when and why LDSS deployments and run-time behaviors can and cannot be BET-compliant.

6.1 Industry Case Study

The examplar case study is taken from the work of Robert O'Dwyer master's thesis [4] which evaluated the run-time performance behavior of a production-scale software system within a real-world cloud deployment environment. More specifically, this SaaS system handled hundred millions of mobile ad placements per month. In [4], a thorough statistical analysis was conducted across 10 identical real-world cloud deployments of the same LDSS executable, serving statistically identical incoming test workloads generated within the cloud environment. All testing occurred within a single day in the same commercial cloud facility. Given the costliness of such real-world performance envelope testing, [4] was limited to only 10 test runs. Consequently,

questions naturally arise regarding whether the observed predictability losses in the study would persist over a larger set of experimental runs or across other similar or dissimilar LDSSes. This Chapter addresses these concerns by expanding the Monte Carlo ensemble by an order of magnitude, simulating queue drops and hypervisor schedulers effects while also evaluating the relative impacts of reliable versus non-reliable protocols on LDSS performance.

The assessed application is a web application in which the user engages through a series of user enacted decision points, describable via a simple Markov chain as illustrated in Figure 6.1. The user progresses from the initial *http* request node in the chain through to subsequent nodes provided the required preconditions for the proceeding node have been met.

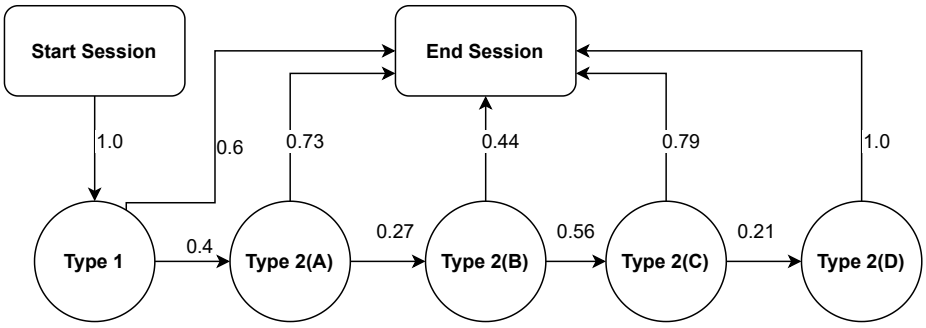


Figure 6.1: Markov Chain for the case study SaaS [4].

While we employed the case study's SaaS framework, we also made slight adjustments to the Markov Chain probabilities to simulate both relatively small and relatively large distributed systems. These LDSSes varied in levels of user interaction, with one type having lower user interaction and the other involving more intensive user engagement. The latter type results in deeper traversal within the middle servers, demanding greater server resources. Additionally, we expanded our testing to include both Poisson workloads and bursty workload patterns.

In each experimental scenario, full simulations were conducted for both of the architectures of two mentioned cloud-based distributed software systems. The initial architecture is characterized by simplicity, comprising 6 distributed servers, each middle server possessing a mere 0.27 probability of advancing to the next component in the system as depicted in Figure 6.2. Comparatively, the second architecture features a greater number of distributed components, each with a higher probability of 0.9 to progress to the subsequent component as shown in Figure 6.3. Consequently, we anticipate longer response times with more variation in the latter architecture.

Henceforth, we refer to the first design as the **"Small System"** and the second one as the **"Large System"**.

6.1.1 Small System

To simulate the Small System, we utilized a modified version of the Markov Chain probabilities outlined in Figure 6.2, which represents the first type of LDSS characterized by lower user interaction. For simplicity and as depicted in Figure 6.2, it is assumed that all servers, excluding the initial web server, have an identical 0.27 probability of advancing the workload onto the next server in the sequence. The Figure 6.2 transition probabilities are a simplified version of those of the actual industry-held production system, as detailed in [4].

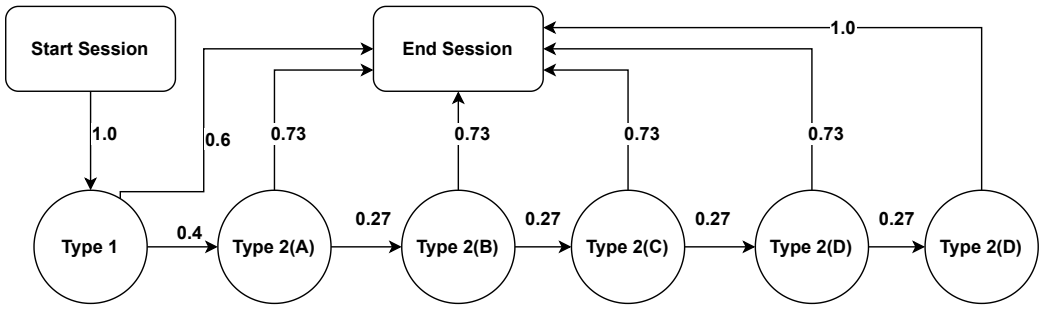


Figure 6.2: Markov Chain for the small system.

Within the conducted cloud-deployed LDSS simulations, run-time performance is assessed via measuring the within-cloud application-layer response time encompassing the entire session duration from when a *http* client request enters the modelled cloud environment to be serviced to when the subsequent LDSS response leaves the modeled cloud environment. The above SaaS LDSS is modelled as a distributed cloud-deployed system composed of a sequence of 6 servers communicating via TCP/IP within the modelled cloud internal network.

6.1.2 Large System

The second system augments the first system, with sessions now being relatively longer due to the addition of more states to the Markov chain. This adjustment is accompanied by higher probabilities of transitioning to the next state, increasing from 0.27 to 0.9, as illustrated in Figure 6.3.

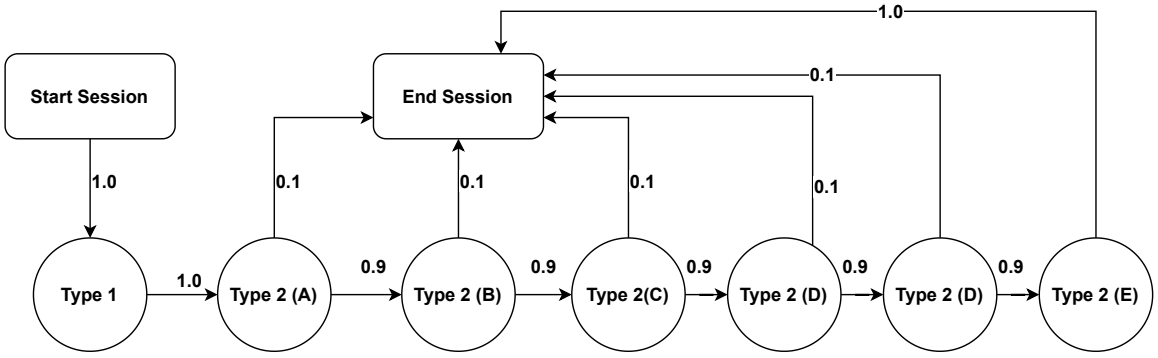


Figure 6.3: Markov Chain for large system.

6.1.3 Workload

For both systems, keeping the system's structure, we simulated and assessed them by subjecting them to two of the most common workload patterns:

- **Poisson Traffic:** Inter-arrival times between requests follow an exponentially distributed pattern with a configurable parameter λ_{inter}.
- **Bursty Traffic:** Bursts of requests arrive at the system simultaneously for a specified time duration in an on/off manner.

Additionally, the think time between phases within each session was modeled as an exponential distribution with a configurable parameter λ_{think}. However, to facilitate comparison among different scenarios as will be explained in the next sections, we applied the same λ_{think} and λ_{inter} value across all of them. This ensured consistency in the simulation setup, allowing for meaningful comparisons of results across various experimental conditions.

While a quite simple LDSS is explored in this work, our constructed simulator is fully capable of modeling far more complex systems. Focusing on a relatively simple SaaS system though is more useful to support the research focus on assessing and more fully understanding cloud-induced performance variability and predictability constraints, given more complex LDSS and workloads can easily mask out the actual sources of observed run-time predictability issues.

6.2 Simulator

This simulator enhances our previous work introduced in [30], enabling simulation of load balancing, resource sharing, and VM/container scheduling. This enhancement allows for the simulation of an arbitrary number of interconnected servers to handle requests efficiently. It enables us to incorporate the richness of modern cloud computing LDSS deployments, encompassing operating system schedulers, application-layer clients, physical servers, VMs within servers, containers within VMs, and more.

The developed simulator is built upon the latest version of the OMNET++ discrete event simulation platform, OMNET++ V5.6.2 [31] (Objective Modular Network Testbed in C++ [274; 275]). To enable full simulation of network layers and protocols, the INET framework is utilized. As for the application layer protocol, TCP is adopted as this is the protocol used in the exemplar system of [4]. The simulator has been implemented such that various attributes, including OS scheduling and queue drops, can be enabled or disabled, or attributes such as server utilization rate can be adjusted within any given simulation run. Hence, the impacts of each attribute on performance predictability can be properly assessed in isolation. A full detailed description of the developed OMNet++ simulator and relevant design choices provided in the following sub-Sections.

6.2.1 Simulator Architecture

Our simulator is implemented on top of OMNET++, a modular, component-based C++ simulation library and framework. While OMNET++ is primarily utilized for network simulation, its core lacks models for network protocols such as IP or HTTP. Therefore, external libraries are required to incorporate these protocols. Among the commonly used ones is INET, which offers a diverse range of models for various network protocols, layers, and technologies, including TCP, IPv6, BGP, and more. We employed the INET framework to fully simulate all the necessary network layers and protocols. TCP serves as the application layer protocol, mirroring the protocol used in the production industry system employed as the simulation's LSS exemplar. Alternate simulation frameworks, such as ns-2 [276] or ns-3 [277], were not used as these tend to abstract out packet-level payload details in favor of improved simulation performance. These packet-level payload details though are critical in the run-time analyses of LDSSes.

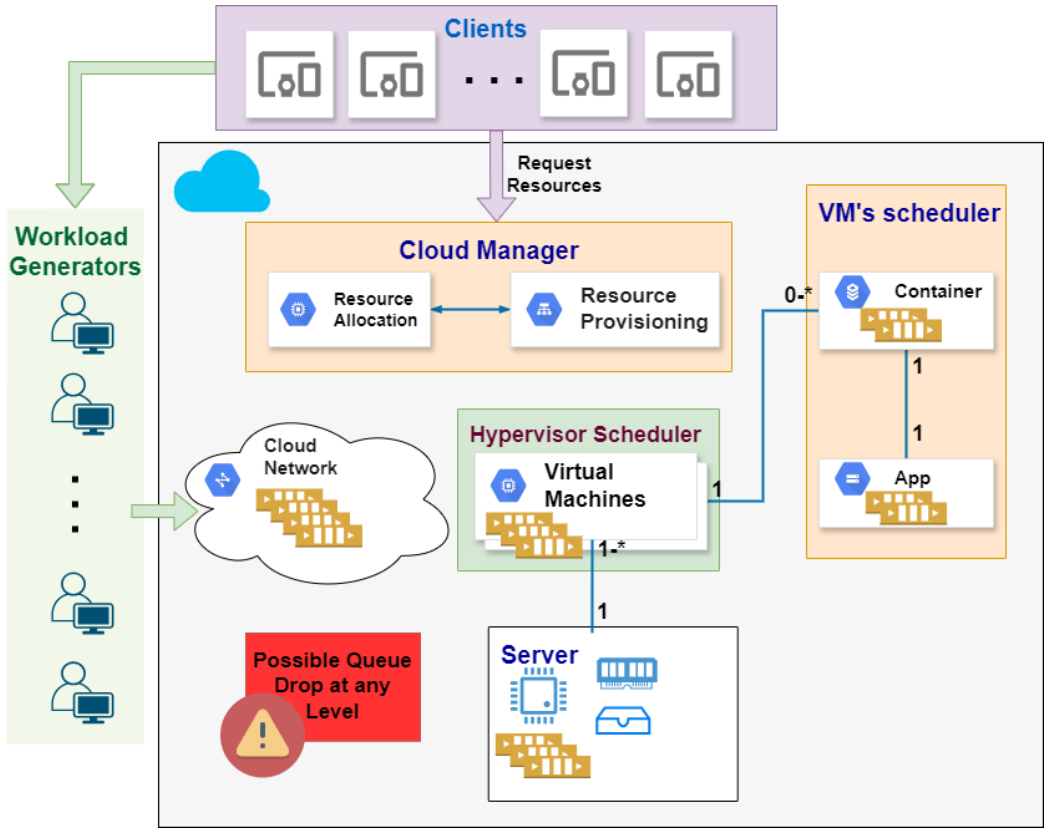


Figure 6.4: Simulator architecture.

Each simulation consists of sets of clients and servers interacting over a modeled cloud network and deployment environment. All the network components including the components needed for communication between different nodes in the cloud or inside each physical node or in application layers, are sourced from INET framework. Each simulation has several workload generators embedded in clients and concurrently sending "*http*" requests to the servers using TCP protocol. Each server in the cloud represents a physical host that hosts several virtual machines. While we haven't implemented detailed hardware components such as CPU, memory, and storage, we have implemented a hypervisor scheduler to emulate resource sharing among hosted virtual machines. Each virtual machine can host multiple containers and applications, competing for available resources. The high-level architecture of the simulation is illustrated in Figure 6.4.

6.2.2 Simulator's Main Components

- **Client Node:**
  The "Client Node" is one of the most important components of the simulation because serves as a tool for modeling the behavior and interactions of the simulated system's clients within the simulated cloud environment. Each "Client Node" is represented as a physical node that emulates either an IPv4 or IPv6 and is equipped with SCTP, TCP, UDP, and an application layer that supports the execution of multiple workload generator applications simultaneously. The "Client Node" aggregates performance measures such as response times and failed requests from all running workload generator applications, providing client-side performance measurements of the simulated system.

  - **Workload Generators:**
    "Client Node" hosts applications which are responsible to generate the workload for a request-response style protocol over TCP. They can be used as a rough model of HTTP or FTP users which are compatible with both IPv4 and IPv6.

    A workload generator application is tasked with orchestrating interactions with server modules in accordance with the prescribed communication protocols and application logic. This entails tasks such as establishing and managing connections, transmitting data packets, processing acknowledgments, and overseeing mechanisms for timeout handling and re-transmission strategies as necessitated by the simulated scenario. INet provides full support for all network protocol details, including TCP/IP handshaking, etc.

    A client application communicates with the server via sessions. Within each session the client initiates a single TCP connection to the server, transmits multiple requests, ensuring it receives the complete reply before sending the next request, and subsequently closing the connection.

  - **Ergodicity Test Client**:
    We have introduced a new client application named the "Ergodicity Test Client", which not only serves as a workload generator but also includes the essential components for implementing reliable protocols. These components incorporate mechanisms for timeout handling and re-transmission strategies. Furthermore, the "Ergodicity Test Client" application imple-

ments the on/off bursty traffic pattern, capable of generating bursts of traffic based on specified burst length parameters. This module collects performance measures such as response time, failed requests, end-to-end delays, as well as sent and received packets. Table 6.1 lists the configuration parameter within the Client Node module.

Table 6.1: Configuration parameters within the Client Node module

Configuration Parameter	**Description**
Local address	The address of the local client node
Local port	The port number of the local client node
Connect address	The address to which the client node connects
Connect port	The port number to which the client node connects
Start time	The time when the client node starts its operation
Stop time	The time when the client node stops its operation
Number of requests per session	The number of requests per session, which depends on workload structure
Request length	The length of each request, which can be stochastic
Reply length	The length of each reply, which can be stochastic
Think time distribution	The distribution that determines the time between two consecutive requests
Idle interval distribution	The distribution that determines the idle time between two client sessions
Reconnect interval	The interval for retrying to connect to the server in case of losing the connection

- **Internet Cloud:**
  In the most recent version of INET framework, there is a new module called the "InternetCloud" which implements IPV4 routers in cloud communications that can delay or drop packets while retaining their order based on which interface card the packet arrived on and on which interface it is leaving the cloud. Our simulator uses this component for handling communications between nodes in the simulator. This module lets us configure the delay, drop and data-rate parameters via XML file.

- **Servers:**
  Each cloud server in the simulation represents a physical server equipped with an IPv4/IPv6 host, along with support for SCTP, TCP, and UDP communication

protocols. Additionally, each server is equipped with an application layer that enables the concurrent execution of multiple virtual machines (VMs) within its infrastructure. Furthermore, a hypervisor scheduler is integrated into each physical server, tasked with ensuring fair scheduling among the virtual machines it hosts.

The number of VMs hosted on each server is configurable, allowing users to adjust this parameter according to their simulation requirements. This configurability enables the customization of the simulation environment to explore various deployment scenarios and evaluate the performance of cloud-based systems under different VM configurations.

– **Hypervisor Scheduler:**
Whenever a new virtual machine get created on the physical host, this module assigns a queue for it which keeps all the requests in order. The queues get on and off depending on the scheduler configurations. By default, the scheduler is a fair scheduler with a configurable share amount. A turned-on queue means that the according virtual machine is up and running, using the system resources to respond to the requests. There is a small configurable service time associated with processing each queue's request to mimic the scheduling overhead and physical host resource usage.
The configurable parameters for each modeled queue within an OS Scheduler layer are given in Table 6.2.

Table 6.2: Configurable parameters for each modeled queue within an OS Scheduler layer

Configuration Parameter	**Description**
Capacity	If set, it may result in queue drops.
Fetching algorithm	Options include priority, random, round-robin, and longest queue.
Service time	Mimics the overhead associated with processing.
Status	Determines if there are other unknown loads on the server.
Sleep time	Effective when there are other VMs on the server.
Scheduler Share	Effective when there are other VMs on the server.

– **Virtual Machines:**
Each virtual machine can accommodate several application/containers. Virtual machine module has an operating system scheduler which is responsible to do a fair scheduling between applications/containers. This scheduler is similar to the hypervisor scheduler with configurable share amount and service time. This approach is chosen as it models real-world commercial cloud deployments in which clients deploy their containers into VMs they have leased on within a given cloud platform (or facility).

– **Applications or Containers:**
Containers execute server applications to model TCP-based request-reply style protocols or applications. This module is designed to handle any number of incoming TCP connections and expects messages specifying the desired size of the reply (in bytes). The server application adjusts the length of the received message accordingly and sends back the modified message object. Additionally, the reply can be delayed by a constant time. The per-container configurable parameters are: local address, local port, and reply delay.

6.3 Simulation-based Validation of DST-derived Insights

This Section details specific simulation scenarios and results that have been conducted to investigate the insights derived from the DST theory regarding LDSS performance predictability. The focus of these simulations is on assessing the underlying cdfs describing the simulated LDSS's performance behavior and how they vary across different independent simulation runs. More formally, the analyses assess the BET-compliance of the simulations. More specifically, the interest lies in the richness of the family of cdfs producible under any given simulation scenario and set of experiment runs (or ensemble). A wider variety of producible cdfs indicates larger losses in run-time performance predictability. For these assessments, the focus is restricted to evaluating the application layer response times of the simulated LDSS.

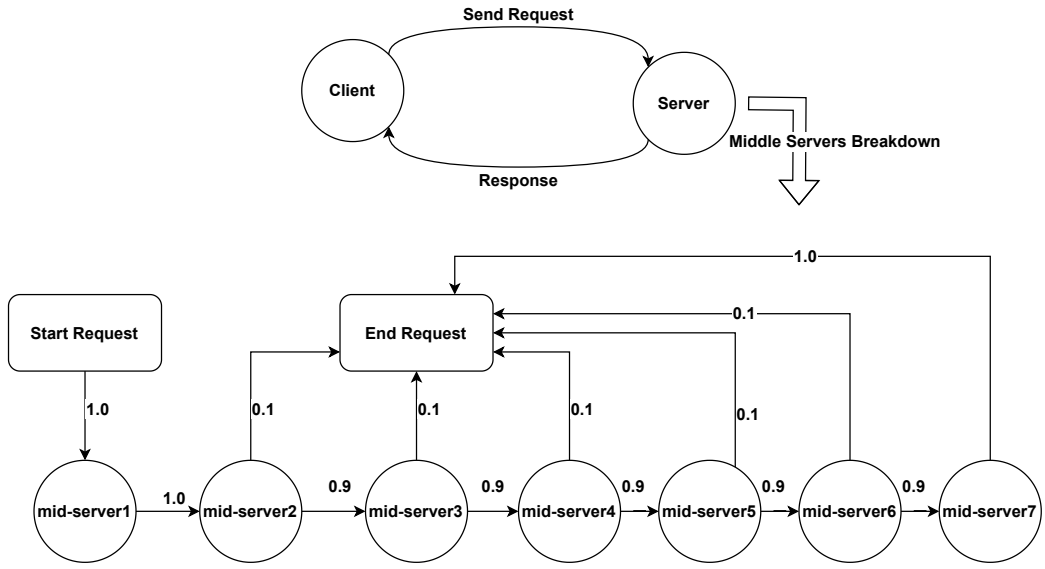


Figure 6.5: Distributed system architecture with seven nodes which is equivalent to large system's Markov Chain.

6.3.1 Simulation Scenarios

The described user-system interaction in both "Small' and "Large" systems correspond to distributed systems comprising six and seven layers or middle servers communicating over the network, as illustrated in Figure 6.5 for the Large System. This architecture closely resembles the popular micro-architectures observed in modern systems. Throughout our experiments, we consistently utilized six and seven layers across all scenarios to ensure comparability. However, it's worth noting that in our simulator, the number of layers and their associated probabilities are entirely configurable. This flexibility allows for the simulation of both smaller and larger distributed systems, accommodating a diverse range of architectural configurations.

Comparing to modern LDSS, both "Small" and "Large" systems are considered non-complex systems which can be modelled using Markov chains and therefore their performance behavior expected to be predictable and well-behaved. The main objective of simulating such systems is to demonstrate that even in supposedly simple well-behaved systems, if we break the BET-compliance rules then we lose predictability. To assess run-time performance, we measure the within-cloud application-layer response time simulating progressively complex scenarios as shown in Table 6.3

By providing configurable workloads and distributed system architectures, and by employing six key scenarios spanning from simple to complex, we can facilitate

Table 6.3: Description of conducted Monte Carlo experiment scenarios.

Experiment	Description	Pages
Scenario 1	System with uncontested resources and infinite queues.	179 to 194
Scenario 2	System with uncontested resources and finite queues.	194 to 207
Scenario 3	System with shared servers servicing fixed number of VMs with infinite queues.	207 to 215
Scenario 4	System with shared servers servicing fixed number of VMs with finite queues.	216 to 224
Scenario 5	System with shared servers servicing dynamic varying load with infinite queues.	224 to 232
Scenario 6	System with shared servers servicing dynamic varying load with finite queues.	232 to 242

effective comparison between the performance of a straightforward BET-compliant software system and that of more realistic, BET non-compliant systems. Moreover, the flexibility to toggle various factors such as reliable protocol, queue drops, or resource competition in the simulation allows us to systematically investigate and understand how these different elements influence system performance predictability in cloud environments.

In each scenario, we imposed both distinct incoming workloads on the system. The first involved a regular workload characterized by a Poisson distribution (P). The second workload was of a bursty (B) nature, with intermittent surges of requests flooding the system at a rate 10 times higher than the normal traffic. The volume and duration of these bursts are adjustable within the simulator. In this Chapter, the term **"Request"** refers to the complete session, while **"Internal Request"** denotes the back-and-forth requests between middle servers.

Furthermore, within each scenario, we conduct simulations using both a reliable protocol (TCP) and a non-reliable protocol (UDP) to evaluate the impact of reliable protocols on performance predictability. In reliable protocol, if we don't get response from any components of the system in an expected time window e.g. 32s, then we re-send the request and this process will be repeated for maximum of 8 times. In non-reliable protocol, we wait for a specified time period e.g. 128s, and if we don't get any response, we consider the request as a failed request and we proceed with the next request. It should be noted that in the response time figures for all scenarios, only successful requests considered to have a response time.

The resulting six distinct scenarios with their variations as summarized in Table 6.51 serve as a sufficient test suite for the developed simulator while also enabling the derived DST insights to be properly assessed, i.e., whereby extraneous factors are abstracted out of the LDSS application-layer simulations so as to focus on the core issues leading to performance predictability losses. In Table 6.51, "Sx-Ey" indicates the scenario number and the experiment number within that scenario. For example, S3-E6 represents experiment 6 within scenario 3. System can be either "Small" (S) or "Large" (L) and the scheduling refers to either a common hypervisor OS fair scheduling (F) or a quiescent scheduling without resource competition (Q). Servers can either have a static number of VMs assigned to them (S) or they can have a dynamically varying load (D) assigned to them. Queues can be either very large to simulate infinite queues without queue drops (I) or finite queues (F) with queue drop possibilities. This approach allows experiments to be easily differentiated by their coded labeling. For example, [S:P:F:F:TCP:S], denotes a simulation of a small system servicing Poisson traffic on a shared server with static load and fair scheduling having finite queues using TCP protocol.

In each simulation run, for the sake of consistency between different scenarios, the same number of workload generators were used for the same type of workloads. For normal traffic, 10 workload generators concurrently dispatch "*http*" requests to the system. For bursty traffic, 100 workload generators are used in the same manner. Each session mirrors one user's interaction with the system, aligning with the workload model illustrated in Figure 6.2 and Figure 6.3. Session time, measured as the total response time, is recorded for each interaction. The shared simulation parameters, common across all simulations, are summarized in Table 6.5.

Each simulation run's overall system response time distribution is computed as the aggregate of all the response time distributions computed for each simulated workload generator, as illustrated in an example with 14 workload generators in Figure 6.6. For ease of comparison between runs, these are then combined into the resulting overall system response time distribution, as shown in Figure 6.7. However, this process results in the same overall system response time distribution that would occur if the raw response time data were first aggregated and then used to produce the overall cross-ensemble response time distribution. Figures 6.6 and 6.7 highlight the statistical simplicity of the incoming workloads and, hence, that workload complexities are not the source of the observed run-time behavioral complexities.

Table 6.4: Comparison of simulation scenarios and experiments with various configurations and workloads.

Set of Conducted Simulations (Ordered by increasing complexity)						
Ensemble Code	**System (S/L)**	**Workload (P/B)**	**Scheduling (F/Q)**	**Queue (F/I)**	**Protocol (TCP/UDP)**	**Load (S/D)**
S1-E1 (100)	S	P	Q	I	TCP	S
S1-E2 (100)	S	P	Q	I	UDP	S
S1-E3 (100)	L	P	Q	I	TCP	S
S1-E4 (100)	L	P	Q	I	UDP	S
S1-E5 (100)	S	B	Q	I	TCP	S
S1-E6 (100)	S	B	Q	I	UDP	S
S1-E7 (100)	L	B	Q	I	TCP	S
S1-E8 (100)	L	B	Q	I	UDP	S
S2-E1 (100)	S	P	Q	F	TCP	S
S2-E2 (100)	S	P	Q	F	UDP	S
S2-E3 (100)	L	P	Q	F	TCP	S
S2-E4 (100)	L	P	Q	F	UDP	S
S2-E5 (100)	S	B	Q	F	TCP	S
S2-E6 (100)	S	B	Q	F	UDP	S
S2-E7 (100)	L	B	Q	F	TCP	S
S2-E8 (100)	L	B	Q	F	UDP	S
S3-E1 (100)	S	P	F	I	TCP	S
S3-E2 (100)	S	P	F	I	UDP	S
S3-E3 (100)	L	P	F	I	TCP	S
S3-E4 (100)	L	P	F	I	UDP	S
S4-E1 (100)	S	P	F	F	TCP	S
S4-E2 (100)	S	P	F	F	UDP	S
S4-E3 (100)	L	P	F	F	TCP	S
S4-E4 (100)	L	P	F	F	UDP	S
S5-E1 (100)	S	P	F	I	TCP	D
S5-E2 (100)	S	P	F	I	UDP	D
S5-E3 (100)	L	P	F	I	TCP	D
S5-E4 (100)	L	P	F	I	UDP	D
S6-E1 (100)	S	P	F	F	TCP	D
S6-E2 (100)	S	P	F	F	UDP	D
S6-E3 (100)	L	P	F	F	TCP	D
S6-E4 (100)	L	P	F	F	UDP	D

Table 6.5: General simulation parameters.

Simulation Parameter	Value
Performance Measure	Application Layer Response Time
No. of Independent Runs	100
Idle time distribution	$p(x) = \lambda e^{-\lambda x}; \lambda = 1/100$ ms
Think time distribution	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/10$ ms
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ OR $\lambda = 1/400$ ms
Simulation time	36×10^5 and 36×10^6 ms (1h and 10h)

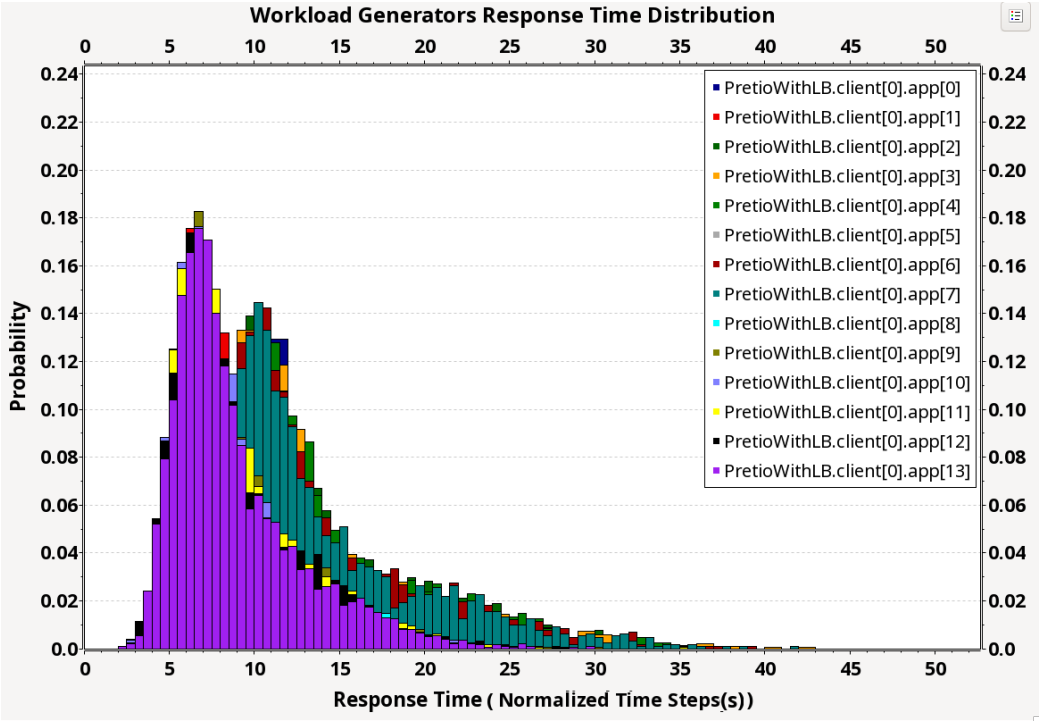


Figure 6.6: Per-workload generator produced workload response time.

It should be noted that in Figure 6.6, the term "Client" refers to an individual software system acting as a cloud customer, while "app" represents workload generators containing TCP client applications that generate "*http*" requests. These applications interact with the web server based on our Case Study's Markov Chain model.

As per Table 6.51,all simulations were conducted and analyzed over independent sets of 100 Monte Carlo runs. This resulted in a total of 3,200 conducted experiments across the 32 experiment scenarios.

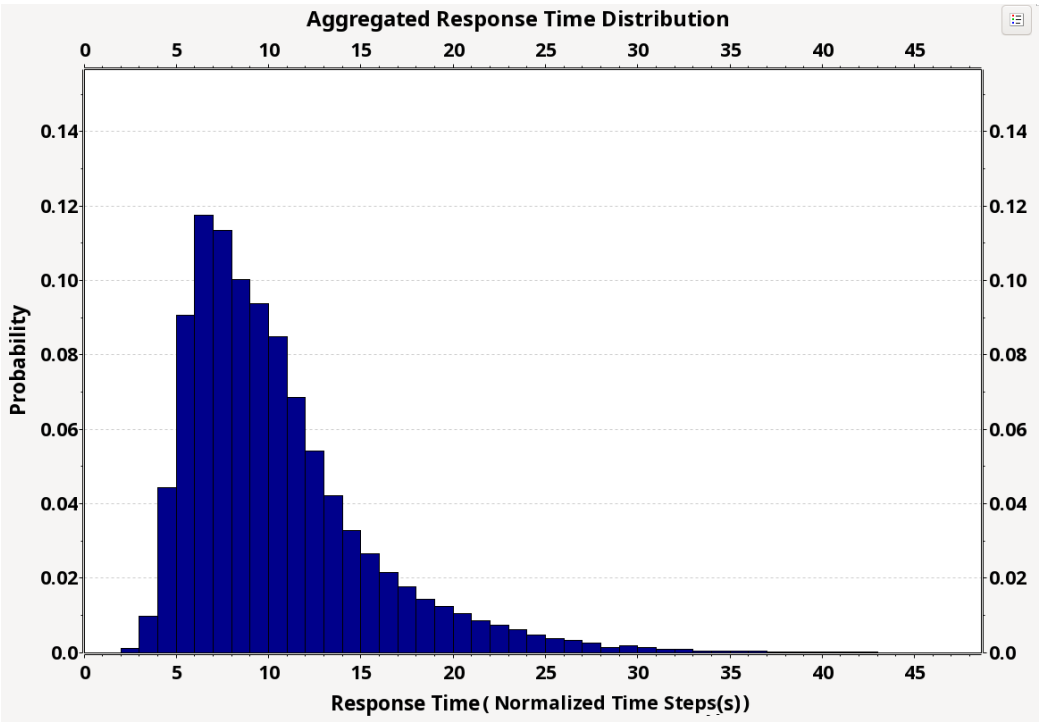


Figure 6.7: Aggregated system's response time for one simulation run.

6.3.2 Scenario 1: System with uncontested resources and infinite queues.

In the baseline simulation scenario, our exemplary distributed systems were deployed in a cloud environment with dedicated servers and sufficiently large queues to prevent queue drops. Note that, queue drops may still arise within the INET simulation network layers and protocols, but sufficient resourcing levels are simulated such that application-layer queue drop events do not arise. Within this scenario, we conducted eight experiments involving both small and large distributed systems exposed to both normal (Poisson) and bursty incoming traffic, employing both reliable and non-reliable protocols. In each experiment, we initially presented the results from the reliable protocol, as it is more common in practice. Then, for the sake of comparison, we presented the results after removing the reliable protocol at the application layer. Each experiment was repeated 100 times to enable the examination of the run-time environment's impact on the system's performance behavior. In the first few experiments, we conducted simulations for duration of 1 hour and 10 hours to investigate whether the (weak) law of large numbers [33] applies as we collect more samples from independent identical runs of the system.

6.3.2.1 S1-E1: Small distributed software system exposed to normal traffic and employing reliable protocol [S:P:Q:I:TCP:S]

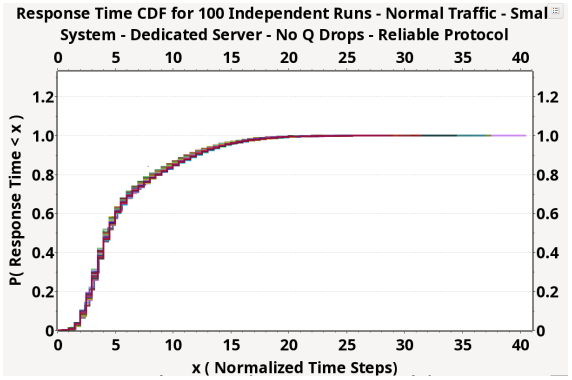
Figure 6.8 presents the application-layer response time pdfs and cdfs derived from 100 independent simulation runs of the exemplary distributed software system. All runs within this experiment exposed to identical statistical Poisson workload, with quiescent scheduling in which there is no resource competition. Table 6.6 provides a summary of the simulation parameters for this experiment. Figure 6.8a and 6.8b display the response time cdfs of the system over one hour and 10 hours of simulation time, respectively. Figure 6.8c and 6.8d illustrate the associated pdfs. These figures reveal that in this quiescent state, no statistically significant variations exist across the 100 independent experimental runs. The generated response time cdfs exhibit nearly identical patterns, with any distinctions attributed to expected statistical variations inherent in ergodic stochastic processes. More particularly, these results highlight that within quiescent run-time environments, the distributed software system produces highly consistent and, therefore, predictable performance results. Furthermore, due to the system's BET-compliant nature, the law of large numbers results in smoother distributions as more data points are considered.

Table 6.6: S1-E1 specific simulation parameters.

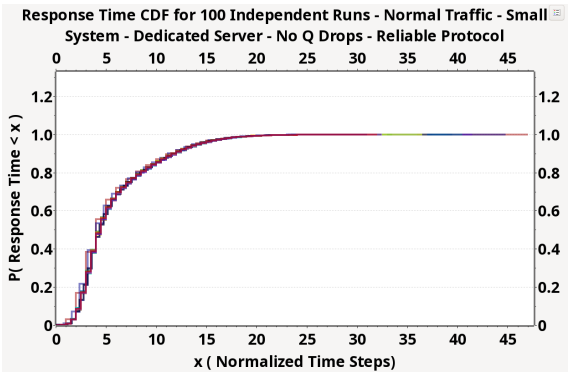
Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

6.3.2.2 S1-E2: Small distributed software system exposed to normal traffic and employing non-reliable protocol [S:P:Q:I:UDP:S]

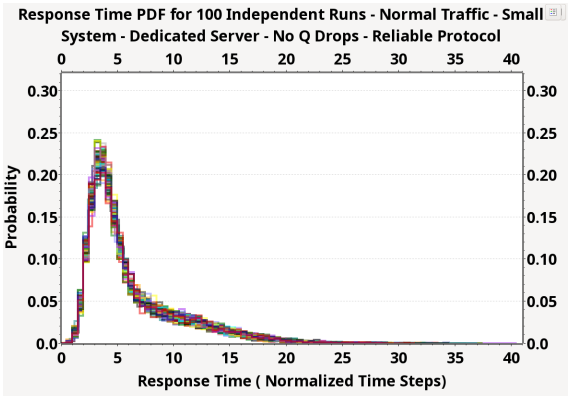
Experiment S1-E2 mirrors Experiment S1-E1, but with the application of a non-reliable protocol. The results confirm and validate the DST-derived insights for quiescent software systems deployments where resource competition and queue drop events do not occur. Consequently, the reliable protocol is lightly activated and therefore, the performance results of the reliable and non-reliable protocols are almost identical. Table 6.7 offers a summary of the simulation parameters for Experiment 2.



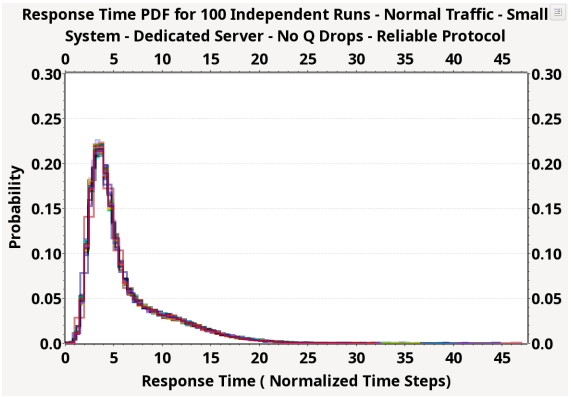
(a) S1-E1 one hour of simulation.



(b) S1-E1 ten hours of simulation.



(c) S1-E1 one hour of simulation.



(d) S1-E1 ten hours of simulation.

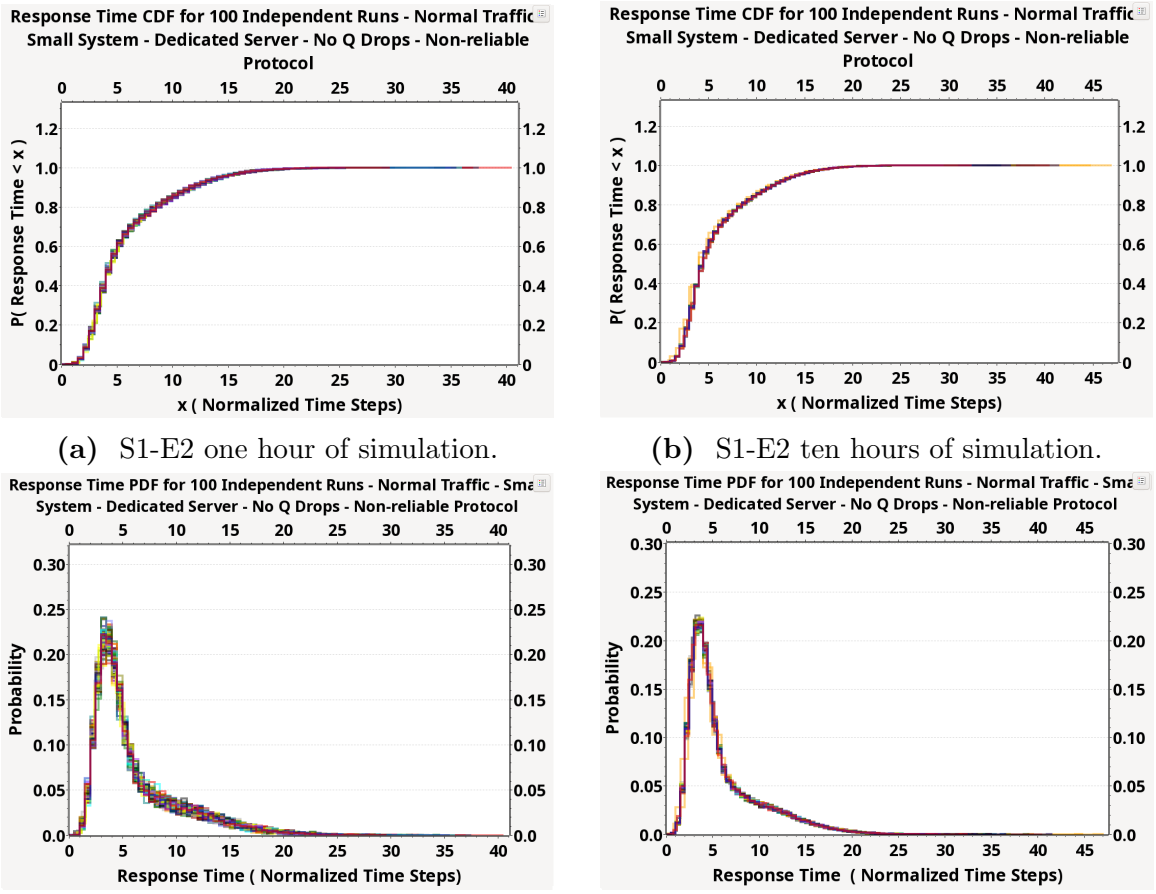
Figure 6.8: S1-E1 Application-layer response time distributions. [S:P:Q:I:TCP:S]

Table 6.7: S1-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

6.3.2.3 S1-E3: Large distributed software system exposed to normal traffic and employing reliable protocol [L:P:Q:I:TCP:S]

Experiment S1-E3 is akin to Experiment S1-E1, with the difference that the system under test is the large distributed system wherein most sessions consist of 5 to 6 requests. All other simulation conditions, such as the incoming workload, processing time at servers, etc., remain identical to Experiment 1. As in Experiment 1, all runs



(a) S1-E2 one hour of simulation.

(b) S1-E2 ten hours of simulation.

(c) S1-E2 one hour of simulation.

(d) S1-E2 ten hours of simulation.

Figure 6.9: S1-E2 Application-layer response time distributions. [S:P:Q:I:UDP:S]

are exposed to identical statistical workloads, with no resource competition or queue drop events. Table 6.8 provides a summary of the simulation parameters for this experiment.

Table 6.8: S1-E3 specific simulation parameters.

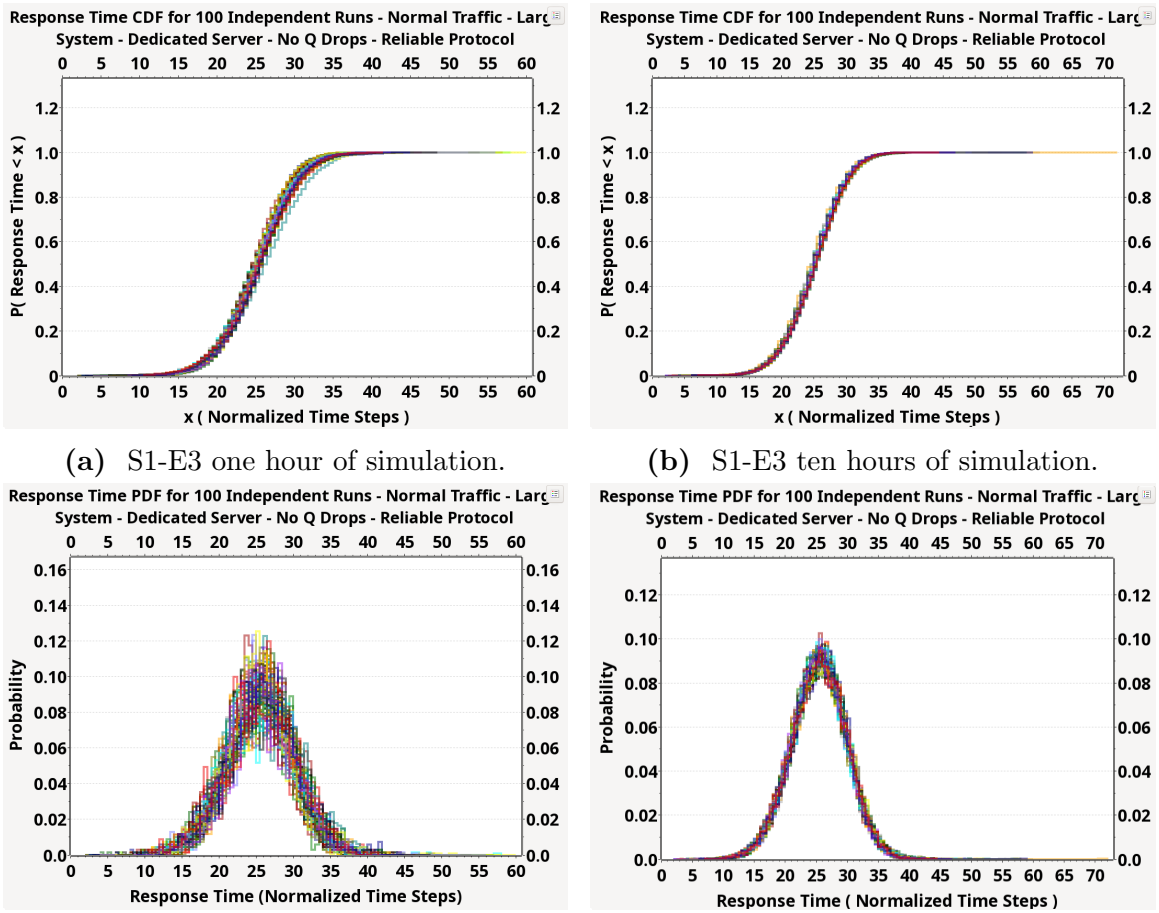
Simulation Parameter	**Value**
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

Figure 6.10c and 6.10d depict the response time pdfs of the system over one hour and 10 hours of simulation time, respectively. Correspondingly, Figure 6.10a and 6.10b illustrate the associated cdfs. These figures unveil that in this quiescent state, irrespective of the system's size, no statistically significant variations are evident across the 100 independent experimental runs. Additionally, due to the similarity in the number of requests per session, the pdfs exhibit a more symmetric shape compared to those of our smaller system, where the probability of sessions with fewer requests is higher. The generated response time cdfs demonstrate nearly identical patterns, with any discernible differences attributed to expected statistical variations inherent in ergodic stochastic processes. These results highlight that within quiescent run-time environments, the large distributed software system produces highly consistent and, therefore, predictable, performance results. Furthermore, similar to the small system, due to the system's BET-compliant nature, the law of large numbers results in a smoother distribution as more data points are considered.

6.3.2.4 S1-E4: Large distributed software system exposed to normal traffic and employing non-reliable protocol [L:P:Q:I:UDP:S]

Experiment S1-E4 replicates the structure of the previous one, focusing on a large distributed software system operating under normal traffic conditions but with the omission of a reliable protocol at the application layer. The main objective here is to explore the variation in the performance behavior of the system resulting from the removal of the reliable protocol from the application layer. Given the BET-compliant nature of the system and insights derived from DST in Chapter 5, the reliable protocol in the previous experiment is not frequently triggered in a manner that could impacts the system's BET-compliance or its performance behavior. Consequently, the expectation is that the removal of the reliable protocol may not affect the system's performance behavior. Table 6.9 summarizes the parameters used for experiment 6.

The results obtained from the simulation, as depicted in Figures 6.11a to 6.11d, validate the expectations derived from the DST-driven insights outlined in Chapter 5.



(a) S1-E3 one hour of simulation. **(b)** S1-E3 ten hours of simulation.

(c) S1-E3 one hour of simulation. **(d)** S1-E3 ten hours of simulation.

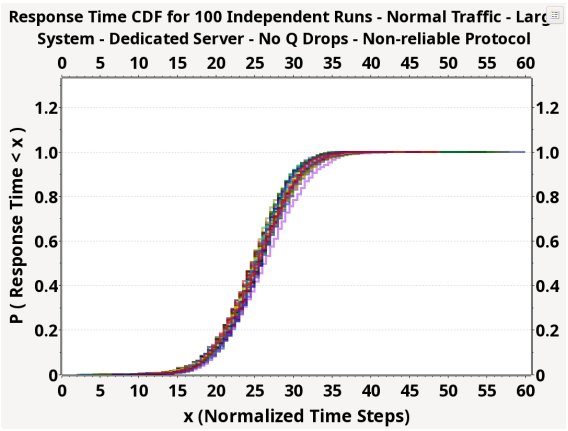
Figure 6.10: S1-E3 Application-layer response time distributions. [L:P:Q:I:TCP:S]

Table 6.9: S1-E6 specific simulation parameters.

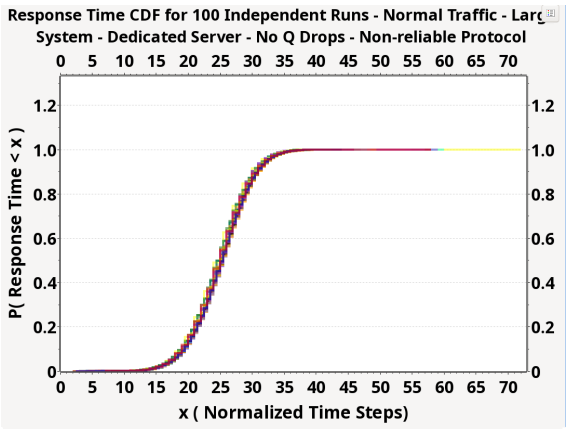
Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

6.3.2.5 S1-E5: Small distributed software system exposed to bursty traffic and employing reliable protocol [S:B:Q:I:TCP:S]

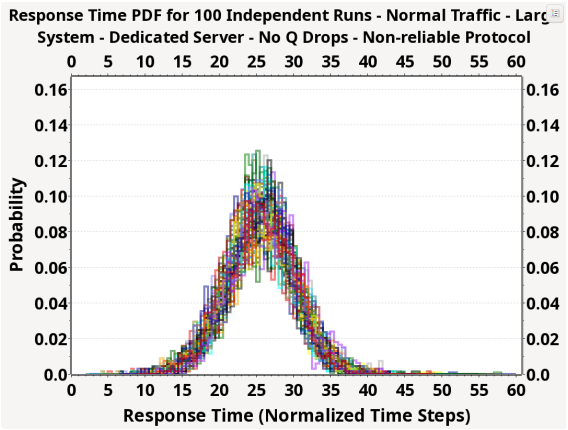
Experiment S1-E5 replicates the conditions of Experiment S1-E1 but introduces an incoming bursty workload, consisting of bursts lasting 300 seconds with a traffic in-



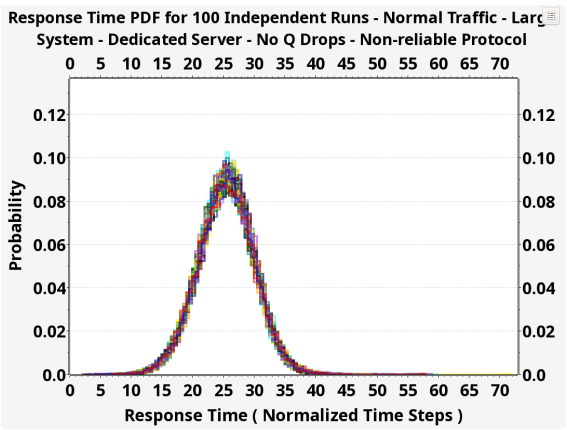
(a) S1-E4 one hour of simulation.



(b) S1-E4 ten hours of simulation.



(c) S1-E4 one hour of simulation.



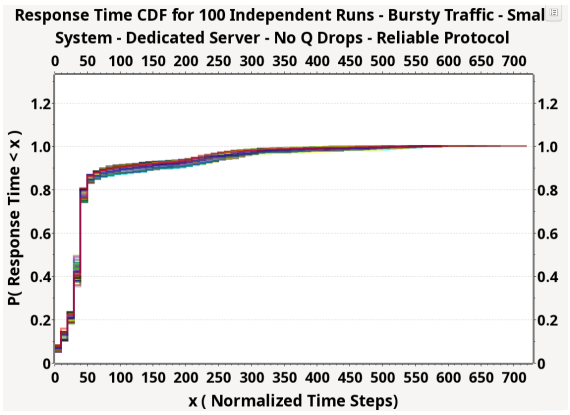
(d) S1-E4 ten hours of simulation.

Figure 6.11: S1-E4 Application-layer response time distributions. [L:P:Q:I:UDP:S]

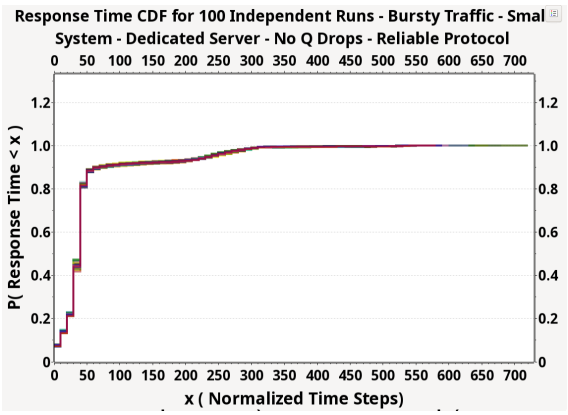
tensity at least 10 times higher. It should be noted that the burst length and intensity is configurable. Upon the arrival of a burst, the queues begin to accumulate, allowing the initial requests to be processed similar to a normal traffic. However, subsequent requests encounter prolonged waits within the queues of each server. As the burst concludes, the queues gradually diminish until the occurrence of the next burst. With the reliable protocol, we resend requests whenever we don't receive a reply from any server within 32 seconds. This occurrence is frequent as the burst progresses. This behavior overloads the server and disrupts the natural order of requests, resulting in long delays in response times as shown in Figures 6.12a to 6.12d but the occurrences are not large enough to make the performance behavior unpredictable within the experiment. Table 6.10 provides a summary of the simulation parameters for Experiment 3.

Table 6.10: S1-E3 specific simulation parameters.

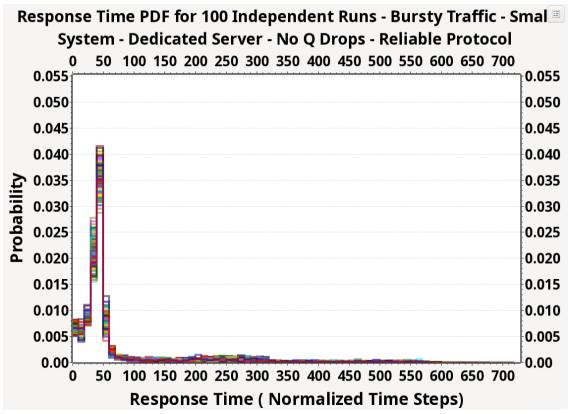
Simulation Parameter	Value
Workload	Bursty
Burst Length	300s
Burst Intensity	10x
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms



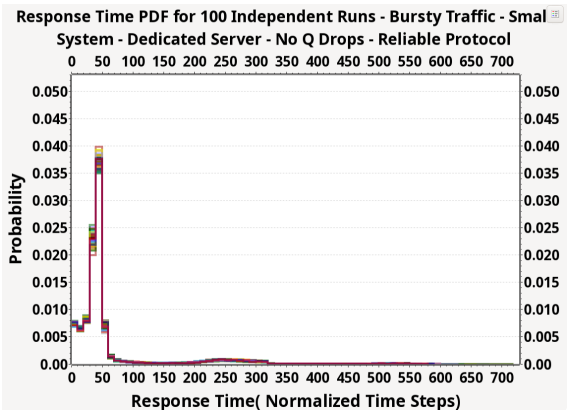
(a) S1-E5 one hour of simulation.



(b) S1-E5 ten hours of simulation.



(c) S1-E5 one hour of simulation.



(d) S1-E5 ten hours of simulation.

Figure 6.12: S1-E5 Application-layer response time distributions. [S:B:Q:I:TCP:S]

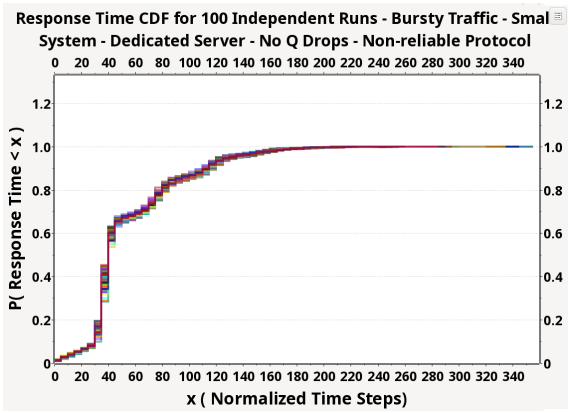
6.3.2.6 S1-E6: Small distributed software system exposed to bursty traffic and employing non-reliable protocol [S:B:Q:I:UDP:S]

Experiment S1-E6 replicates the conditions of Experiment S1-E3 but applying a non-reliable protocol. Using a reliable protocol in Experiment 3 entails re-sending the request if no response is received within a specified time frame. This, in turn, leads to heightened queue congestion, consequently causing longer response times as shown in Figures 6.12a to 6.12d, compared to the non-reliable protocol shown in Figures 6.13a to 6.13d. Without the reliable protocol, we simply wait for the response to arrive. For each middle server, for the sake of simplicity in comparison, we wait four times the timeout time used in the reliable protocol, which in this case equals 128 seconds, before considering it a failed request. Consequently, we observe bumps in the pdf, indicating system's 5 phases in each requests require an additional processing step. These jumps are observable because the requests maintain their arrival order, unlike in the case of the reliable protocol. Table 6.11 provides a summary of the simulation parameters for Experiment 4.

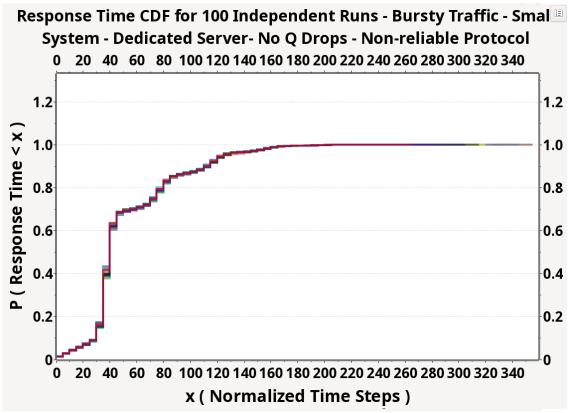
Table 6.11: S1-E4 specific simulation parameters.

Simulation Parameter	Value
Workload	Bursty
Burst Length	300s
Burst Intensity	10x
System Architecture	Small system in Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

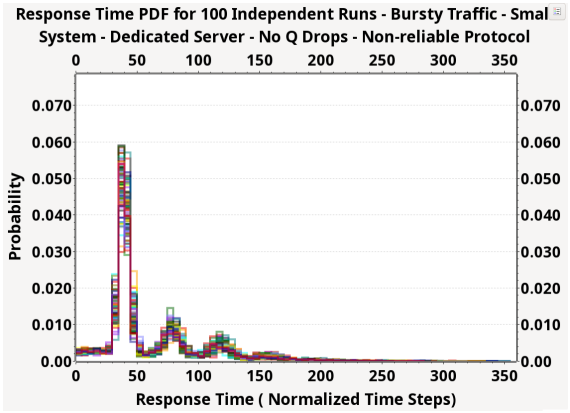
In the comparison of reliable and non-reliable protocols for bursty traffic, it is observed that the system applying non-reliable protocol has shorter response times at the 95^{th} percentile around 125 normalized time steps compared to the reliable protocol system's 250 steps. However, at the 90^{th} percentile, the difference in response times is less which presents the consequences of request resending that can strain the server, resulting in longer response times and increased failure rates. Consequently, the system with the reliable protocol experiences a failure rate of approximately 14%. In contrast, the non-reliable protocol system logs no failures, suggesting that a timeout of 128 seconds is sufficient for waiting for each server to handle the incoming request



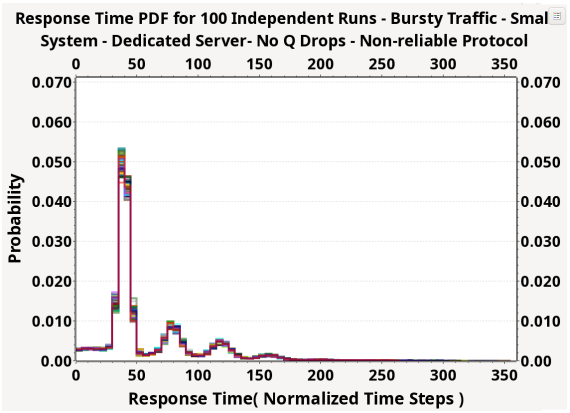
(a) S1-E6 one hour of simulation.



(b) S1-E6 ten hours of simulation.



(c) S1-E6 one hour of simulation.



(d) S1-E6 ten hours of simulation.

Figure 6.13: S1-E6 Application-layer response time distributions. [S:B:Q:I:UDP:S]

load. It is worth noting that shorter timeouts might also lead to failures in systems using non-reliable protocols.

6.3.2.7 S1-E7: Large distributed software system exposed to bursty traffic and employing reliable protocol [L:B:Q:I:TCP:S]

Experiment S1-E7 mirrors the conditions of Experiment S1-E5 but on a larger distributed system. The incoming traffic consists of on-and-off bursts with a traffic intensity at least 10 times higher. It's important to note that both the burst length and intensity are configurable parameters in our simulator. When there is no queue drop, the reliable protocol merely exacerbates queue congestion, resulting in generally longer response times. The burst length indicates the duration of bursts in the incoming workload. In three separate tests within this experiment, burst lengths of 150 seconds, 300 seconds, and 500 seconds were explored, with equal durations for

both the "on" and "off" periods. Table 6.12 provides a summary of the simulation parameters for Experiment S1-E7.

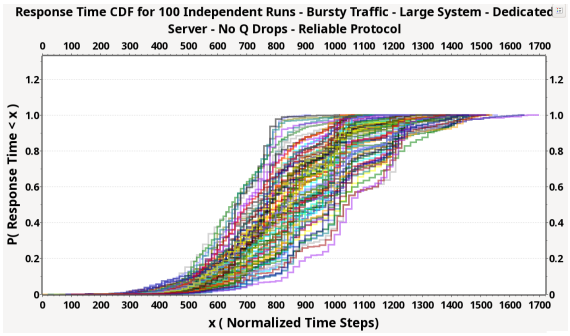
Table 6.12: S1-E7 specific simulation parameters.

Simulation Parameter	Value
Workload	Bursty
Burst Length	150s, 300s, 500s
Burst Intensity	10x
System Architecture	Large system in Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

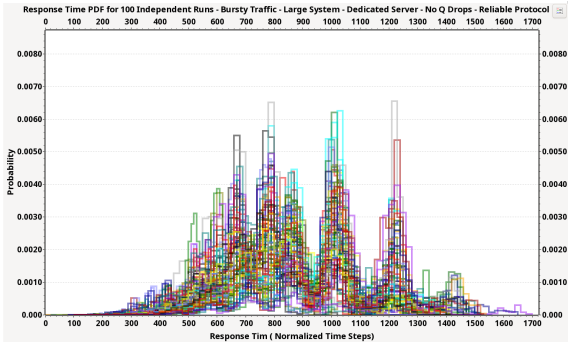
The results clearly indicate that bursty traffic in larger systems triggers the reliable protocol to a significant extent, disrupting the system's BET-compliant state and leading to unpredictable performance behavior regardless of the burst length. In reality, the burst length can vary over time, exacerbating the situation by causing transitions between these cdfs. The following figures present the results of simulations for bursts of lengths 150, 300, and 500 seconds, respectively. We can compare these cdfs at their 95^{th} percentile. From the figures, we can see that the worst-case scenario, with the longest response time and the widest cdfs, occurs for bursts with a length of 150s because the off time is not long enough to conclude the burst. On the other hand, from all figures, we observe that the more the reliable protocol triggers, the higher the congestion, leading to more and more triggers of the reliable protocol, which causes the systems' response time behavior to diverge further from each other.

6.3.2.8 S1-E8: Large distributed software system exposed to bursty traffic and employing non-reliable protocol [L:B:Q:I:UDP:S]

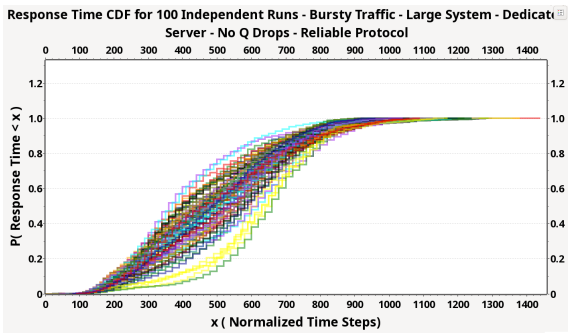
Experiment S1-E8 replicates the conditions of Experiment S1-E7 but applies a non-reliable protocol. Using a reliable protocol in Experiment 7 entails resending the request if no response is received within a specified time frame. This, in turn, leads to heightened queue congestion, consequently causing longer response times as shown in Figures 6.14a to 6.14f, compared to the non-reliable protocol shown in Figures 6.15a to 6.15f.



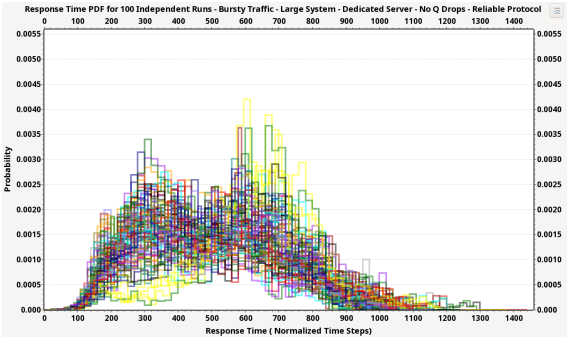
(a) S1-E7 - cdfs for burst length of 150s.



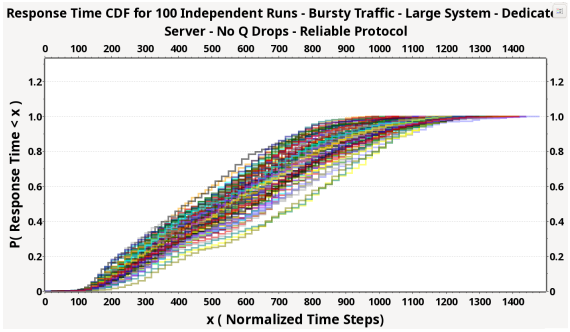
(b) S1-E7 -pdfs for burst length of 150s.



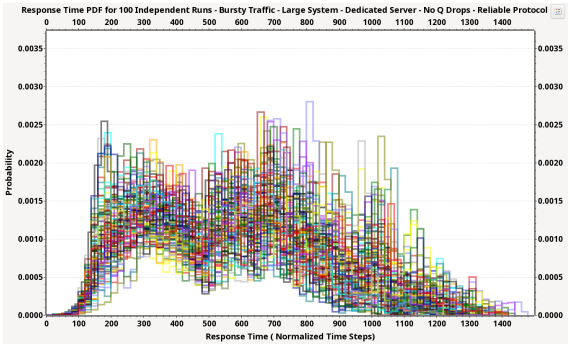
(c) S1-E7 - cdfs for burst length of 300s.



(d) S1-E7 - pdfs for burst length of 300s.



(e) S1-E7 - cdfs for burst length of 500s.



(f) S1-E7 - pdfs for burst length of 500s.

Figure 6.14: S1-E7 Application-layer response time distributions. [L:B:Q:I:TCP:S]

Table 6.13 provides a summary of the simulation parameters for Experiment S1-E8. It should be noted that for each middle server response, we wait 128s before we consider it a failed internal request.

From the simulation results illustrated in Figures 6.15a to 6.15f, it is evident that eliminating the reliable protocol from the application layer irrespective of the burst length, restores the system to a BET-compliant state. This restoration brings back performance predictability and validates the DST-driven insights from Chapter 5.

Table 6.13: S1-E8 specific simulation parameters.

Simulation Parameter	Value
Workload	Bursty
Burst Length	150s, 300s, 500s
Burst Intensity	10x
System Architecture	Large system in Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

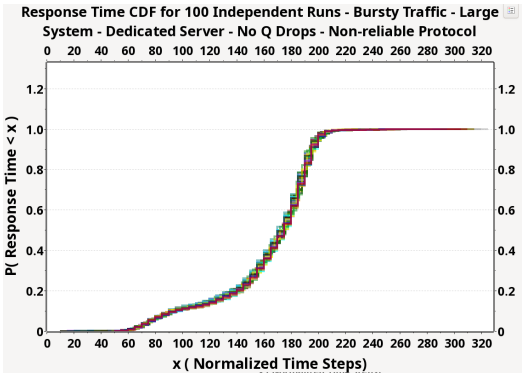
6.3.2.9 Scenario 1 Results Summary

We summarized the configurations and results of scenario 1 experiments in Table 7.2. The burst's length is 300s for the bursty traffic results in S1-E5 to S1-E8.

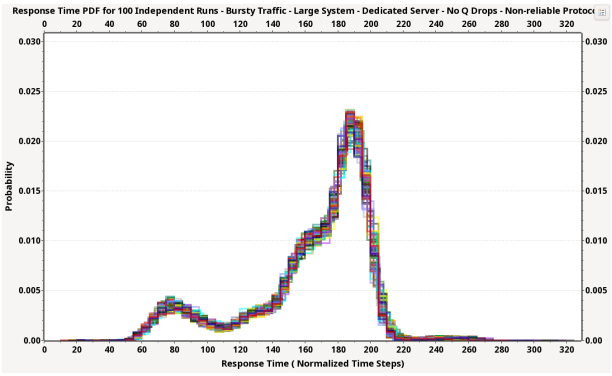
Table 6.14: Scenario 1 experiments results comparison.

Ensemble Code	Experiment Code	Failure Rate [Min. , Max.]	Response Time (NTS) Percentiles [Min. , Max.]		
			90^{th}	95^{th}	99.99^{th}
S1-E1(100)	S:P:Q:I:TCP:S	[0.0% , 0.0%]	[11.2 , 12.0]	[14.0 , 14.5]	[28.5 , 43.2]
S1-E2(100)	S:P:Q:I:UDP:S	[0.0% , 0.0%]	[11.2 , 12.0]	[13.6 , 14.0]	[27.0 , 45.0]
S1-E3(100)	L:P:Q:I:TCP:S	[0.0% , 0.0%]	[30.0 , 31.0]	[31.5 , 32.5]	[39.0 , 61.5]
S1-E4(100)	L:P:Q:I:UDP:S	[0.0% , 0.0%]	[30.0 , 31.0]	[31.5 , 32.5]	[39.0 , 63.0]
S1-E5(100)	S:B:Q:I:TCP:S	[0.0% , 0.0%]	[120.0 , 135.0]	[230.0 , 238.0]	[510.0 , 590.0]
S1-E6(100)	S:B:Q:I:UDP:S	[0.0% , 0.0%]	[110.0 , 114.0]	[125.0 , 130.0]	[205.0 , 270.0]
S1-E7(100)	L:B:Q:I:TCP:S	[0.0% , 0.0%]	[630.0 , 860.0	[710.0 , 920.0]	[900.0 , 1260.0]
S1-E8(100)	L:B:Q:I:UDP:S	[0.0% , 0.0%]	[200.0 , 204.0]	[205.0 , 214.0]	[260.0 , 285.0]

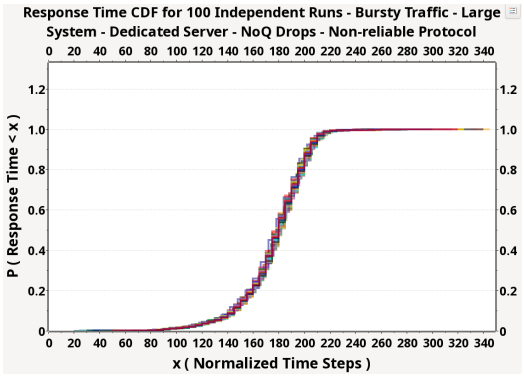
Note that in none of the experiments conducted in Scenario 1, did we encounter any failed requests. This is primarily because, with the non-reliable protocol, there were no queue drops, and the timeout duration for each reply was sufficiently long to prevent failed requests. Similarly, for the reliable protocol, the situation remained the same. However, it's important to note that using smaller timeout durations could potentially lead to failed requests. We further conducted additional experiments with shorter timeouts (16s and 8s), resulting in instances of failed requests. The results of these experiments are detailed in the appendix. Moreover, the only variations in performance observed when the reliable protocol triggered enough in experiments S1-E5 and S1-E7 which is validating the DST-driven insights from Chapter 5 about the



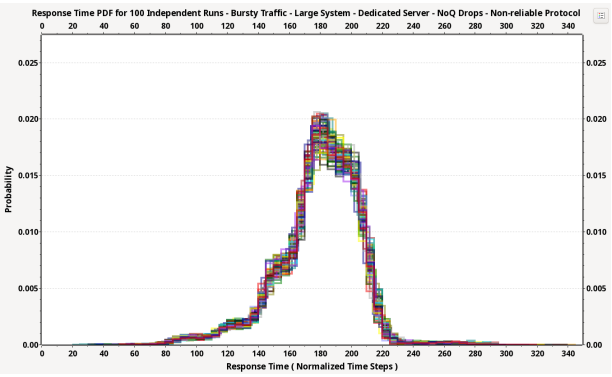
(a) S1-E8 - cdfs for burst length of 150s.



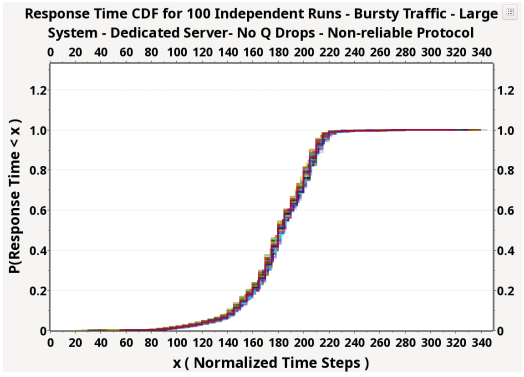
(b) S1-E8 - pdfs for burst length of 150s.



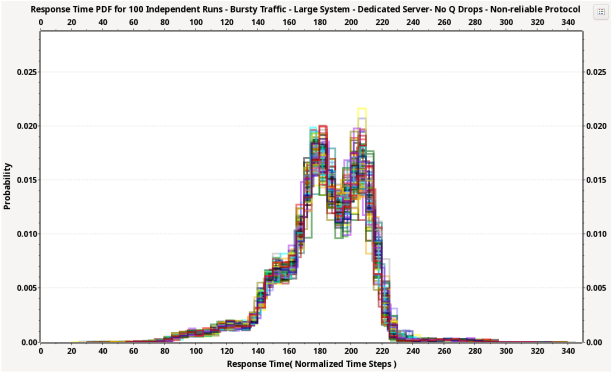
(c) S1-E8 - cdfs for burst length of 300s.



(d) S1-E8 - pdfs for burst length of 300s.



(e) S1-E8 - cdfs for burst length of 500s.



(f) S1-E8 - pdfs for burst length of 500s.

Figure 6.15: S1-E8 Application-layer response time distributions. [L:B:Q:I:UDP:S]

role of reliable protocol in producing wandering sets of non-zero measure which lead system to a BET non-compliant system with unpredictable performance behavior. Therefore, the main reason we see unpredictable performance with bursty workload is due to the reliable protocol triggers.

Table 6.15 compares the simulations conducted in Scenario 1 according to their compliance with the four Software Engineering Design Rules formally derived from Chapter 5 analyses and summarized in Section8.1.1.

Table 6.15: Comparison of simulations conducted in Scenario 1 according to compliance with software engineering design rules.

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	**Experiment Code**	**Software Eng. Design Rules**				**BET Compliant**	**cdfs**
		No. 1	**No. 2**	**No. 3**	**No. 4**		
S1-E1 (100)	S:P:Q:I:TCP:S	✓	✓	✓	✓	✓	6.8
S1-E2 (100)	S:P:Q:I:UDP:S	✓	✓	✓	✓	✓	6.9
S1-E3 (100)	L:P:Q:I:TCP:S	✓	✓	✓	✓	✓	6.10
S1-E4 (100)	L:P:Q:I:UDP:S	✓	✓	✓	✓	✓	6.11
S1-E5 (100)	S:B:Q:I:TCP:S	✓	✗	✓	✗	✗	6.12
S1-E6 (100)	S:B:Q:I:UDP:S	✓	✓	✓	✓	✓	6.13
S1-E7 (100)	L:B:Q:I:TCP:S	✓	✗	✓	✗	✗	6.14
S1-E8 (100)	L:B:Q:I:UDP:S	✓	✓	✓	✓	✓	6.15

6.3.3 Scenario 2: System with uncontested resources and finite queues

Scenario 2 is different from Scenario 1 by the introduction of non-zero probabilities for queue drops at the application layer. Operating under the same workload and without resource competition as in Scenario 1, Scenario 2's focus is on examining the impact of the application-layer queue drops on system performance. Additionally, as in Scenario 1, queue drop events may still occur at the application level within the INET simulated network layers and protocols. The following Figures and experiments display the outcomes of application-layer response time cdfs generated from 100 independent simulation runs in each experiment. These figures highlight that the introduction of application-layer queue drops induces variations between the cdfs that did not arise under Scenario 1. More particularly, the Scenario 2 cdfs would now fail standard statistical goodness-of-fit tests.

This result is expected through the DST-derived insights due to the application-layer queue drops introducing wandering sets of non-zero measure into the DST. Moreover, as TCP is used as the application-layer protocol, the queue drops result in

the re-transmission of any dropped application-layer events. From a DST perspective these re-transmissions create a loss of measure invariance, which also gives rise to the variations in the cdfs. As can be seen, the resulting Scenario 2 cdfs highlight a decrease in performance predictability in that the resulting cdfs now exhibit an increased dependence on each run's initial conditions. Hence, data collected from one run can no suffices to provide sufficient information to predict the system's performance in other runs, where information in these sense denoted Shannon information. The introduction of application-layer queue drop event has therefore transitioned the BET-compliant software system of Scenario 1 to the BET non-compliant software system of Scenario 2. Experiments S2-E1 to S2-E4 simulate a small distributed system, as illustrated in Figure 6.2, employing both normal and bursty on/off workload with both reliable and non-reliable protocols. Experiments S2-E5 to S2-E8 replicate the same conditions but simulate a larger distributed system, as depicted in Figure 6.3.

6.3.3.1 S2-E1: Small distributed software system exposed to normal traffic and employing reliable protocol with possible queue drops [S:P:Q:F:TCP:S]

Figure 6.16 presents the application-layer response time cdfs for the small distributed system exposed to normal traffic, derived from 100 independent simulation runs of the exemplary distributed software system. All runs within this experiment exposed to identical statistical workloads, with no resource competition but possible queue drop events. The following table provides a summary of the simulation parameters for Experiment S2-E1.

Table 6.16: S2-E1 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drops
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

Across the 100 runs, varying rates of queue drops were observed, resulting in between 0 to 150 failed requests in each run, compared to approximately 60,000 successful requests. Comparing this outcome with the results of Experiment 1 of Scenario

1, (S1-E1), it becomes evident how queue drops lead to multiple re-sends of requests (with a maximum limit of 8 retries per missed request) and subsequently contribute to increased queue congestion and longer response times. While the maximum response time in S1-E1 was 46 normalized time steps, it escalated to around 360 steps in experiments with more frequent queue drops. In fact, the maximum response time ranged from 35 to 364 across the 100 runs, depending on the occurrence of queue drops.

Based on the insights from Chapter 5, Experiment S2-E1 violates the Software Engineering Design Rule 1 and 2 (summarized in 8.1.1) which results in producing two types of wandering sets of non-zero measure which in turn leads to a BET non-compliant system. The simulation results depicted in Figure 6.16 validates the result of formal analysis in Chapter 5.

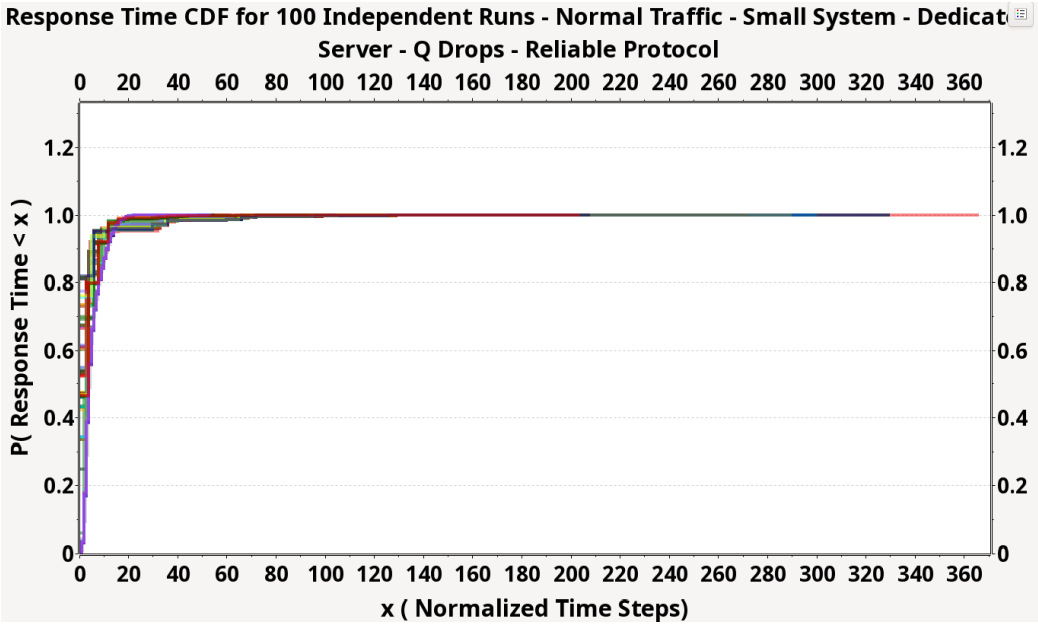


Figure 6.16: S2-E1 Application-layer response time distributions. [S:P:Q:F:TCP:S]

6.3.3.2 S2-E2: Small distributed software system exposed to normal traffic and employing non-reliable protocol with possible queue drops. [S:P:Q:F:UDP:S]

Experiment S2-E2 mirrors Experiment S2-E1, but with removing the reliable protocol effect. The results confirm and validate the insights derived from DST for software system deployments where queue drop events may occur. Experiment S2-E2 and

S2-E4 are one of the most important experiments because they isolate the impact of queue drops on the system. A summary of the simulation specific parameters for S2-E2 is provided in Table 6.17.

Table 6.17: S2-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drops
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

Figure 6.17 presents the outcomes of Experiment 2, where the reliable protocol was excluded compared to Experiment 1. Throughout the 100 runs, varying rates of queue drops were observed, resulting in failure rates between 0.01% to 1.6%. Comparing these results with those of S1-E2, it becomes evident how queue drops contribute to variations in response time cdfs. While the maximum response time in S1-E1 was 46 time steps, it increased to around 60 in runs with larger queues and therefore less frequent queue drops. The slight increase in response time is due to overhead of queue drops and finite queue management compared to infinite queues. In fact, across the 100 runs, the maximum response time ranged from 19 to 59, depending on the frequency of queue drops. In shorter queues, we have a better response time for successful requests but a higher failure rate.

Based on the insights from Chapter 5, Experiment S2-E2 violates the Software Engineering Design Rule 1 (summarized in 8.1.1), which results in wandering sets of non-zero measure and leads to a BET non-compliant system. The simulation results depicted in Figure 6.17 validate the result of formal analysis in Chapter 5.

6.3.3.3 S2-E3: Large distributed software system exposed to normal traffic and employing reliable protocol with possible queue drops [L:P:Q:F:TCP:S]

Experiment S2-E3 replicates S1-E3 but introduces queue drops into the system. Similar to previous observations, this leads to the re-sending of dropped requests/internal requests, causing congestion in queues. Consequently, longer waiting times ensue, triggering additional re-sending of requests due to timeouts, thereby perpetuating

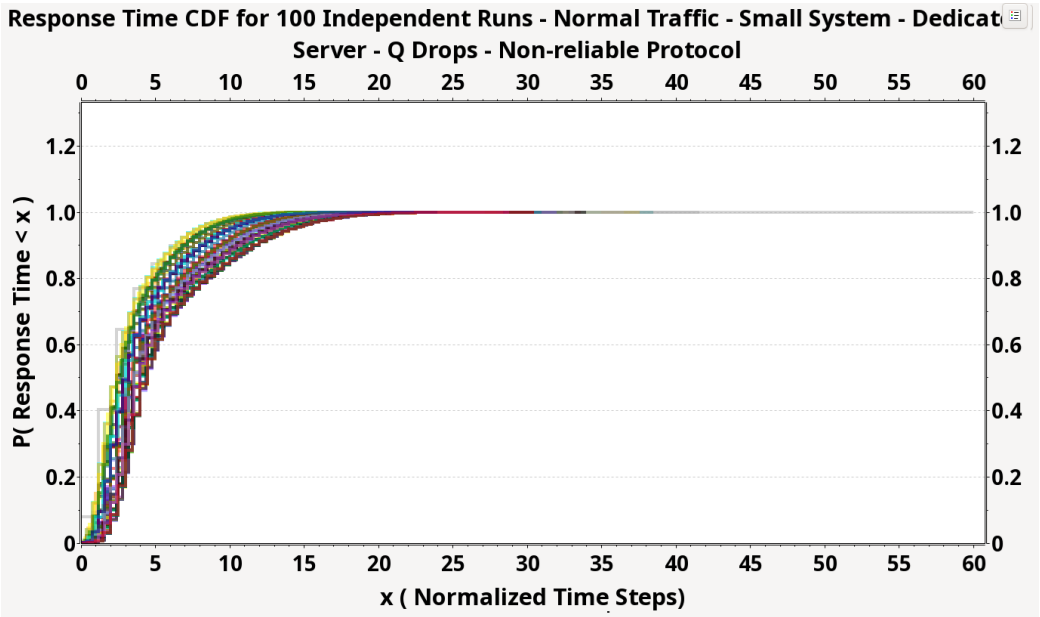


Figure 6.17: S2-E2 Application-layer response time distributions. [S:P:Q:F:UDP:S]

longer queues and response times compared to S1-E3. In runs with more queue drops, response times increase from around 75 normalized time steps to 500 steps. Based on the insights from Chapter 5, Experiment S2-E3 violates Software Engineering Design Rules 1 and 2 (summarized in 8.1.1), resulting in the production of two types of wandering sets of non-zero measure, which in turn leads to a BET non-compliant system. The simulation results depicted in Figure 6.18 validate the formal analyses presented in Chapter 5.

Table 6.18: S2-E3 specific simulation parameters.

Simulation Parameter	**Value**
Workload	Poisson
System Architecture	Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drops
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

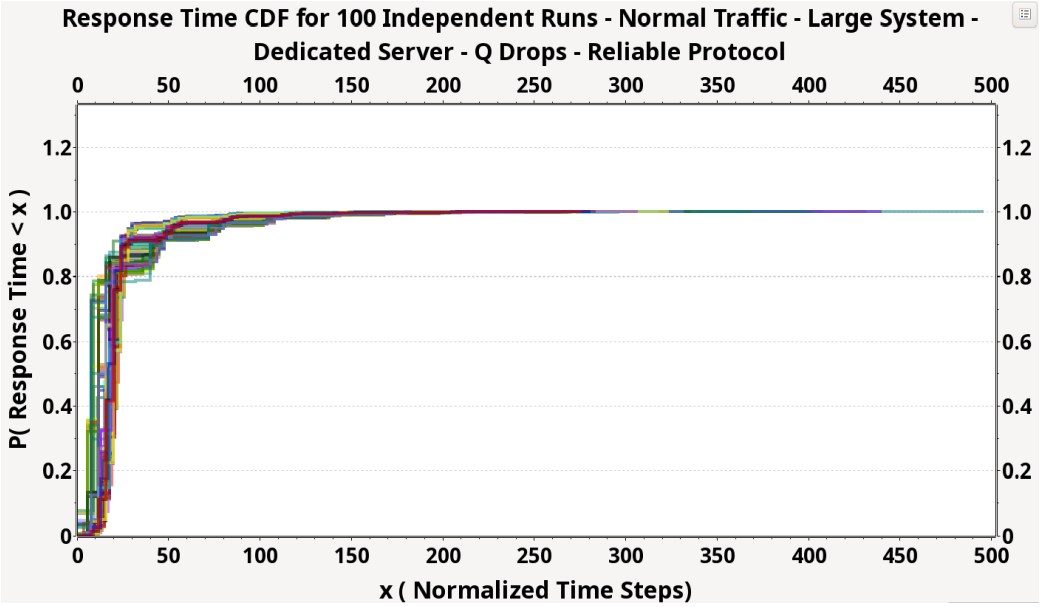


Figure 6.18: S2-E3 Application-layer response time distributions. [L:P:Q:F:TCP:S]

6.3.3.4 S2-E4: Large distributed software system exposed to normal traffic and employing non-reliable protocol with possible queue drops. [L:P:Q:F:UDP:S]

Experiment S2-E4 mirrors S2-E3 but employs a non-reliable protocol. With non-reliable protocol, any dropped request is deemed failed, effectively reducing congestion as dropped packets are not re-sent. In contrast to S1-E4, which operated with infinite queues, the current experiment demonstrates improved performance with shorter queues, although at the expense of a higher failure rate due to the absence of retransmission.

Table 6.19: S2-E4 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drops
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

Notably, the response times do not extend as dramatically as they did with the retransmissions of a reliable protocol in the previous experiment. Instead, the response

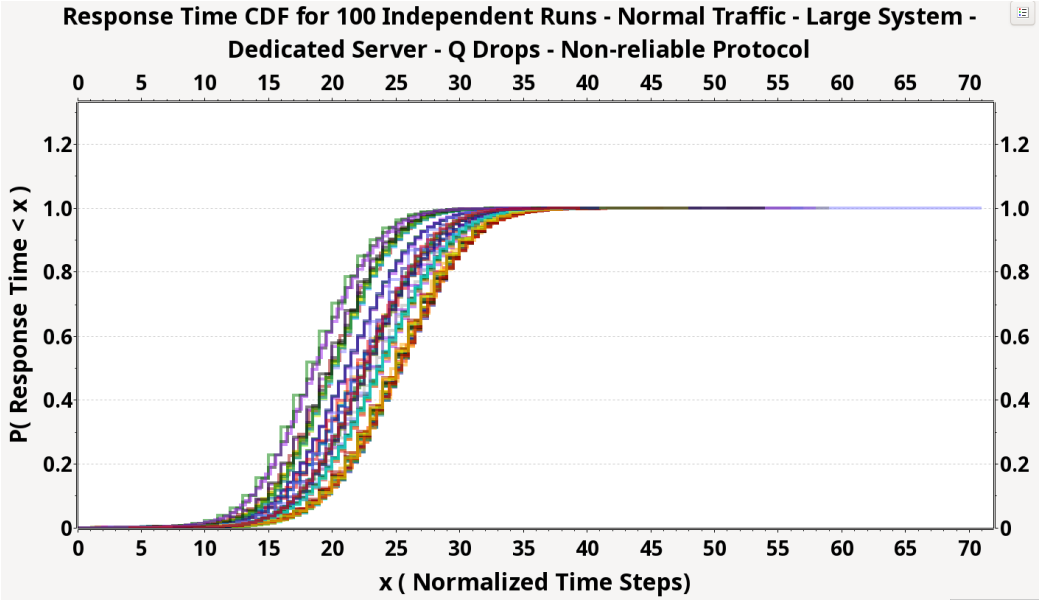


Figure 6.19: S2-E4 Application-layer response time distributions. [L:P:Q:F:UDP:S]

time increases modestly compared to the similar experiment in the first scenario presented in S1-E4. However, there is a wider dispersion in response times, reflecting the direct impact of packet loss (queue drops) on performance predictability, rather than the compounded delays caused by retransmissions. On the other hand, when compared with S2-E3, this clearly illustrates the direct effects of the protocol choice on system performance predictability. The result of S2-E4 validates the Chapter 5 formal analyses where violating Software Engineering Design Rule 1 (summarized in 8.1.1) leads to production of wandering sets of non-zero measure which breaks the required condition for BET-compliance.

6.3.3.5 S2-E5: Small distributed software system exposed to bursty traffic and employing reliable protocol with possible queue drop events [S:B:Q:F:TCP:S]

Experiment S2-E5 replicates the conditions of S2-E1 but introduces an incoming bursty workload, consisting of bursts lasting 300 seconds with a traffic intensity at least 10 times higher. It should be noted that the burst length and intensity are configurable as before. Upon the arrival of a burst, the queues begin to accumulate, allowing the initial requests to be processed quickly and advance to the next station. However, subsequent requests encounter prolonged waits within the queues of each

server. As the burst concludes, the queues gradually diminish until the occurrence of the next burst. If we compare these results with S1-E5, we observe that the maximum response time decreased from approximately 700 to 550. This reduction occurs because the queues are finite, which imposes a limit on the waiting times in queues. The following table provides a summary of the simulation parameters for S2-E5.

Table 6.20: S2-E5 specific simulation parameters.

Simulation Parameter	**Value**
Workload	Bursty
Burst Length	300s
Burst Intensity	10x
System Architecture	Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drop events
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

Based on the insights from Chapter 5, Experiment S2-E5 violates Software Engineering Design Rules 1, 2 and 4 (summarized in 8.1.1), resulting in the production of two types of wandering sets of non-zero measure, which in turn leads to a BET non-compliant system. The simulation results depicted in Figure 6.20 validate the formal analyses presented in Chapter 5.

6.3.3.6 S2-E6: Small distributed software system exposed to bursty traffic and employing non-reliable protocol with possible queue drops. [S:B:Q:F:UDP:S]

Experiment S2-E6 replicates the conditions of S2-E5 but removing the effects of reliable protocol. Using a reliable protocol entails re-sending the request if no response is received within a specified time frame. This, in turn, leads to heightened queue congestion, consequently causing longer response times as shown in Figure 6.20, compared to the non-reliable protocol shown in Figure 6.21. Table 6.21 provides a summary of the simulation parameters for Experiment E2-E6.

Figure 6.21 clearly illustrates the effect of queue drops on the response time distribution. Similar to the previous experiment, the longest response time is shorter compared to S1-E6 due to finite queues, which impose a limit on the queue waiting

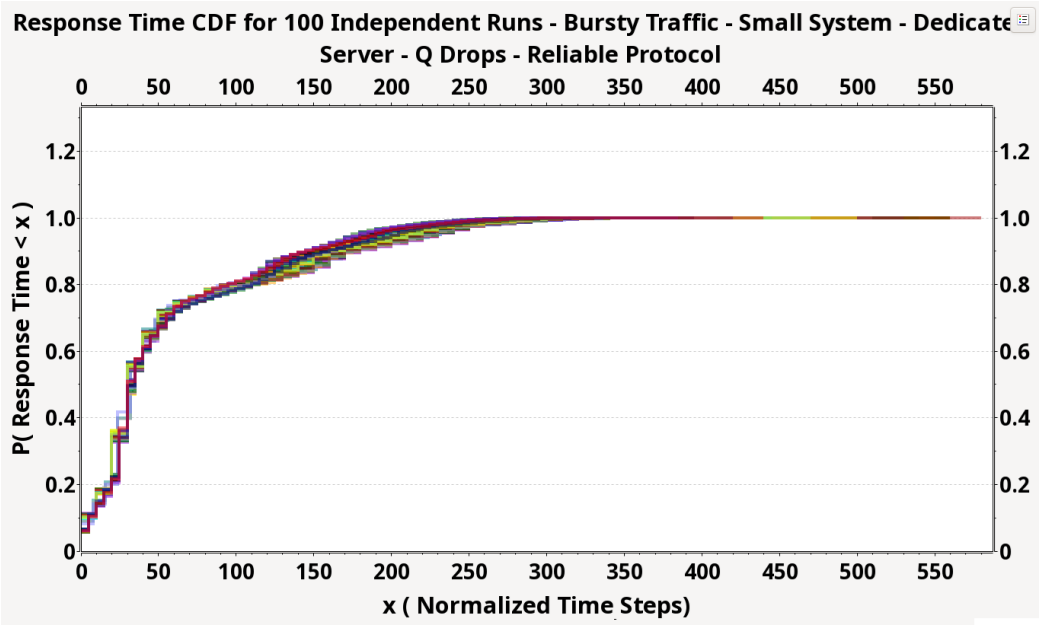


Figure 6.20: S2-E5 Application-layer response time distributions. [S:B:Q:F:TCP:S]

Table 6.21: S2-E6 specific simulation parameters.

Simulation Parameter	**Value**
Workload	Bursty
Burst Length	300s
Burst Intensity	10x
System Architecture	Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drop events
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

time. However, this leads to a failure rate ranging from 0.16% to 15.7%, whereas in S1-E6, we observed zero failed requests. Experiment S2-E6 is particularly significant because it highlights the effect of queue drops on the system's BET-compliance state, thereby validating the insights derived from Chapter 5, which indicate that queue drops result in the production of wandering sets of non-zero measure, leading to unpredictable behavior. Moreover, bursty traffic results in a high utilization rate in servers, which itself violates the measure-preserving condition required for BET-compliance.

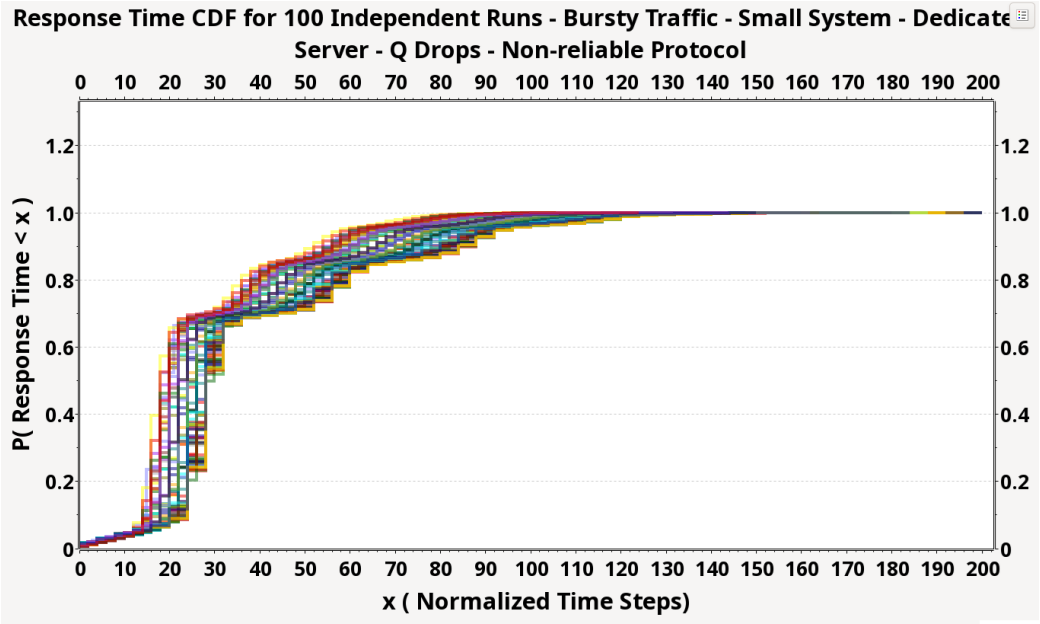


Figure 6.21: S2-E6 Application-layer response time distributions. [S:B:Q:F:UDP:S]

6.3.3.7 S2-E7: Large distributed software system exposed to bursty traffic and employing reliable protocol with possible queue drops. [L:B:Q:F:TCP:S]

Experiment S2-E7 duplicates the setup of Experiment S2-E5, but on a larger distributed system, characterized by bursts spanning 300 seconds with a traffic intensity at least 10 times higher. Upon the initiation of a burst, queues start to accumulate with subsequent requests experiencing prolonged waits within the queues of each server. As the burst subsides, the queues gradually diminish until the next burst emerges. In comparison to S1-E7, which features infinite queues, as expected we observe a decrease in the longest response time from around 1400 steps to around 1000, albeit with a higher failure rate. However, queue drops continue to contribute to greater variability in the response time cdfs and consequently leading to unpredictability in performance. Table 6.22 provides a summary of the simulation parameters for Experiment S2-E7.

Based on the insights from Chapter 5, Experiment S2-E7 violates Software Engineering Design Rules 1 due to queue drops and Rule 2 due to overuse of reliable protocol (summarized in 8.1.1), resulting in the production of two types of wandering sets of non-zero measure, which in turn leads to a BET non-compliant system. Moreover, bursty traffic results in a high utilization rate in servers, which itself vio-

Table 6.22: S2-E7 specific simulation parameters.

Simulation Parameter	Value
Workload	Bursty
Burst Length	300s
Burst Intensity	10x
System Architecture	Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drop events
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

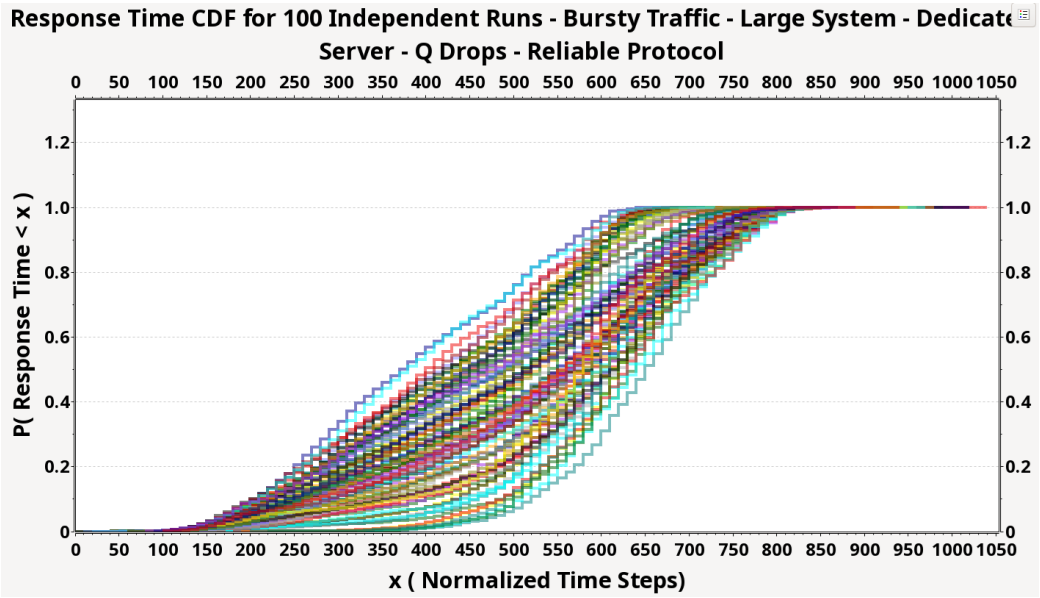


Figure 6.22: S2-E7 Application-layer response time distributions. [L:B:Q:F:TCP:S]

lates Rule 4 and the measure-preserving condition required for BET-compliance. The simulation results depicted in Figure 6.22 which shows a wide range of cdfs validate the formal analyses presented in Chapter 5.

6.3.3.8 S2-E8: Large distributed software system exposed to bursty traffic and employing non-reliable protocol with possible queue drops. [L:B:Q:F:UDP:S]

Experiment S2-E8 replicates the conditions of Experiment S2-E7 but applying a non-reliable protocol. Using a reliable protocol in Experiment S2-E7 leads to heightened queue congestion, consequently causing longer response times as shown in Figure

6.22, compared to the non-reliable protocol shown in Figure 6.23. The following table provides a summary of the simulation parameters for Experiment S2-E8.

Table 6.23: S2-E8 specific simulation parameters.

Simulation Parameter	Value
Workload	Bursty
Burst Length	300s
Burst Intensity	10x
System Architecture	Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Dedicated
Type of Queues	Finite with possible queue drop events
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/300$ ms

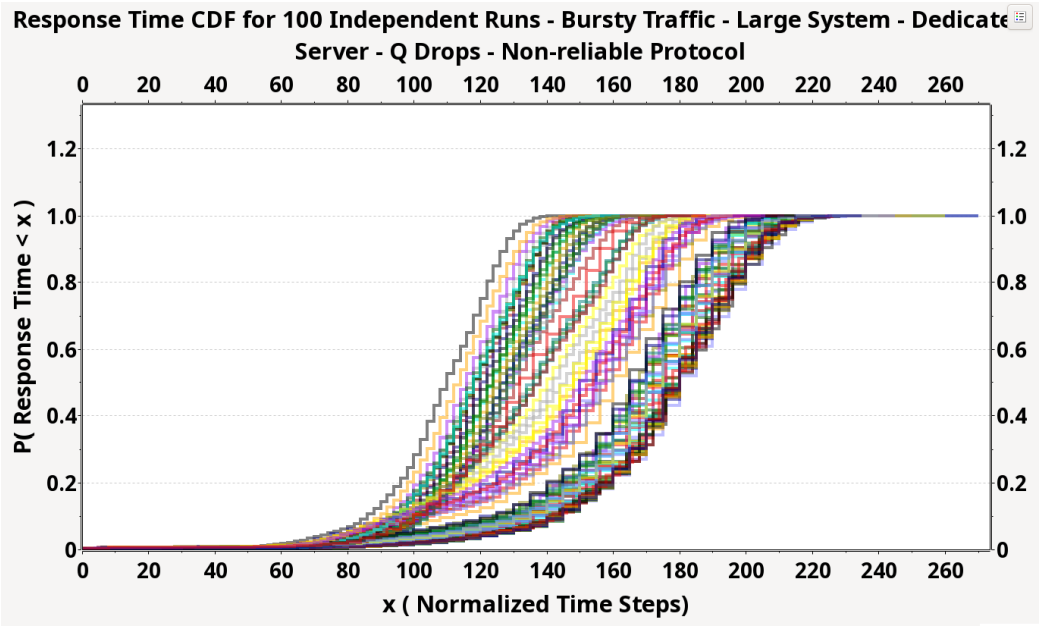
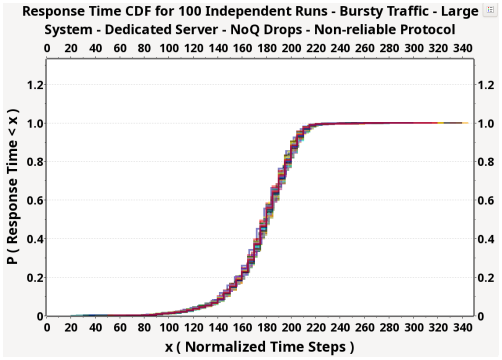
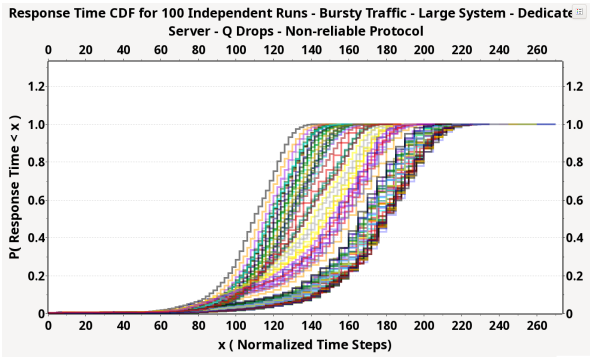


Figure 6.23: S2-E8 Application-layer response time distributions. [L:B:Q:F:UDP:S]

Since the only distinction between S1-E8 and S2-E8 is the presence of queue drops in the latter, comparing these two in Figure 6.24 can directly illustrates the impact of queue drops which violates Software Engineering Design Rule 1 on BET-compliance and run-time performance predictability which is validating the DST-driven insights from Chapter 5.



(a) S1-E8 ten hours of simulation. [L:B:Q:I:UDP:S]



(b) S2-E8 ten hours of simulation. [L:B:Q:F:UDP:S]

Figure 6.24: The effect of queue drops on the baseline scenario predictability under bursty traffic.

6.3.3.9 Scenario 2 Results Summary

The results of scenario 2 experiments are summarized in Table 6.24. The odd-numbered experiments with reliable protocols consistently show lower failure rates and generally wider response times, validating the DST-driven insights of Chapter 5 regarding the effect of queue drops and reliable protocols on breaking system's BET-compliance by producing wandering sets of non-zero measure. In contrast, even-numbered experiments with non-reliable protocols exhibit higher failure rates and generally narrower response times, as there are fewer wandering sets of non-zero measure created by the reliable protocol. These results clearly show the effect of two types of wandering sets of non-zero measure produced by queue drops and reliable protocols as explained in details in Chapter 5 on system's BET-compliance.

Table 6.24: Scenario 2 experiments results comparison.

Ensemble Code	**Experiment Code**	**Failure Rate [Min., Max.]**	**Response Time (NTS) Percentiles [Min.,Max.]**		
			90^{th}	**95^{th}**	**99.99^{th}**
S2-E1(100)	S:P:Q:F:TCP:S	[0.0% , 0.2%]	[4.0 , 11.0]	[6.0 , 33.5]	[33.0 , 354.0]
S2-E2(100)	S:P:Q:F:UDP:S	[0.1% , 1.6%]	[5.0 , 12.0]	[7.0 , 15.0]	[11.0 , 58.0]
S2-E3(100)	L:P:Q:F:TCP:S	[0.0% , 1.0%]	[20.0 , 50.6]	[29.4 , 100.0]	[146.0 , 497.0]
S2-E4(100)	L:P:Q:F:UDP:S	[0.0% , 4.0%]	[23.0 , 31.0]	[22.0 , 33.8]	[28.0 , 70.5]
S2-E5(100)	S:B:Q:F:TCP:S	[0.0% , 0.1%]	[140.2 , 190.0]	[170.5 , 240.0]	[242.5 , 470.3]
S2-E6(100)	S:B:Q:F:UDP:S	[0.2% , 15.7%]	[51.0 , 89.8]	[60.4 , 100.8]	[81.0 , 156.5]
S2-E7(100)	L:B:Q:F:TCP:S	[0.0% , 1.0%]	[560.5 , 770.0]	[590.0 , 800.3]	[625.0 , 932.8]
S2-E8(100)	L:B:Q:F:UDP:S	[0.5% , 64.0%]	[126.0 , 209.0]	[130.0 , 215.0]	[139.0 , 264.0]

Table 6.25 compares the simulations conducted in Scenario 2 according to their compliance with the four Software Engineering Design Rules formally derived from Chapter 5 analyses and summarized in Section8.1.1.

Table 6.25: Comparison of simulations conducted in Scenario 2 according to compliance with software engineering design rules.

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	**Experiment Code**	**Software Eng. Design Rules**				**BET Compliant**	**cdfs**
		No. 1	**No. 2**	**No. 3**	**No. 4**		
S2-E1 (100)	S:P:Q:F:TCP:S	✗	✗	✓	✓	✗	6.16
S2-E2 (100)	S:P:Q:F:UDP:S	✗	✓	✓	✓	✗	6.17
S2-E3 (100)	L:P:Q:F:TCP:S	✗	✗	✓	✓	✗	6.18
S2-E4 (100)	L:P:Q:F:UDP:S	✗	✓	✓	✓	✗	6.19
S2-E5 (100)	S:B:Q:F:TCP:S	✗	✗	✓	✗	✗	6.20
S2-E6 (100)	S:B:Q:F:UDP:S	✗	✗	✓	✓	✗	6.21
S2-E7 (100)	L:B:Q:F:TCP:S	✗	✗	✓	✗	✗	6.22
S2-E8 (100)	L:B:Q:F:UDP:S	✗	✗	✓	✓	✗	6.23

6.3.4 Scenario 3: System with shared servers servicing fixed number of VMs with infinite queues.

The figures 6.25 to 6.30 display the results of the application-layer response time cdfs generated from 100 independent simulation runs for each experiment. The exemplary software system is identical to the distributed software system in Scenario 1, with the same statistical Poisson workloads. However, in this scenario, the application-layer distributed system is simulated to operate within server-level VMs. More specifically, each simulated physical server must now support a composite of VMs into which the application-layer distributed software systems are placed, with each simulation run denoting different levels of active VMs. This means that the simulated base OS within each physical server must now schedule the VMs for execution, i.e., via a standard UNIX fair (round robin) scheduling process. This induces a time slicing of the VMs, similar to real-world cloud deployments where executing VMs exist as processes within the DOM-0 physical server's base OS, which are brought into and out of execution via this OS's scheduler, adhering to the fair scheduling policy applied across all OS executing processes.

Given that we have already demonstrated in both Scenario 1 and Scenario 2 that bursty traffic disrupts predictability triggering queue drops, reliable protocol and high utilization rates, there is no necessity to replicate experiments involving bursty traffic.

6.3.4.1 S3-E1: Small distributed system on shared servers exposed to normal traffic and employing reliable protocol. [S:P:F:I:TCP:S]

Figure 6.25 presents the application-layer response time cdfs obtained from 100 independent simulation runs of the exemplary distributed software system in Figure 6.2. All runs within this experiment were subjected to identical statistical workload executed on shared servers, with no occurrence of application layer queue drop events. Table 6.26 provides a summary of the simulation parameters for this experiment. Figure 6.25 displays the response time cdfs of the system over 10 hours of simulation time.

Table 6.26: S3-E1 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

Figure 6.25 clearly illustrates the effect of fair OS scheduling on the response time distribution. Unlike previous experiments in Scenario 2 that involved queue drops, this experiment focuses on the implications of fair scheduling policies. These cdfs reveal that fair scheduling depending on the load on shared servers can dramatically affect the predictability of response times, dispersing 90^{th} percentile response times between 15 normalized time steps on lightly loaded servers to 140 steps on heavily loaded servers. The wider response time cdfs in S3-E1 experiment are significantly influenced by the fair (round robin) scheduling and overuse of reliable protocol which violate Software Engineering Design Rule 3 and 2 respectively. The effect of violating these rules is clearly illustrated in Figure 6.26b when compared to Figure 6.26a where 100 runs of the same system were executed on dedicated servers.

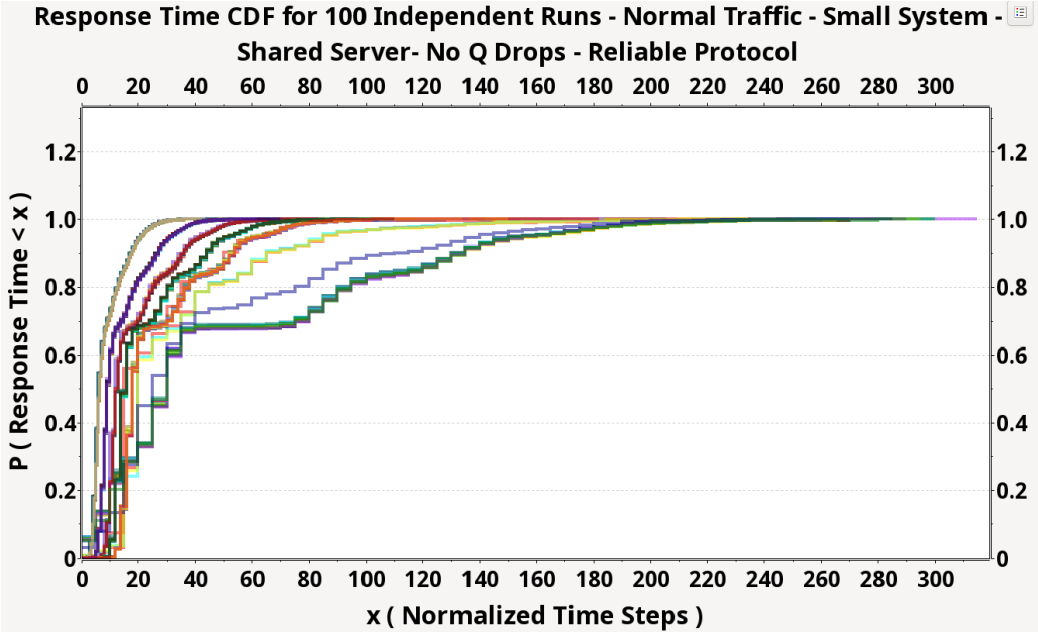
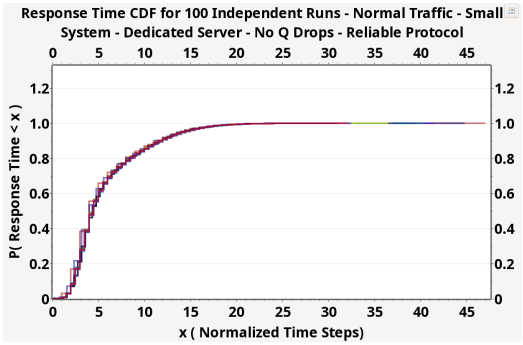
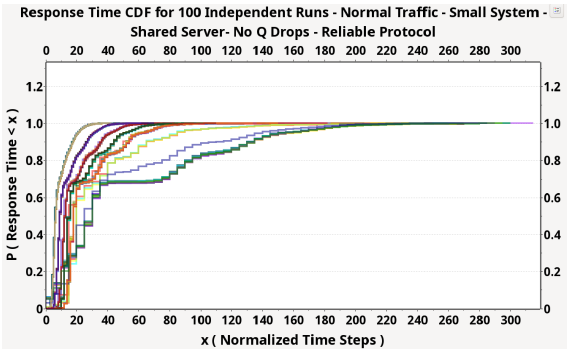


Figure 6.25: S3-E1 Application-layer response time distributions. [S:P:F:I:TCP:S]



(a) S1-E1 ten hours of simulation. [S:P:Q:I:TCP:S]



(b) S3-E1 ten hours of simulation. [S:P:F:I:TCP:S]

Figure 6.26: The effect of hypervisor fair scheduling (violating Software Engineering Design Rule 3) on BET-compliance of baseline scenario.

6.3.4.2 S3-E2: Small distributed software system exposed to normal traffic and employing non-reliable protocol. [S:P:F:I:UDP:S]

Experiment S3-E2 mirrors Experiment S3-E1, but with the application of a non-reliable protocol. The results confirm and validate the DST-derived insights for software systems deployments on shared resources with fair scheduling. Comparing the results of this experiment with S3-E1, we can observe that in the previous experiment, the reliable protocol is activated more frequently resulting in noticeable differences in

performance between the reliable and non-reliable protocols, particularly on servers with higher loads. In the non-reliable protocol scenario, servers with lower loads exhibit similar behavior, but those with higher loads demonstrate improved performance compared to S3-E1, narrowing the 90^{th} percentile between 15 and 70 time steps. However, despite this improvement, the performance still shows considerable variability and cannot be reliably predicted. Table 6.27 offers a summary of the simulation parameters for S3-E2.

Table 6.27: S3-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

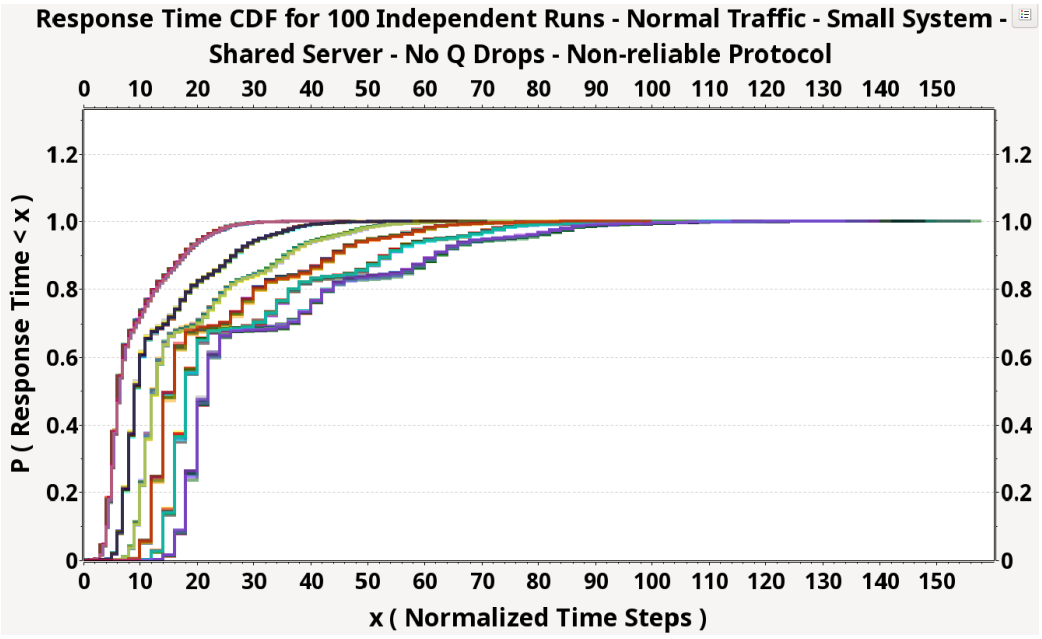


Figure 6.27: S3-E2 Application-layer response time distributions. [S:P:F:I:UDP:S]

Experiment S3-E2 is particularly significant because it isolates the effect of violating Software Engineering Design Rule 3 by applying fair OS scheduling. This validates the insights derived from Chapter 5, which indicate that such scheduling violates the measure-preserving condition required for BET-compliance.

6.3.4.3 S3-E3: Large distributed system on shared servers exposed to normal traffic and employing reliable protocol. [L:P:F:I:TCP:S]

Experiment S3-E3 mirrors Experiment S1-E3, with 100 independent identical runs but executed inside VMs on shared servers instead of dedicated servers. Figure 6.28 illustrates the impact of fair scheduling on BET-compliance and therefore performance predictability. The results reveal that depending on the variety in number of VMs on shared servers, the performance predictability decreases. This is evident from the broader distribution of response times observed compared to S1-E3 results as depicted in Figure 6.29b. Table 6.28 provides a summary of the simulation parameters for this experiment.

Table 6.28: S3-E3 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

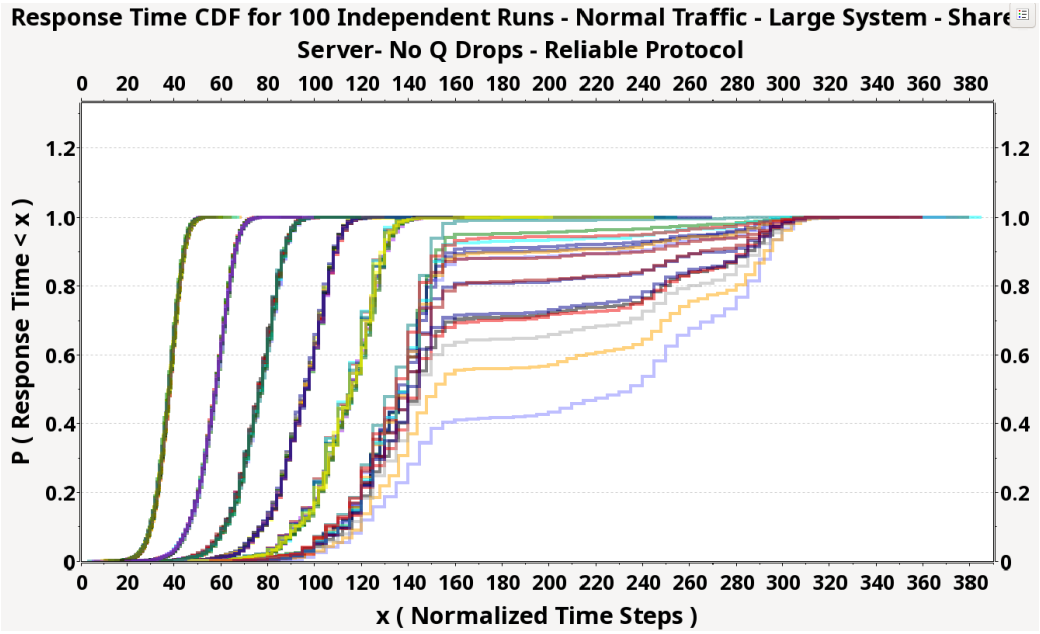
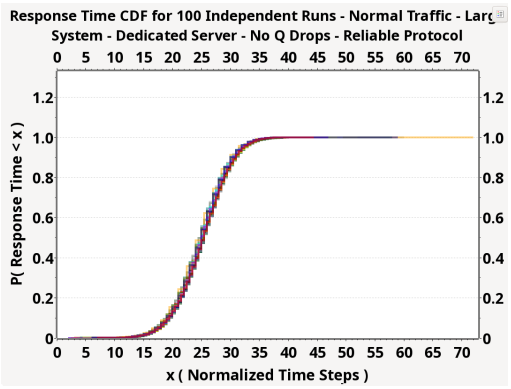
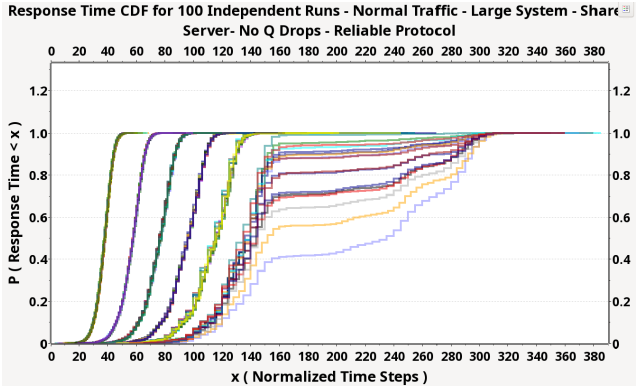


Figure 6.28: S3-E3 Application-layer response time distributions. [L:P:F:I:TCP:S]

Based on the insights from Chapter 5, Experiment S3-E3 violates Software Engineering Design Rule 2 in servers with higher loads and Rule 3 by applying fair OS scheduling (summarized in 8.1.1). These violations result in producing wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.28 validate the findings of the formal analysis in Chapter 5.



(a) S1-E3 ten hours of simulation. [L:P:Q:I:TCP:S]



(b) S3-E3 ten hours of simulation. [L:P:F:I:TCP:S]

Figure 6.29: The impact of hypervisor fair scheduling (violating Software Engineering Design Rule 3) on BET-compliance of baseline scenario.

6.3.4.4 S3-E4: Large distributed software system exposed to normal traffic and employing non-reliable protocol. [L:P:F:I:UDP:S]

Experiment S3-E4 mirrors Experiment S3-E3, but with the implementation of a non-reliable protocol. The results reaffirm and validate the DST-derived insights regarding software systems deployments on shared resources with fair scheduling. Table 6.29 offers a summary of the simulation parameters for S3-E4.

Table 6.29: S3-E4 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared
Type of Queues	Infinite
Service time per server	$p(x) = \lambda e^{-\lambda x}$; $\lambda = 1/400$ ms

In the experiment with the non-reliable protocol, servers with lower loads exhibit comparable behavior, while those with higher loads demonstrate enhanced performance and reliability compared to S3-E3. However, despite this enhancement, performance still exhibits significant variability and cannot be reliably predicted, especially if the number of hosted VMs changes over time or with each deployment. Comparing the outcomes of this experiment with those of S3-E3, it becomes evident that in that experiment the reliable protocol is employed more frequently, particularly on servers with higher loads. This leads to noticeable discrepancies in performance not only between the reliable and non-reliable protocols but also among servers with similar high loads, as depicted in Figure 6.52a.

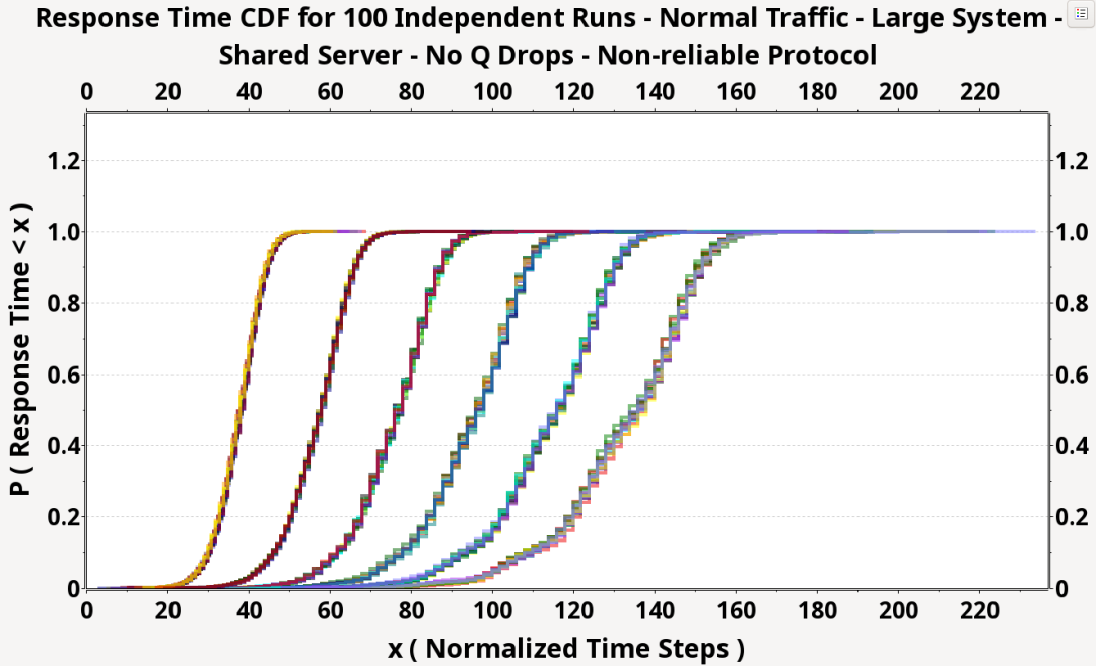
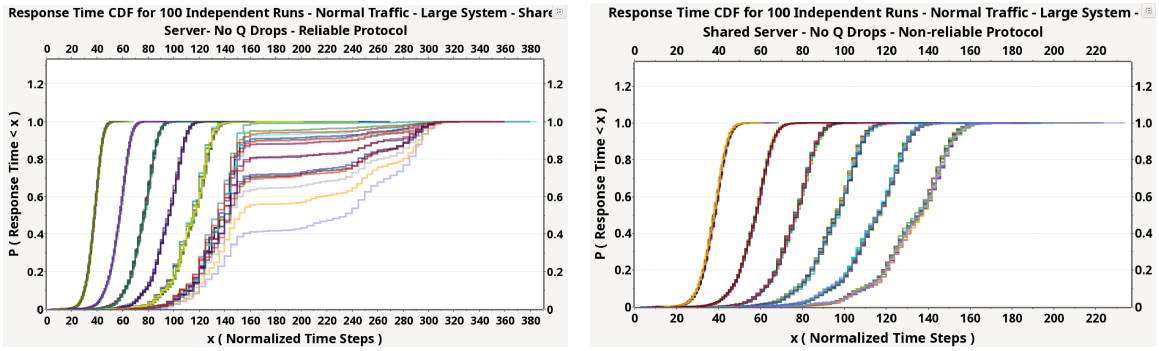


Figure 6.30: S3-E4 Application-layer response time distributions. [L:P:F:I:UDP:S]

Based on the insights from Chapter 5, Experiment S3-E4 violates Software Engineering Design Rule 3 by applying a fair OS scheduling (summarized in 8.1.1). This violation results in breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.30 validate the findings of the formal analysis in Chapter 5.



(a) S3-E3 ten hours of simulation. [L:P:F:I:TCP:S]

(b) S3-E4 ten hours of simulation. [L:P:F:I:UDP:S]

Figure 6.31: The impact of reliable protocol on the performance predictability of shared servers especially overloaded ones within a cloud environment.

6.3.4.5 Scenario 3 Results Summary

As can be seen through the results of Scenario 3's experiments, varying the number of VMs a physical server must service, significantly changes the distributed software system's response time distributions. But, as queue drops do not occur within Scenario 3, consistency exists between simulation runs that involve the same number of physical layers supported VMs. It should be noted that the six visible modes in Scenario 3 results are produced due to the simulation's six possible different loads on the server. In real-world systems, there is no such constraint on server loads, especially with time-varying loads. This causes the system behavior to move across different cdfs, making it unpredictable. However, variations in cdfs arise between differing numbers of VMs per-physical server as increasing the number of VMs entails that more time must be devoted by the base OS schedule to the task swapping of the VMs, i.e., as required to time slice the set of "executing" VMs over the smaller set of simulated CPUs that can perform each VMs run-time execution. Hence, performance predictability is retained when the number of executing VMs remains constant and is known and for a known distributed software system. But performance predictability is lost between instances with differing numbers of executing VMs. This latter point is important in real-world cloud deployments as, in general, the number of VMs "executing" at any given time within any given physical server is itself a stochastic process. Hence, its value change dynamically and is not knowable a priori, thereby, resulting in a direct loss of performance predictability. Again, these simulation results validate the DST-derived insights. The following table summarizes the failure

rate and 95^{th} percentile for experiments done in Scenario 3. Since there are no queue drops, and the servers are fast enough to service requests before the timeout, we do not observe any failures in any of the experiments.

Table 6.30: Scenario 3 experiments results comparison.

Ensemble Code	Experiment Code	Failure Rate [Min. , Max.]	Response Time (NTS) Percentiles [Min. , Max.]		
			90^{th}	95^{th}	99.99^{th}
S3-E1(100)	S:P:F:I:TCP:S	[0.0% , 0.0%]	[17.0 , 130.0]	[21.0 , 155.5]	[38.5 , 285.0]
S3-E2(100)	S:P:F:I:UDP:S	[0.0% , 0.0%]	[17.0 , 62.0]	[20.0 , 76.0]	[37.5 , 138.0]
S3-E3(100)	L:P:F:I:TCP:S	[0.0% , 0.0%]	[44.0 , 292.0]	[46.0 , 295.0]	[53.0 , 325.0]
S3-E4(100)	L:P:F:I:UDP:S	[0.0% , 0.0%]	[44.0 , 150.0]	[46.5 , 154.0]	[53.5 , 190.0]

Table 6.31 compares the simulations conducted in Scenario 3 according to their compliance with the four Software Engineering Design Rules formally derived from Chapter 5 analyses and summarized in Section8.1.1. In Table 6.31, "T" stands for "Transitional" refers to situations where an LDSS deployment can move back and forth across the Software Engineering Design Rule boundary due to changes in its run-time deployment and/or dynamic variations in its service workloads.

Table 6.31: Comparison of simulations conducted in Scenario 3 according to compliance with software engineering design rules.

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	Experiment Code	Software Eng. Design Rules				BET Compliant	cdfs
		No. 1	No. 2	No. 3	No. 4		
S3-E1 (100)	S:P:F:I:TCP:S	✓	✗	✗	*T*	✗	6.25
S3-E2 (100)	S:P:F:I:UDP:S	✓	✓	✗	*T*	✗	6.27
S3-E3 (100)	L:P:F:I:TCP:S	✓	✗	✗	*T*	✗	6.28
S3-E4 (100)	L:P:F:I:UDP:S	✓	✓	✗	*T*	✗	6.30

6.3.5 Scenario 4: System with shared servers servicing fixed number of VMs with finite queues

Scenario 4 repeats the simulations of Scenario 3, but with the inclusion of non-zero application-layer queue drop probabilities using finite queues. The result of Scenario 4's experiments validates the DST-derived insights of Chapter 5 by showing that the introduction of application-layer queue drop events creates increased levels of statistical variation between the simulation runs, where an increased spread now can be seen to exist between runs involving the same number of VMs per physical server. As with Scenario 2, these variations are caused, from a DST perspective, by the queue drop events causing wandering sets of non-zero measure to arise with the application-layer use of TCP then causing re-transmission of the dropped events which then induces a lack of measure invariance across simulation runs. All runs within this scenario exposed to identical statistical workloads executing on shared servers with possible queue drop events.

6.3.5.1 S4-E1: Small distributed software system on shared servers exposed to normal traffic and employing reliable protocol with possible queue drop events [S:P:F:F:TCP:S]

Experiment S4-E1 mirrors the S3-E1 experiment by introducing possible queue drops. It holds significant importance as it collectively investigates the impacts of scheduling, queue drops, and reliable protocol. Application-layer response times distributions (cdfs), derived from 100 independent simulation runs of the exemplary distributed software system presented in Figure 6.32. In the cdfs plot, it's evident that queue drops contribute to increased variation and spread among the response times. This outcome aligns with the BET-compliance insights from Chapter 5, where queue drops and their re-transmissions generate wandering sets of non-zero measure, leading to lose of BET-compliance and therefore unpredictability even among servers experiencing the same load. Table 6.32 provides a summary of the simulation parameters for this experiment.

The increase in the variability of response times as shown in Figure 6.43b due to queue drops is not merely a random occurrence identifiable through simulation; rather, it stems from fundamental formal behavioral characteristics of software systems. As discussed in Chapter 5, queue drops produce wandering sets of non-zero

Table 6.32: S4-E1 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared
Type of Queues	Finite with possible queue drops

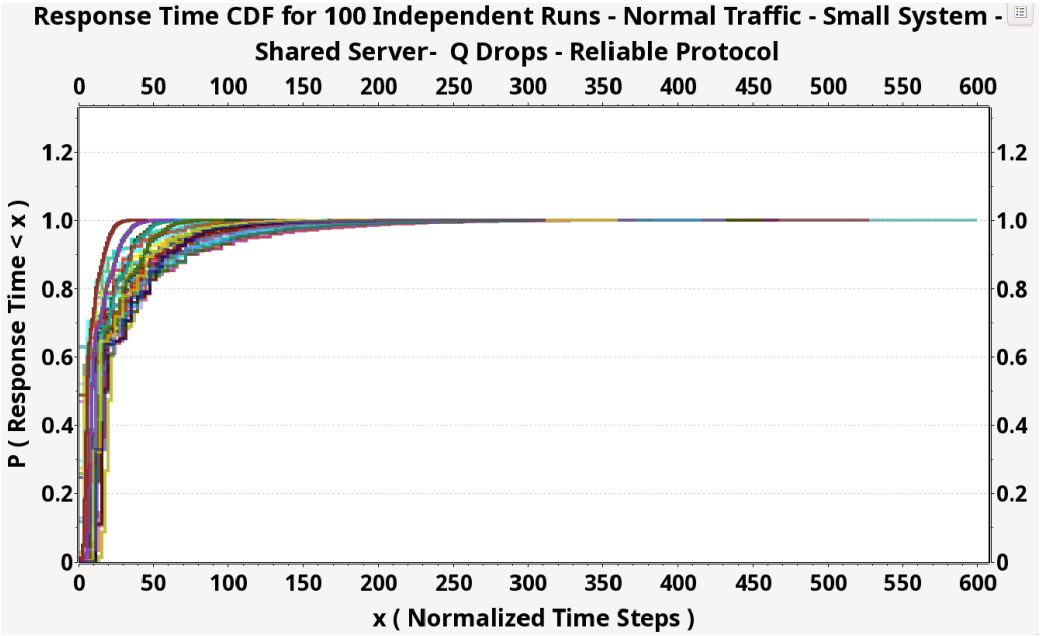
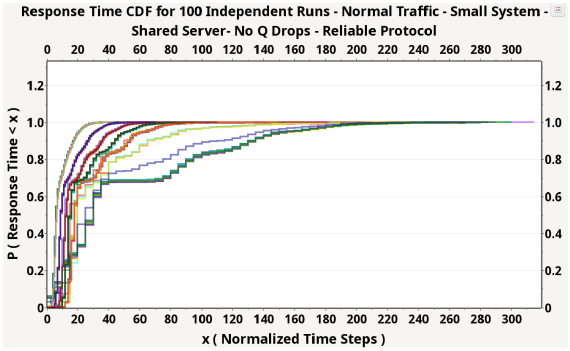


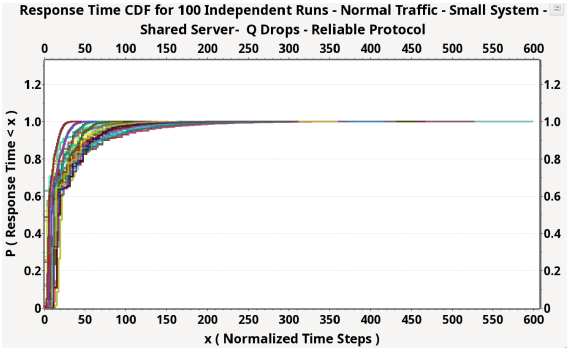
Figure 6.32: S4-E1 Application layer response time distributions. [S:P:F:F:TCP:S]

measure which violates the condition required by BET-compliance system results in a system with unpredictable behavior.

Based on the insights from Chapter 5, Experiment S4-E1 violates Software Engineering Design Rule 1,2 and Rule 3 by applying fair OS scheduling (summarized in 8.1.1). These violations result in producing two types of wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.32 validate the findings of the formal analysis in Chapter 5.



(a) S3-E1 ten hours of simulation. [S:P:F:I:TCP:S]



(b) S4-E1 ten hours of simulation. [S:P:F:F:TCP:S]

Figure 6.33: The effect of queue drops on performance predictability of shared servers using reliable protocols.

6.3.5.2 S4-E2: Small distributed software system exposed to normal traffic and employing non-reliable protocol with possible queue drop events [S:P:F:F:UDP:S]

Experiment S4-E2 mirrors Experiment S4-E1 but applies a non-reliable protocol, and is identical to S3-E2 except for the inclusion of possible queue drops. As evidenced by the following figures, queue drops increase the variability of response time cdfs particularly in servers experiencing higher loads, where the likelihood of queue drops is greater. Conversely, compared to S4-E1, removing the reliable protocol reduces system overload and significantly improves the tail of the distribution, decreasing from approximately 600 to around 150 normalized time steps. Figures 6.35 and 6.36 illustrate these effects, respectively. Table 6.33 offers a summary of the simulation parameters for S4-E2.

Table 6.33: S4-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared
Type of Queues	Finite with possible queue drops

Based on the insights from Chapter 5, Experiment S4-E2 violates Software Engineering Design Rule 1 and 2 in servers with higher loads and Rule 3 by applying

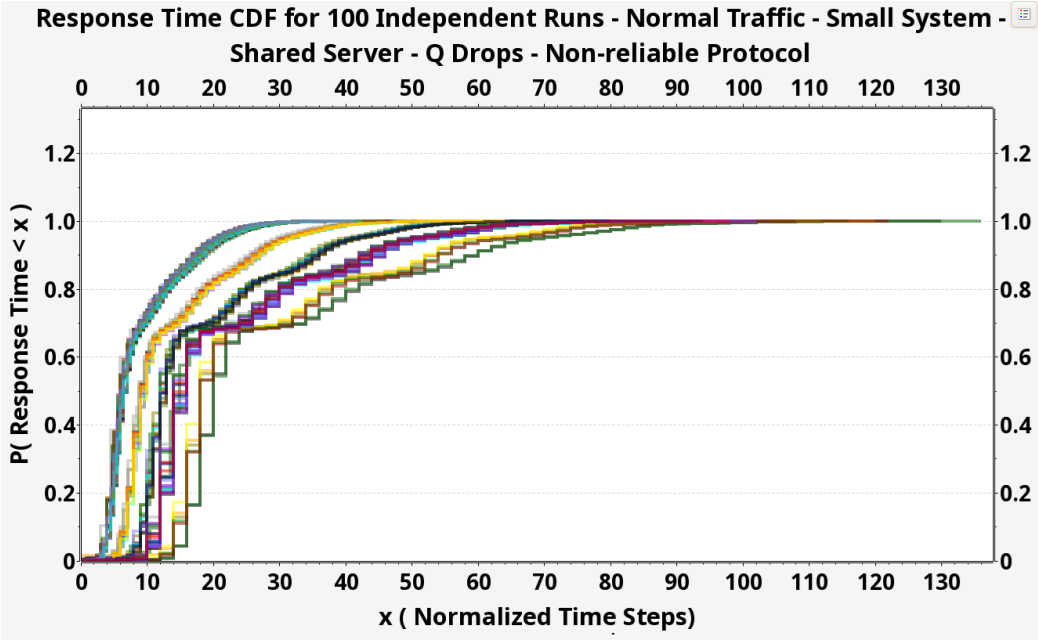
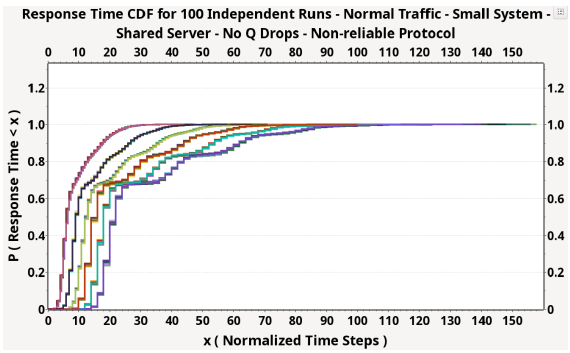
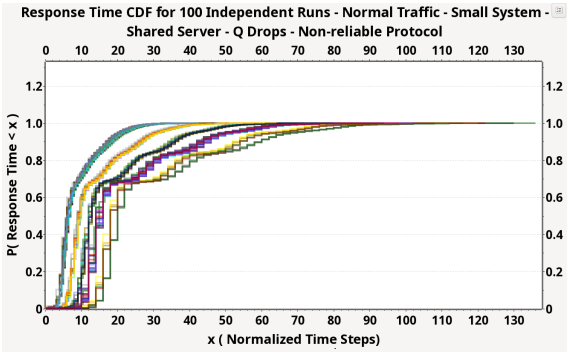


Figure 6.34: S4-E2 Application layer response time distributions. [S:P:F:F:UDP:S]



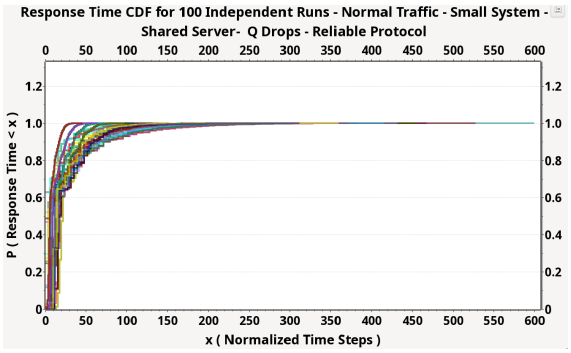
(a) S3-E2 ten hours of simulation. [S:P:F:I:UDP:S]



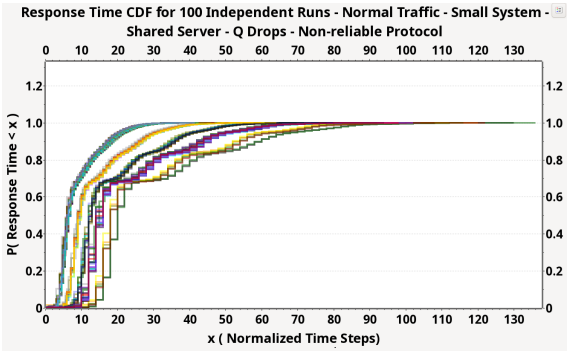
(b) S4-E2 ten hours of simulation. [S:P:F:F:UDP:S]

Figure 6.35: The effect of queue drops on performance predictability of shared servers.

fair OS scheduling (summarized in 8.1.1). These violations result in producing two types of wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.34 validate the findings of the formal analysis in Chapter 5.



(a) S4-E1 ten hours of simulation. [S:P:F:F:TCP:S]



(b) S4-E2 ten hours of simulation. [S:P:F:F:UDP:S]

Figure 6.36: The effect of reliable protocol on performance predictability of shared servers in the presence of queue drops.

6.3.5.3 S4-E3: Large distributed software system on shared servers exposed to normal traffic and employing reliable protocol with possible queue drop events [L:P:F:F:TCP:S]

Experiment S4-E3 mirrors Experiment S4-E1 but extends to a larger distributed system, as depicted in Figure 6.3, and also mirrors Experiment S3-E3, albeit with the inclusion of possible queue drops. Figure 6.37 presents the application-layer response time cdfs derived from 100 independent simulation runs of the exemplary distributed software system. Table 6.34 provides a summary of the simulation parameters for this experiment.

Table 6.34: S4-E3 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared
Type of Queues	Finite with possible queue drops

As depicted in Figure 6.37, queue drops introduce dispersion and variability, even in servers with lower loads, depending on the frequency of queue drops. They disrupt the cdfs, causing significant fluctuations. Additionally, they elongate the tails of the distributions, increasing worst-case scenarios from approximately 350 to around 750 normalized time steps. Figure 6.38 clearly illustrates the impact of queue drops on

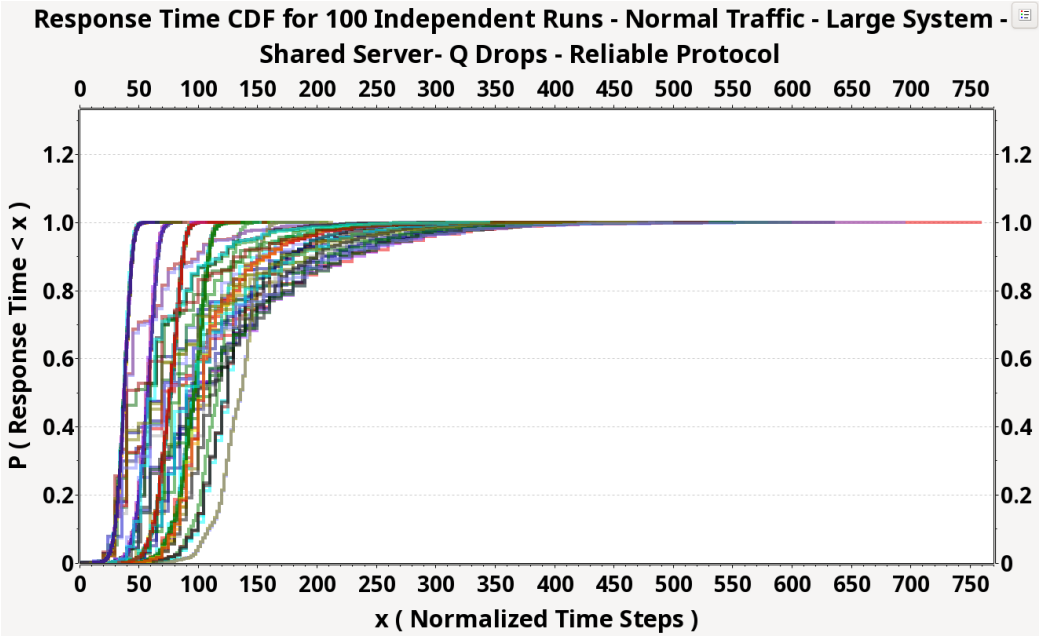
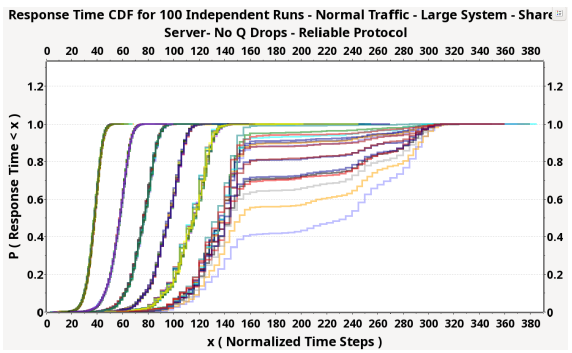
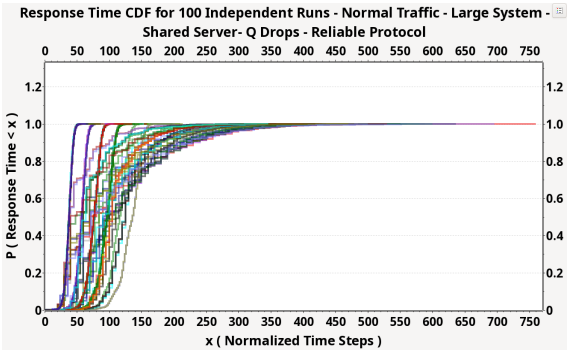


Figure 6.37: S4-E3 Application layer response time distributions. [L:P:F:F:TCP:S]

performance predictability. As anticipated from insights in Chapter 5, queue drops, combined with fair scheduling and reliable protocols, drive the system into a BET non-compliant state, rendering it unpredictable as they violate Software Engineering Design Rules 1,2,3 and Rule 4 for servers with high loads.



(a) S3-E3 ten hours of simulation. [L:P:F:I:TCP:S]



(b) S4-E3 ten hours of simulation. [L:P:F:F:TCP:S]

Figure 6.38: The effect of queue drops on performance predictability of shared servers for a larger distributed system.

6.3.5.4 S4-E4: Large distributed system exposed to normal traffic and employing non-reliable protocol with possible queue drop events [L:P:F:F:UDP:S]

Experiment S4-E4 is identical to S4-E3 but without the effect of a reliable protocol, and it also mirrors Experiment S3-E4 but with the addition of queue drops. As anticipated from the DST-driven insights and illustrated in Figure 6.40, queue drops introduce variability even among servers with the same load, making the behavior of the system unpredictable even under a known load. Conversely, comparing this experiment with the one applying a reliable protocol, we observe that removing the reliable protocol substantially reduces the tail of the response time distribution, decreasing worst-case scenarios from 750 to around 180 normalized time steps. It also reduces variability among servers with the same load as shown in Figure 6.41. According to insights from Chapter 5, queue drops and reliable protocols produce two different wandering sets of non-zero measure and it is expected that by removing one of them the system should become closer to a BET-compliant system. Table 6.35 offers a summary of the simulation parameters for S4-E4.

Table 6.35: S4-E4 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared
Type of Queues	Finite with possible queue drops

Based on the insights from Chapter 5, Experiment S4-E4 violates Software Engineering Design Rules 1 and 3 by allowing queue drops and applying a fair OS scheduling and using a non-reliable protocol (summarized in 8.1.1). This violation results in producing wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.39 validate the findings of the formal analysis in Chapter 5.

6.3.5.5 Scenario 4 Results Summary

The results of scenario 4 experiments are summarized in Table 6.36. Odd-numbered experiments with reliable protocols consistently demonstrate lower failure rates and

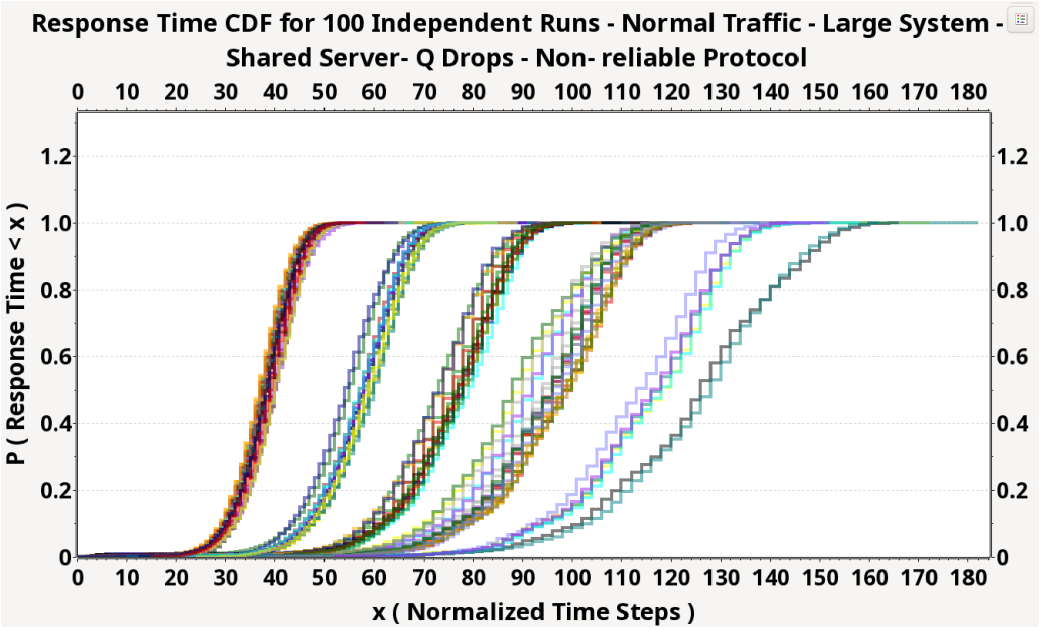
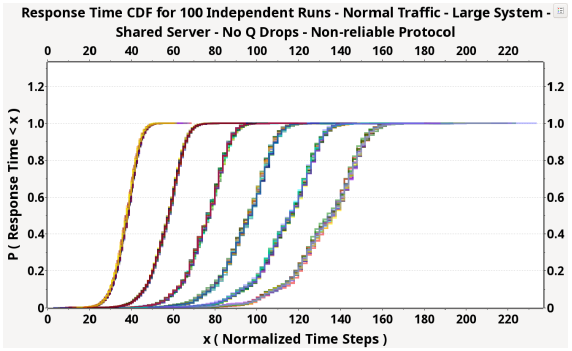
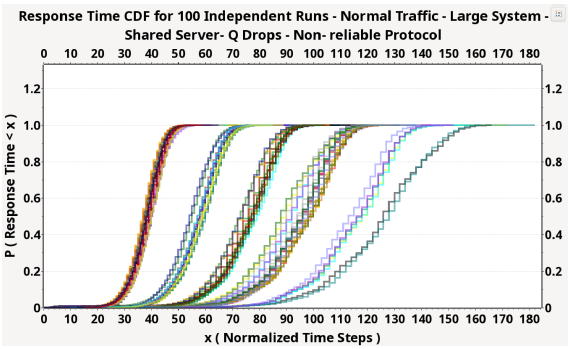


Figure 6.39: S4-E4 Application layer response time distributions. [L:P:F:F:UDP:S]



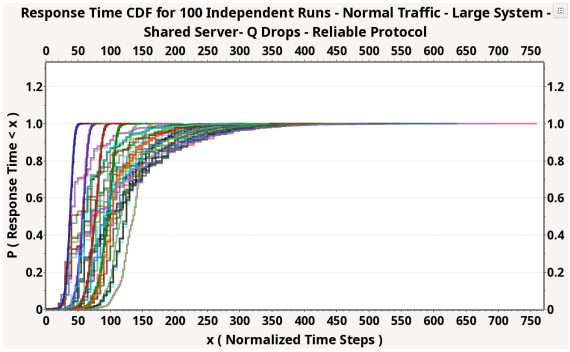
(a) S3-E4 ten hours of simulation. [L:P:F:I:UDP:S]



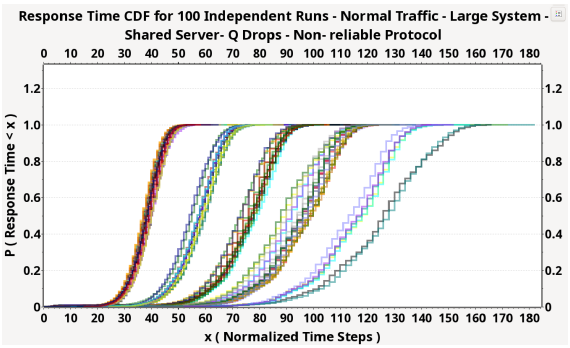
(b) S4-E4 ten hours of simulation. [L:P:F:F:UDP:S]

Figure 6.40: The effect of queue drops on performance predictability of shared servers for a larger distributed system.

typically exhibit broader response times, aligning with the insights derived from Chapter 5 by violating Software Engineering Design Rules 1,2 and 3. Conversely, even-numbered experiments employing non-reliable protocols show higher failure rates and generally narrower response time distributions, attributed to the absence of reliable protocols which means adherence to Design Rule 2, leading to removing one type of wandering sets of non-zero measure and therefore becoming closer to BET-compliant system.



(a) S4-E3 ten hours of simulation. [L:P:F:F:TCP:S]



(b) S4-E4 ten hours of simulation. [L:P:F:F:UDP:S]

Figure 6.41: The effect of reliable protocol on performance predictability of shared servers for a larger distributed system in the presence of queue drops.

Table 6.36: Scenario 4 experiments results comparison.

Ensemble Code	Experiment Code	Failure Rate [Min. , Max.]	Response Time (NTS) Percentiles [Min. , Max.]		
			90^{th}	95^{th}	99.99^{th}
S4-E1(100)	S:P:F:F:TCP:S	[0.0% , 0.08%]	[17.5 , 80.0]	[21.0 , 123.0]	[34.5 , 516.0]
S4-E2(100)	S:P:F:F:UDP:S	[0.0% , 24.0%]	[16.8 , 58]	[20.0 , 68.0]	[34.4 , 112.0]
S4-E3(100)	L:P:F:F:TCP:S	[0.0% , 7.2%]	[44.0 , 230.0]	[46.0 , 290.0]	[63.0 , 680.0]
S4-E4(100)	L:P:F:F:UDP:S	[0.0% , 42.0%]	[43.0 , 146.0]	[45.5 , 150.0]	[60.0 , 162.0]

Table 6.37 compares the simulations conducted in Scenario 4 according to their compliance with the four Software Engineering Design Rules formally derived from Chapter 5 analyses and summarized in Section8.1.1. Again "T" stands for "Transitional" state which means the system can cross over the boundary of Design Rule 4.

Table 6.37: Comparison of simulations conducted in Scenario 4 according to compliance with software engineering design rules.

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	Experiment Code	Software Eng. Design Rules				BET Compliant	cdfs
		No. 1	No. 2	No. 3	No. 4		
S4-E1 (100)	S:P:F:F:TCP:S	✗	✗	✗	*T*	✗	6.32
S4-E2 (100)	S:P:F:F:UDP:S	✗	✓	✗	*T*	✗	6.34
S4-E3 (100)	L:P:F:F:TCP:S	✗	✗	✗	*T*	✗	6.37
S4-E4 (100)	L:P:F:F:UDP:S	✗	✓	✗	*T*	✗	6.39

6.3.6 Scenario 5: System with shared servers servicing dynamic varying load with infinite queues

Scenario 5 further extends and enriches the Scenario 3 simulations by dynamically varying load on servers as it happens in practice, where each physical server hosts differing numbers of VMs coming and going and each VM has differing numbers of application-layer LDSS components running within it. This creates differing loads within the physical server as well as within the differing number of VMs and, hence, creates an increased level of complexity over that of Scenario 3. As can be seen in the following figures, this increased level of operational complexity being simulated, results in additional levels of variability in performance behavior over that seen in Scenario 3. As with the other Scenarios, this increased degree of variation between the cdfs is expected via the DST-derived performance predictability BET-compliance insights. All runs within this experiment exposed to identical statistical workloads, with no queue drop events.

6.3.6.1 S5-E1: Small distributed software system on shared servers exposed to normal traffic and employing reliable protocol with no queue drop events [S:P:F:I:TCP:D]

Experiment S5-E1 mirrors the S3-E1 experiment but introduces dynamically varying loads on servers. It holds significant importance as it collectively investigates the impacts of scheduling with load variability, and reliable protocol. Application-layer response time distributions (cdfs), derived from 100 independent simulation runs of the exemplary distributed software system, are presented in Figure 6.42. In the cdfs plot, it's evident that dynamically varying loads contribute to increased variation and spread among the response times. This outcome aligns with the BET-compliance insights from Chapter 5, where load variability generates wandering sets of non-zero measure, leading to a loss of BET-compliance and therefore unpredictability, even among servers experiencing the same load.Table 6.38 provides a summary of the simulation parameters for this experiment.

Table 6.38: S5-E1 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared with varying load
Type of Queues	Infinite

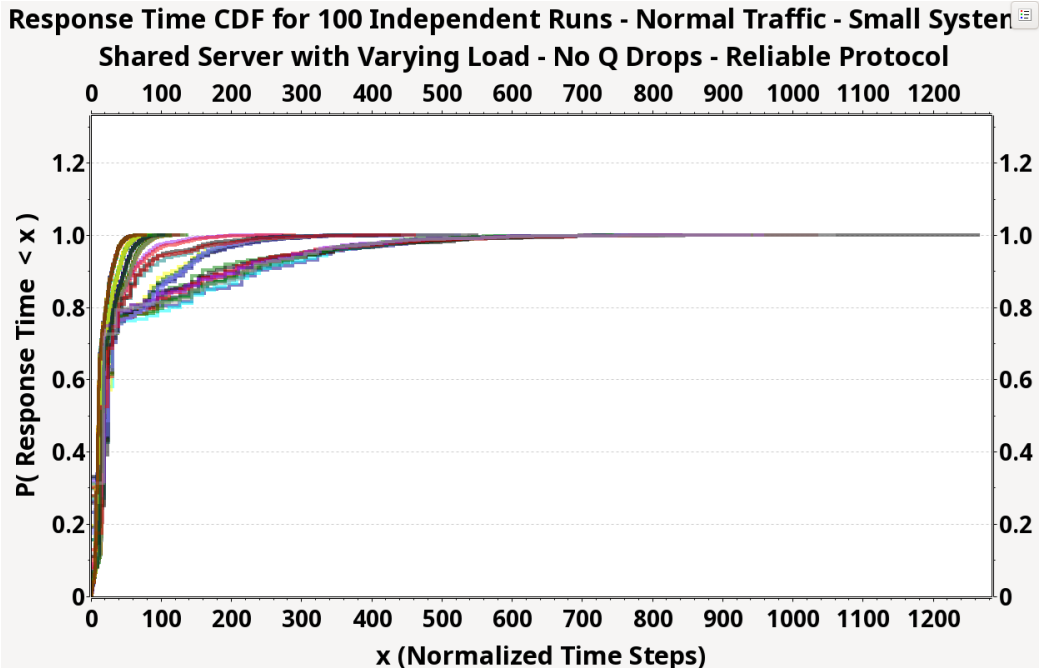
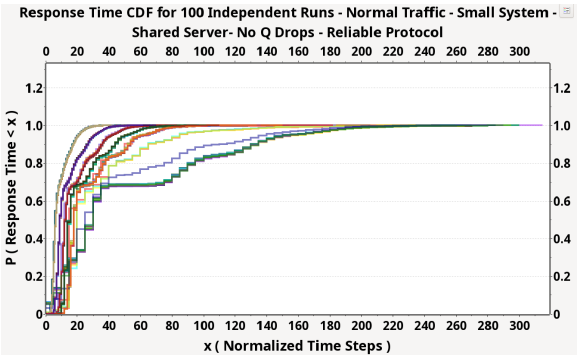


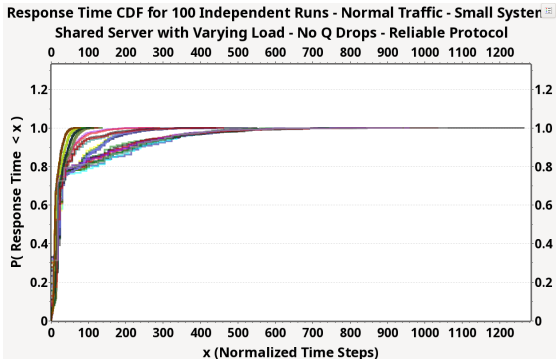
Figure 6.42: S5-E1 Application layer response time distributions. [S:P:F:I:TCP:D]

6.3.6.2 S5-E2: Small distributed software system exposed to normal traffic and employing non-reliable protocol with no queue drop events [S:P:F:I:UDP:D]

Experiment S5-E2 mirrors S5-E1 but with the application of a non-reliable protocol. The results confirm and validate the DST-derived insights for the performance behavior of software systems with cloud deployments where resource competition with several layers of scheduling and virtualization is present but here we eliminated the effects of queue drop events and reliable protocols. As we can see in Figure 6.44, not only the BET-compliance of system is broken but also we are moving between different distributions. In theory and practice, there is no limit for variety of these distributions. However, the tail of the distributions are much shorter in comparison with



(a) S3-E1 ten hours of simulation. [S:P:F:I:TCP:S]



(b) S5-E1 ten hours of simulation. [S:P:F:I:TCP:D]

Figure 6.43: The effect of varying load on performance predictability of shared servers using reliable protocols.

reliable protocol in S5-E1 as shown in Figure 6.45. On the other hand, as depicted in Figure 6.46, if we compare the result of this experiment with S3-E2 in which the server hosts a fixed number of VMs, we can see that in S3-E2 that the performance behavior is predictable for known scheduling regimes which validates the insights of Chapter 5 about scheduling effects.

Table 6.39 offers a summary of the simulation parameters for S5-E2.

Table 6.39: S5-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared with varying load
Type of Queues	Infinite

6.3.6.3 S5-E3: Large distributed software system on shared servers exposed to normal traffic and employing reliable protocol with no queue drop events [L:P:F:I:TCP:D]

Figure 6.47 presents the application-layer response time cdfs derived from 100 independent simulation runs of the larger exemplary distributed software system. All runs within this experiment exposed to identical statistical workloads. Experiment S5-E3 is similar to S5-E1 but for a larger distributed system. As shown in Figure

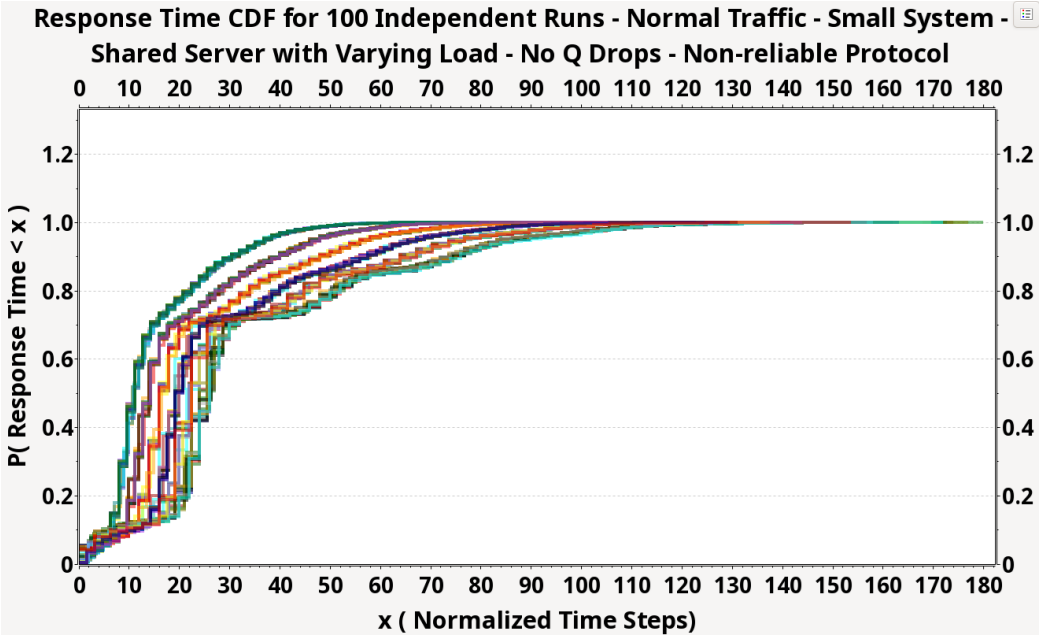
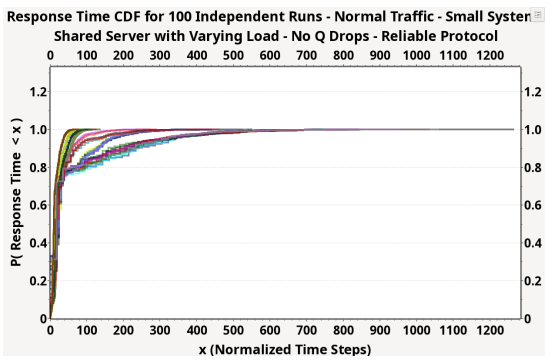
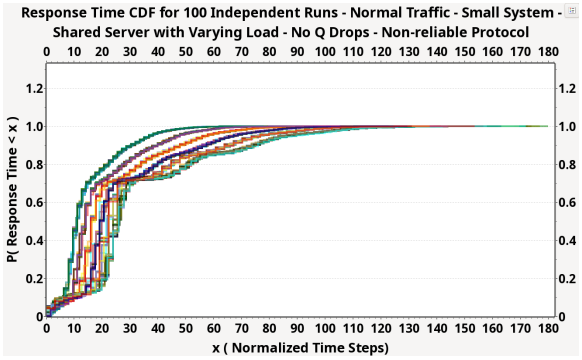


Figure 6.44: S5-E2 Application layer response time distributions. [S:P:F:I:UDP:D]



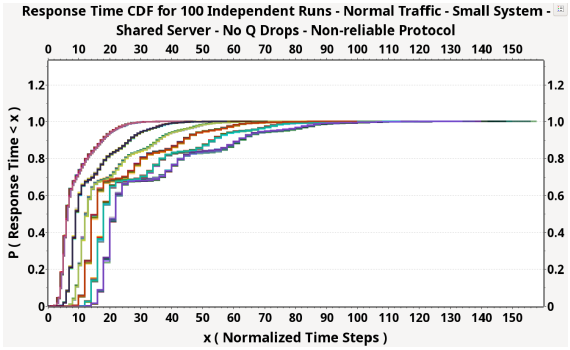
(a) S5-E1 ten hours of simulation. [S:P:F:I:TCP:D]



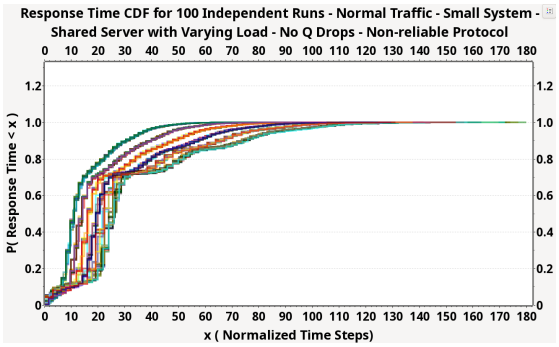
(b) S5-E2 ten hours of simulation. [S:P:F:I:UDP:D]

Figure 6.45: The effect of reliable protocol on performance predictability of shared servers with varying load.

6.48, and detailed in Table 6.42 we can see as expected that the larger system has a generally longer response times across almost all runs and there is more unpredictability for servers with heavier loads as they put more pressure on the already overloaded servers. Experiment S5-E3 is also mirroring S3-E3 but with introducing dynamic varying loads on the servers. The effects of varying load on the servers is shown in Figure 6.49. Table 6.40 provides a summary of the simulation parameters for this experiment.



(a) S3-E2 ten hours of simulation. [S:P:F:I:UDP:S]



(b) S5-E2 ten hours of simulation. [S:P:F:I:UDP:D]

Figure 6.46: The effect of varying load on performance predictability of shared servers.

Table 6.40: S5-E1 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared with varying load
Type of Queues	Infinite

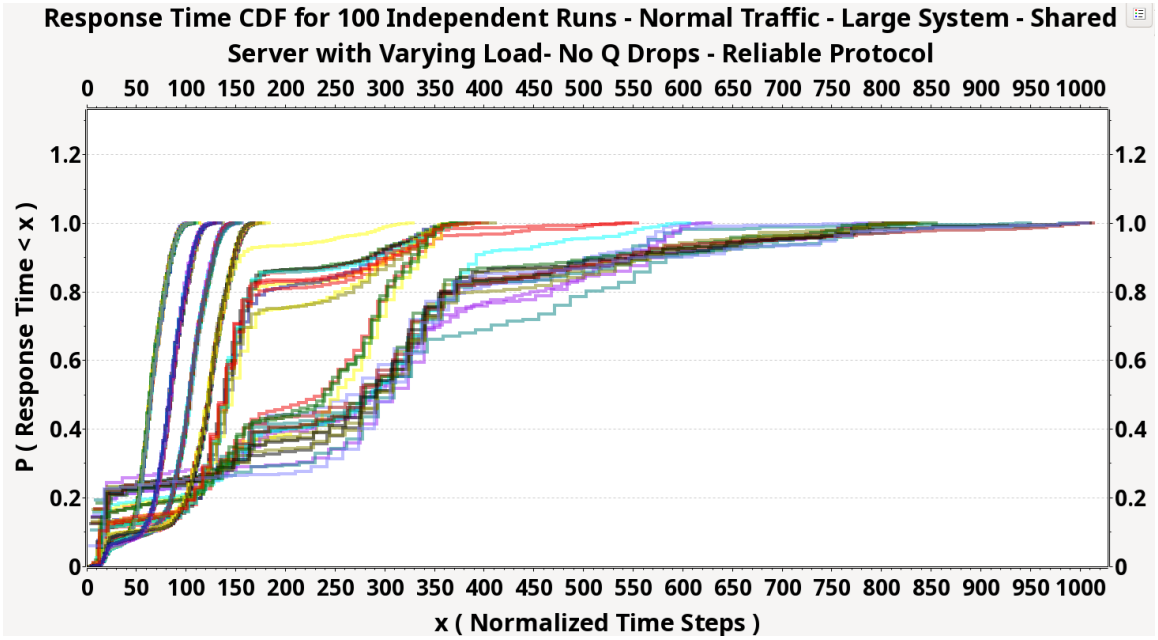
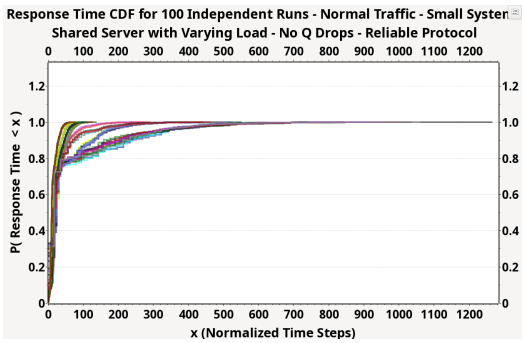
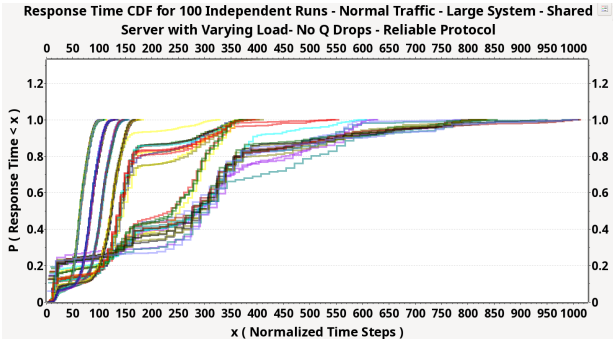


Figure 6.47: S5-E3 Application layer response time distributions. [L:P:F:I:TCP:D]

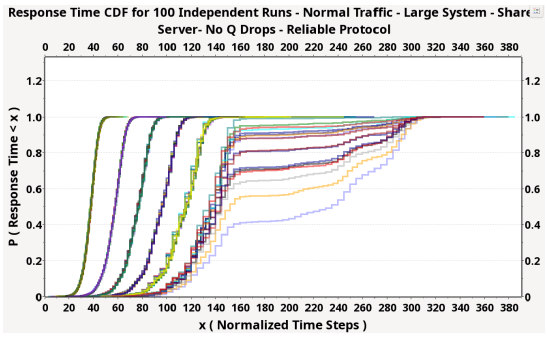


(a) S5-E1 ten hours of simulation. [S:P:F:I:TCP:D]

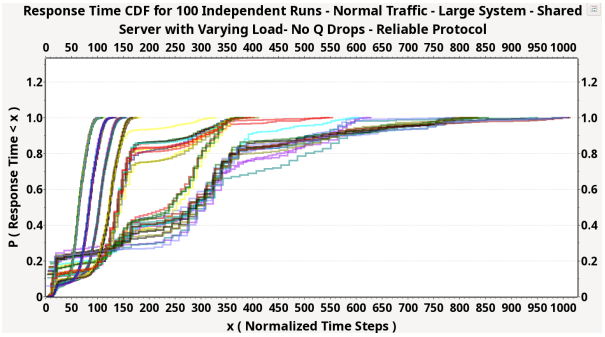


(b) S5-E3 ten hours of simulation. [L:P:F:I:TCP:D]

Figure 6.48: The effect of reliable protocol on performance predictability of shared servers with varying load.



(a) S3-E3 ten hours of simulation. [L:P:F:I:TCP:S]



(b) S5-E3 ten hours of simulation. [L:P:F:I:TCP:D]

Figure 6.49: The effect of varying load on performance predictability of shared servers for the larger distributed system.

Based on the insights from Chapter 5, Experiment S5-E3 violates Software Engineering Design Rules 2 and 3 by applying a reliable protocol and fair OS scheduling (summarized in 8.1.1). This violation results in producing wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.47 validate the findings of the formal analysis in Chapter 5. The modes visible in the cdfs are due to the six base loads on the simulated servers. In servers with higher loads, depending on how often the reliable protocol is triggered, we observe more variety in the cdfs.

6.3.6.4 S5-E4: Large distributed software system exposed to normal traffic and employing non-reliable protocol with no queue drop events [L:P:F:I:UDP:D]

Experiment S5-E4 mirrors S5-E3 experiment, but with the application of a non-reliable protocol. As we can see in Figure 6.51 using non-reliable protocol remove pressure form overloaded servers and results in generally better response times in those servers and a relatively less variations in response time distributions.

Experiment S5-E4 also mirrors S3-E4 experiment but with introducing dynamic varying load on the servers which as expected results in more variations in response time distributions as shown in Figure 6.52. Table 6.41 offers a summary of the simulation parameters for S5-E4.

Table 6.41: S5-E4 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared with varying load
Type of Queues	Infinite

Based on the insights from Chapter 5, Experiment S5-E4 violates Software Engineering Design Rule 3 by applying fair OS scheduling (summarized in 8.1.1). This violation results in breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.50 validate the findings of the formal analysis in Chapter 5. The modes visible in the cdfs are due to the six base loads on the simulated servers.

6.3.6.5 Scenario 5 Results Summary

Table 6.42 presents a comparison of the results from different experiments conducted under Scenario 5, focusing on failure rates and response time percentiles. All experiments report a failure rate of 0.0%, which was expected considering there were zero queue drops in this scenario. Experiments using the UDP protocol (S5-E2 and S5-E4) generally exhibit better response times across all percentiles compared to their TCP counterparts (S5-E1 and S5-E3). This is also expected because the reliable protocol crowds the queues by resending requests that didn't receive a response within

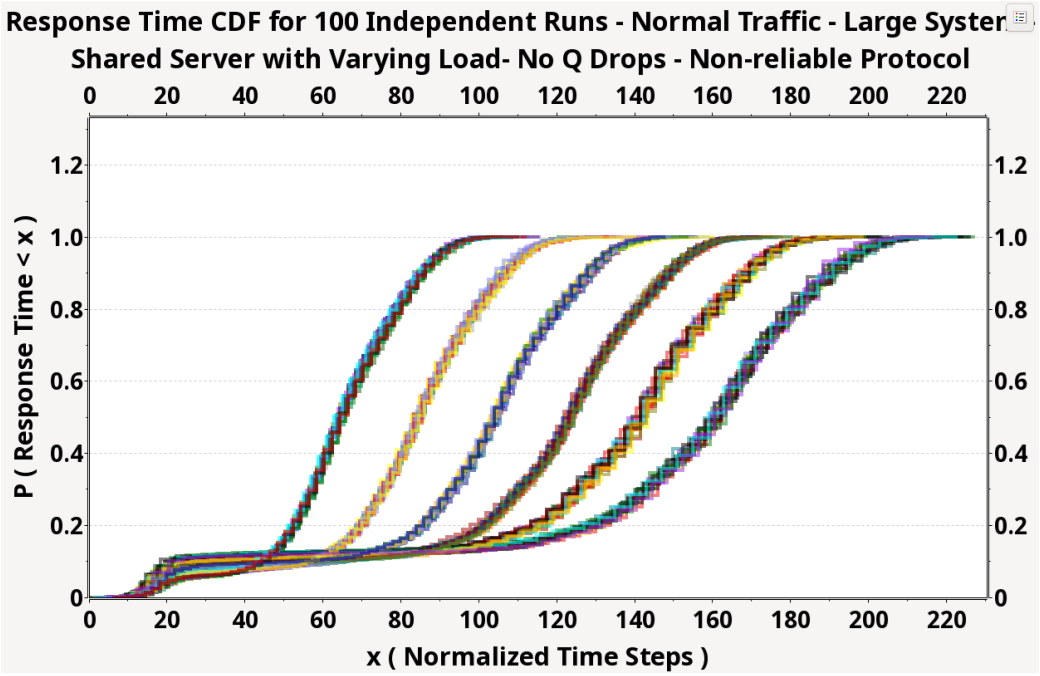
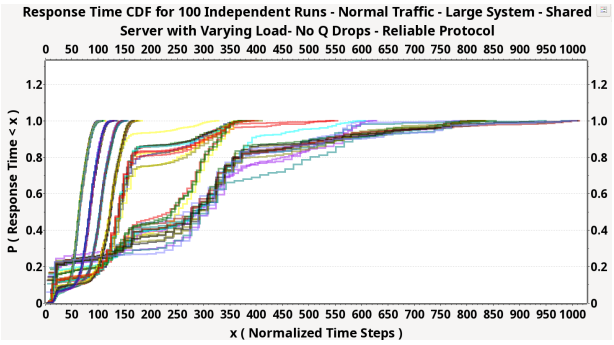
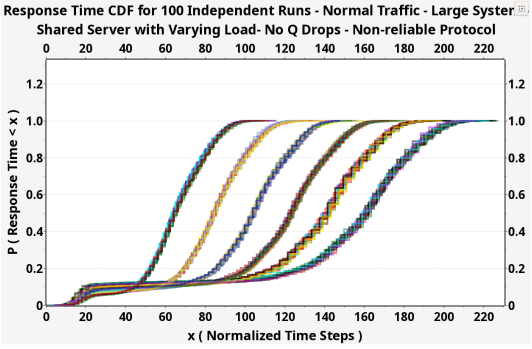


Figure 6.50: S5-E4 Application layer response time distributions. [L:P:F:I:UDP:D]



(a) S5-E3 ten hours of simulation. [L:P:F:I:TCP:D]

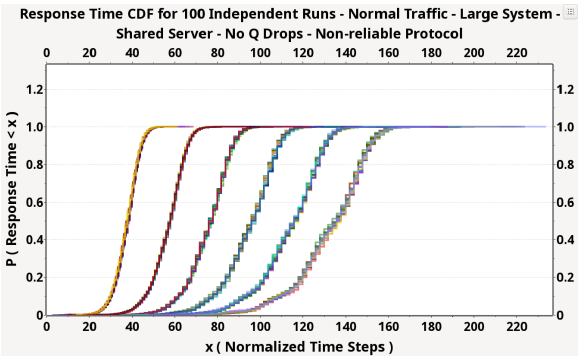


(b) S5-E4 ten hours of simulation. [L:P:F:I:UDP:D]

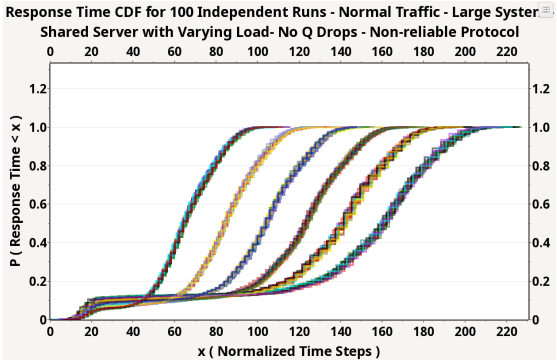
Figure 6.51: The effect of reliable protocol on performance predictability of shared servers with varying load.

a defined threshold. Large system simulations (S5-E3 and S5-E4) result in higher response times compared to smaller system (S5-E1 and S5-E2), reflecting the impact of system's size and load handling capacity on performance.

Table 6.43 compares the simulations conducted in Scenario 5 according to their compliance with the four Software Engineering Design Rules formally derived from Chapter 5 analyses and summarized in Section8.1.1.



(a) S3-E4 ten hours of simulation. [L:P:F:I:UDP:S]



(b) S5-E4 ten hours of simulation. [L:P:F:I:UDP:D]

Figure 6.52: The effect of varying load on performance predictability of shared servers for the larger distributed system.

Table 6.42: Scenario 5 experiments results comparison.

Ensemble Code	**Experiment Code**	**Failure Rate [Min. , Max.]**	**Response Time (NTS) Percentiles [Min. , Max.]**		
			90^{th}	**95^{th}**	**99.99^{th}**
S5-E1(100)	S:P:F:I:TCP:D	[0.0% , 0.0%]	[30.4 , 224.0]	[36.8 , 326.4]	[62.4 , 928.0]
S5-E2(100)	S:P:F:I:UDP:D	[0.0% , 0.0%]	[30.4 , 74.4]	[37.2 , 86.4]	[65.0 , 147.2]
S5-E3(100)	L:P:F:I:TCP:D	[0.0% , 0.0%]	[85.0 , 594.0]	[89.8 , 739.5]	[104.5 , 980.5]
S5-E4(100)	L:P:F:I:UDP:D	[0.0% , 0.0%]	[83.5 , 190.5]	[89.0 , 198.0]	[104.5 , 219.5]

Table 6.43: Comparison of simulations conducted in Scenario 5 according to compliance with software engineering design rules.

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	**Experiment Code**	**Software Eng. Design Rules**				**BET Compliant**	**cdfs**
		No. 1	**No. 2**	**No. 3**	**No. 4**		
S5-E1 (100)	S:P:F:I:TCP:D	✓	✗	✗	***T***	✗	6.42
S5-E2 (100)	S:P:F:I:UDP:D	✓	✓	✗	***T***	✗	6.44
S5-E3 (100)	L:P:F:I:TCP:D	✓	✗	✗	***T***	✗	6.47
S5-E4 (100)	L:P:F:I:UDP:D	✓	✓	✗	***T***	✗	6.50

6.3.7 Scenario 6: System with shared servers servicing dynamic varying load and finite queues

Scenario 6 is an extension of Scenario 5, which, in turn, enriches the simulations by limiting the size of queues which can lead to queue drop events. The results of experiments conducted in Scenario 6 demonstrate how queue drops combined with

increased operational complexity results in additional levels of variability compared to Scenario 5 and Scenario 4. As expected, this heightened complexity leads to variations in cdfs, in line with the performance predictability insights derived from the DST analysis, consistent with the patterns observed in the other scenarios.

6.3.7.1 S6-E1: Small distributed software system on shared servers exposed to normal traffic and employing reliable protocol with possible queue drop events [S:P:F:F:TCP:D]

The subsequent figures present the application-layer response time cdfs derived from 100 independent simulation runs of the exemplary small distributed software system. All runs within this experiment exposed to identical statistical workloads. Experiment S6-E1 is the closest one to the real world as it combines the effect of servers hosting variable number of VMs with the effects of queue drops and reliable protocol. As we can see in Figure 6.53, the cdfs are all over the place and the 99^{th} percentile is anywhere between 68 to 451 seconds which clearly shows the BET non-compliant behavior of the system even for a relatively simple, small cloud-deployed distributed system.

More specifically, Experiment S6-E1 violates all the Software Engineering Design Rules (summarized in 8.1.1) by allowing for queue drops, applying reliable protocol and using fair OS scheduling. These violations result in producing wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. The simulation results depicted in Figure 6.53 validate the findings of the formal analysis in Chapter 5. The modes visible in the cdfs of S6-E1 show more variety depending on how often the queue drop events occurred.

Table 6.44 provides a summary of the simulation parameters for this experiment.

Table 6.44: S6-E1 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared with varying load
Type of Queues	Finite with possible queue drops

Figure 6.54 compares the same experiment with and without queue drop events. In Figure 6.54a which represents the result of experiment with infinite queues, we

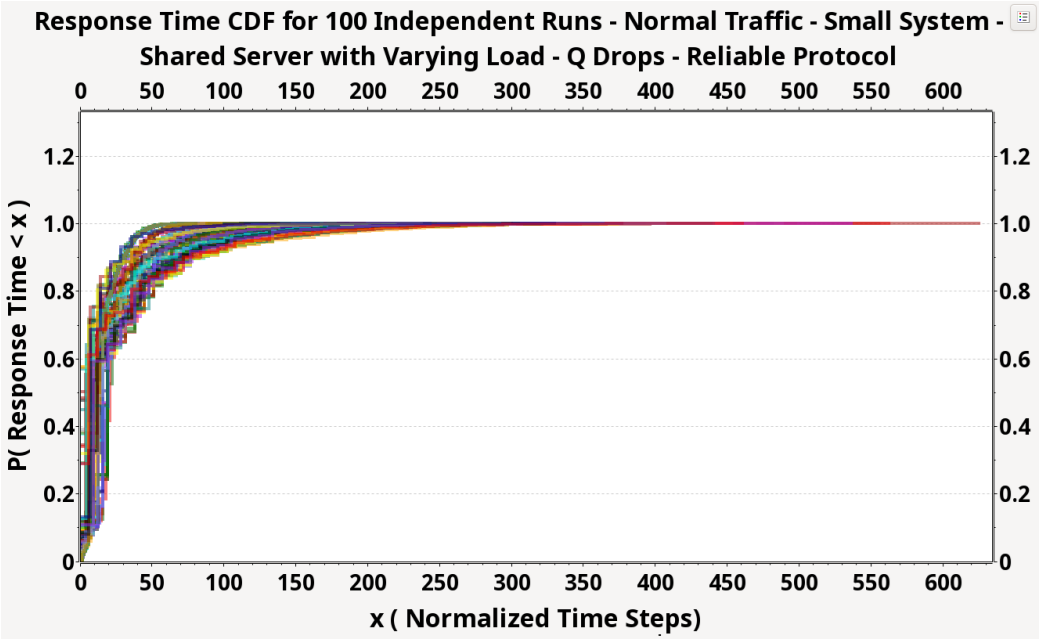
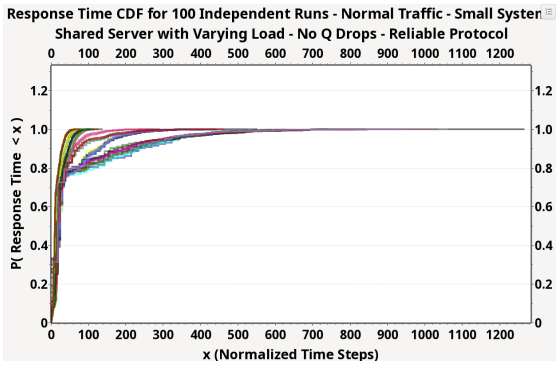
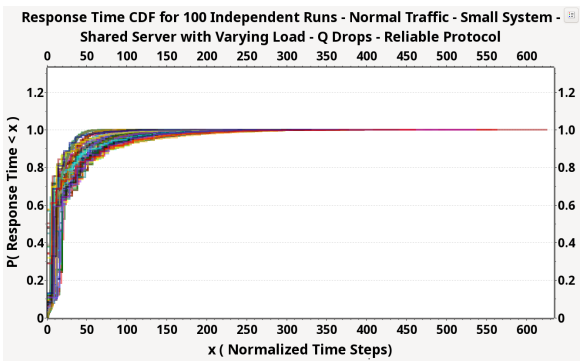


Figure 6.53: S6-E1 Application layer response time distributions. [S:P:F:F:TCP:D]

can see that there is a trade-off between no lost data and the significant delays under heavy load conditions. The infinite queue actually highlights the fair scheduling issue where there is no bound on execution time. Finite queues actually put a bound on the requests execution times and that's the reason we see a tighter bound on the response time but queue drops themselves leads to more variability in the response time distributions.



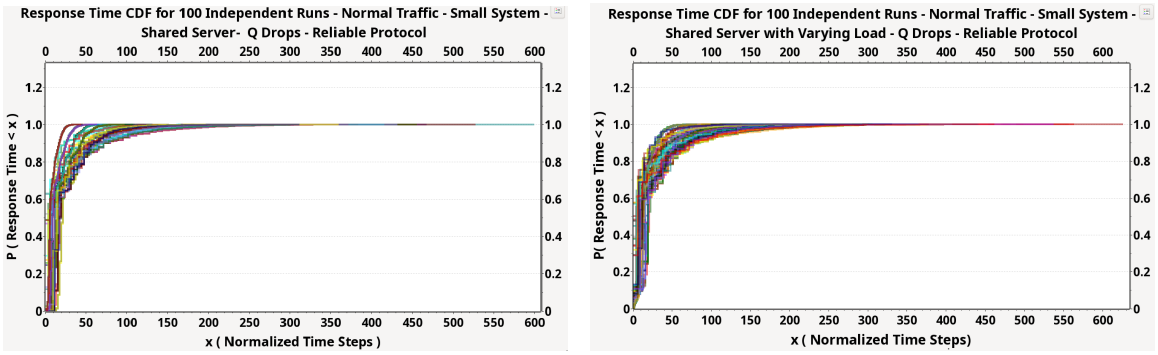
(a) S5-E1 ten hours of simulation. [S:P:F:I:TCP:D]



(b) S6-E1 ten hours of simulation. [S:P:F:F:TCP:D]

Figure 6.54: The effect of queue drops on performance predictability of shared servers with varying load.

Figure 6.55 demonstrated the comparison between the same experiment, but one with servers with fixed number of VMs in Scenario 4 and one with varying number of VMs in Scenario 6. As we expected from the Chapter 5 DST-driven BET-compliance insights, the results of these experiments show that the varying load leads to more variability in response time distributions.



(a) S4-E1 ten hours of simulation. [S:P:F:F:TCP:S]

(b) S6-E1 ten hours of simulation. [S:P:F:F:TCP:D]

Figure 6.55: The effect of varying load on performance predictability of shared servers.

6.3.7.2 S6-E2: Small distributed system exposed to normal traffic and employing non-reliable protocol with possible queue drop events [S:P:F:F:UDP:D]

Experiment S6-E2 mirrors Experiment 1, but with the application of a non-reliable protocol. Again, all runs within this experiment exposed to identical statistical workloads. Table 6.45 offers a summary of the simulation parameters for S6-E2.

Table 6.45: S6-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Small system in Figure 6.2
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared with varying load
Type of Queues	Finite with possible queue drops

As can be seen by comparing Figures 6.57a and Figure 6.57b, while noting the order of magnitude difference in x-axis scaling, the use of reliable protocols produces a

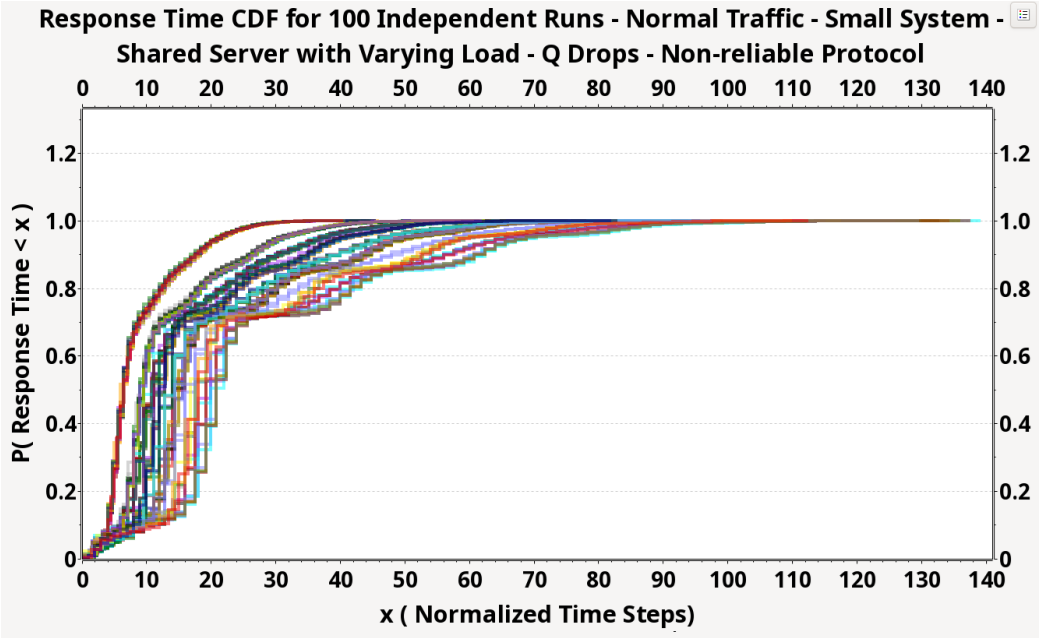
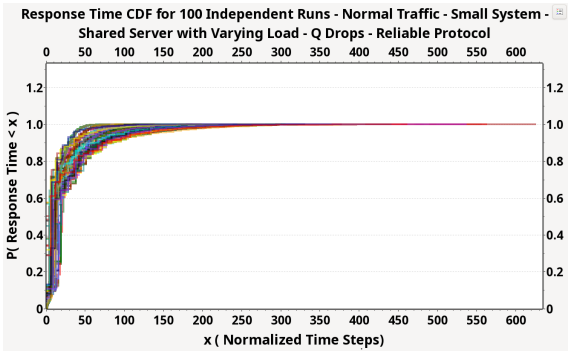


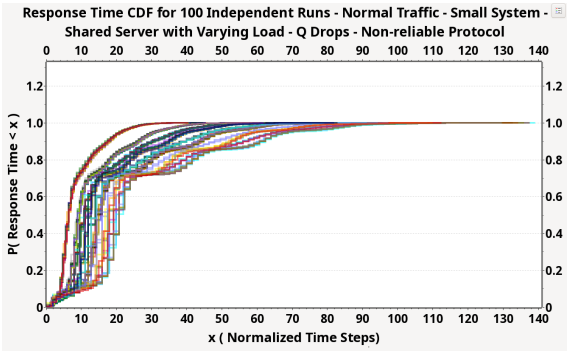
Figure 6.56: S6-E2 Application layer response time distributions. [S:P:F:F:UDP:D]

far broader set of response time cdfs than for the non-reliable protocol case. Variations in the Figure 6.57b observed cdfs are due to the impacts of standard cloud computing issues including number of executing VMs per modeled PH, background workload variations within the modeled VMs, VM scheduling and task swapping, as driven by the modeled OS-level fair scheduling processes, etc. These cloud regime issues though arise equally and are common to S6-E1 and S6-E2. Hence, they are not the cause of S6-E1's far wider statistical behavioral patterns, as seen in the Figure 6.57a.

More specifically, Experiment S6-E2 violates Software Engineering Design Rules 1 and 3 (summarized in 8.1.1) by allowing for queue drops and using fair OS scheduling. Additionally, it can break Design Rule 4 in servers with higher loads. These violations result in producing wandering sets of non-zero measure and breaking the measure-preserving condition, leading to a BET non-compliant system. However, S6-E2 adheres to Design Rule 2, which is why it is closer to a BET-compliant system compared to S6-E1, which breaks all the Design Rules. The simulation results depicted in Figure 6.56 validate the findings of the formal analysis in Chapter 5. The modes visible in the cdfs of S6-E2 show more variety depending on how often the queue drop events occurred.



(a) S6-E1 ten hours of simulation. [S:P:F:F:TCP:D]



(b) S6-E2 ten hours of simulation. [S:P:F:F:UDP:D]

Figure 6.57: The effect of reliable protocol on performance predictability of shared servers with varying load.

6.3.7.3 S6-E3: Large distributed software system on shared servers with varying load exposed to normal traffic and employing reliable protocol with possible queue drop events. [L:P:F:F:TCP:D]

The subsequent figures present the application-layer response time cdfs derived from 100 independent simulation runs of the exemplary large distributed software system. All runs within this experiment exposed to identical statistical workloads. Table 6.46 provides a summary of the simulation parameters for this experiment.

Table 6.46: S6-E3 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Reliable with timeout=32s and 8 re-send attempts
Type of Servers	Shared with varying load
Type of Queues	Finite with possible queue drops

Figure 6.58 shows noticeable variability in the cdfs across different runs suggesting inconsistency in performance resulted from broken BET-compliance conditions in the system.

Figure 7.15 shows the effect of dynamic fair scheduling and queue drops on the large system distributed system's performance. As we can see, these factors break the BET-compliance of the system and validates the result of Chapter 5's investigation on ergodicity in software-centric systems. More particularly, Experiment S6-E3 violates

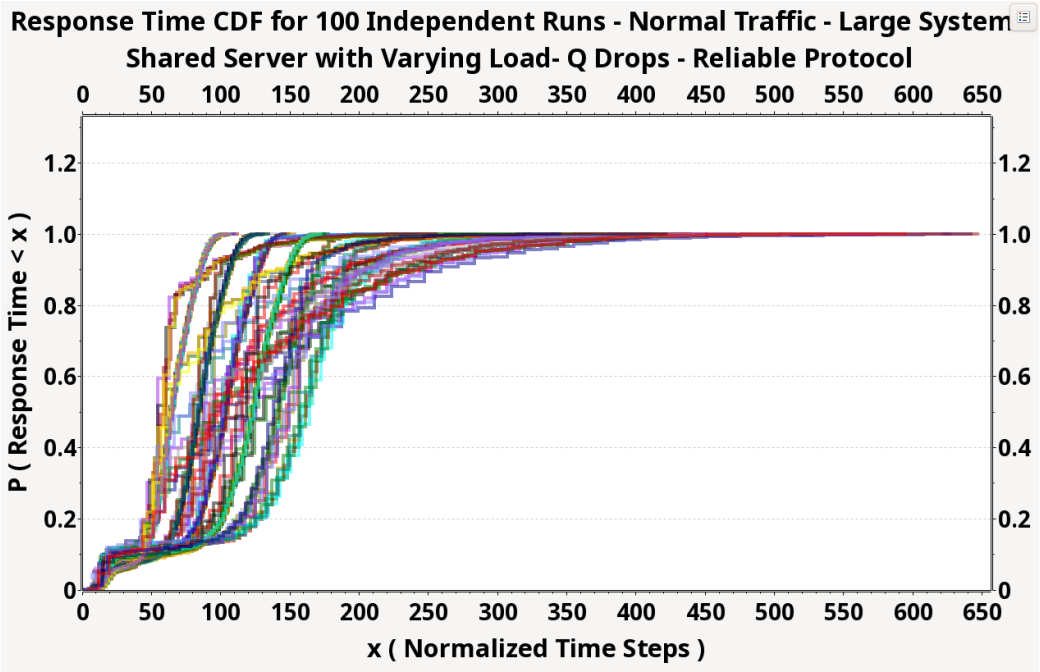
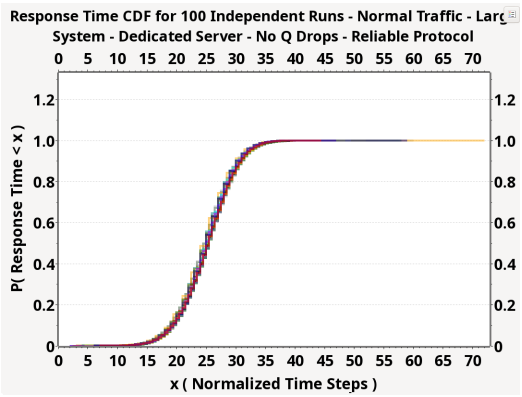


Figure 6.58: S6-E3 Application layer response time distributions. [L:P:F:F:TCP:D]

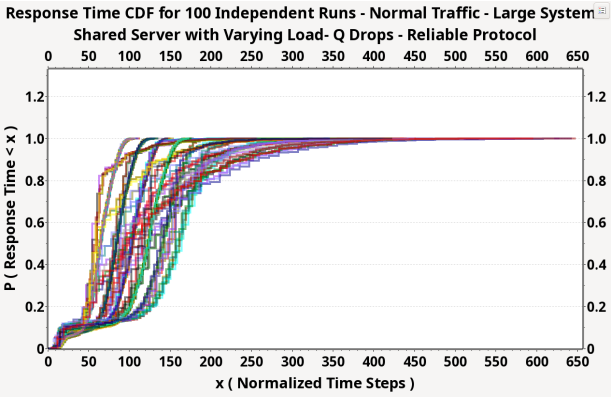
Software Engineering Design Rules (summarized in 8.1.1) 1, 2, and 3 by allowing for queue drops, applying a reliable protocol, and using fair OS scheduling. Moreover, it can break the Design Rule 4 in servers with higher loads. These violations result in producing two types of wandering sets of non-zero measure and breaking the measure-preserving condition in two ways, leading to a BET non-compliant system. The simulation results depicted in Figure 6.58 validate the findings of the formal analysis in Chapter 5. The modes were visible in the cdfs of S5-E3 are experiencing more variety depending on how often the queue drop events happened and how often the reliable protocol is triggered.

6.3.7.4 S6-E4: Large distributed system exposed to normal traffic and employing non-reliable protocol with possible queue drop events. [L:P:F:F:UDP:D]

Experiment S6-E4 mirrors S6-E3 experiment, but with the application of a non-reliable protocol. As we can see in the Figure 6.60, using non-reliable protocol reduces the response time significantly. The system still shows the BET non-compliant behavior by spreading the 100 runs 95^{th} percentile between 86 to 217 seconds compared to the base scenario as shown in Figure 6.61. More specifically, Experiment S6-E4



(a) S1-E3 ten hours of simulation. [L:P:Q:I:TCP:S]



(b) S6-E3 ten hours of simulation. [L:P:F:F:TCP:D]

Figure 6.59: The effect of queue drops,and shared servers with varying load on performance predictability for the larger distributed system in the presence of reliable protocol.

violates Software Engineering Design Rules 1 and 3 (summarized in 8.1.1) by allowing for queue drops and using fair OS scheduling. Additionally, it can break Design Rule 4 in servers with higher loads. These violations result in producing wandering sets of non-zero measure and breaking the measure-preserving condition in two ways, leading to a BET non-compliant system. The simulation results depicted in Figure 6.58 validate the findings of the formal analysis in Chapter 5. The modes visible in the cdfs of S6-E4 show more variety depending on how often the queue drop events occurred.

Table 6.47 offers a summary of the simulation parameters for S6-E2.

Table 6.47: S6-E2 specific simulation parameters.

Simulation Parameter	Value
Workload	Poisson
System Architecture	Large system in Figure 6.3
Communication Protocol	Non-reliable with timeout=128s
Type of Servers	Shared with varying load
Type of Queues	Finite with possible queue drops

Figure 6.62 and Figure 6.63 show the effects of queue drops and dynamic load on the performance behavior of the system respectively. As we can see in Figure 6.62, queue drops produce more variability in performance behavior validating the observations drawn from Theory 4 from Chapter 5.

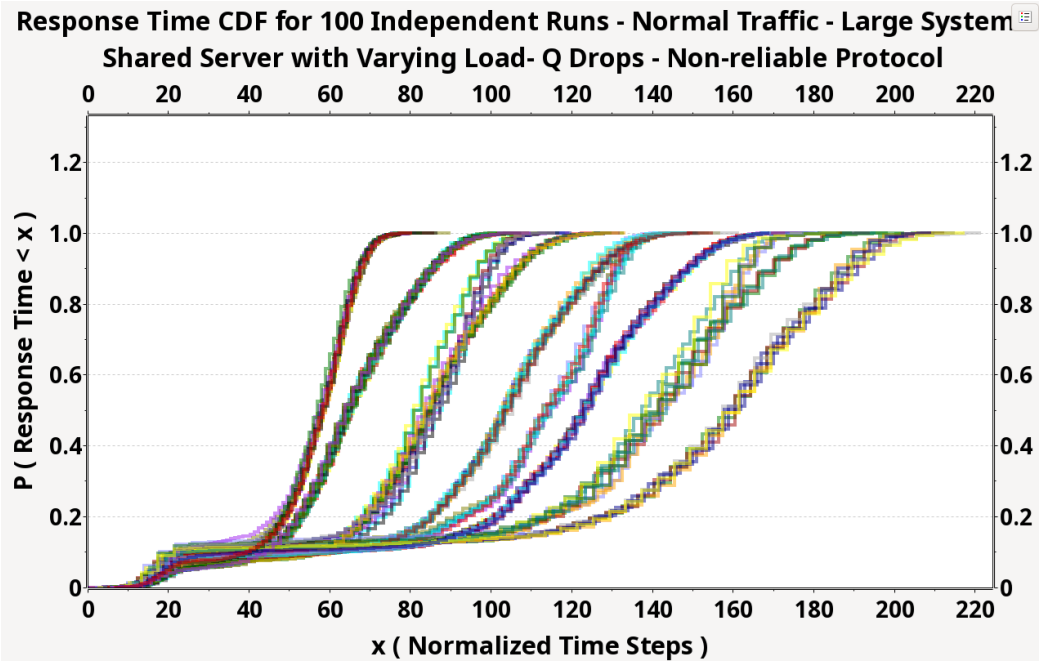
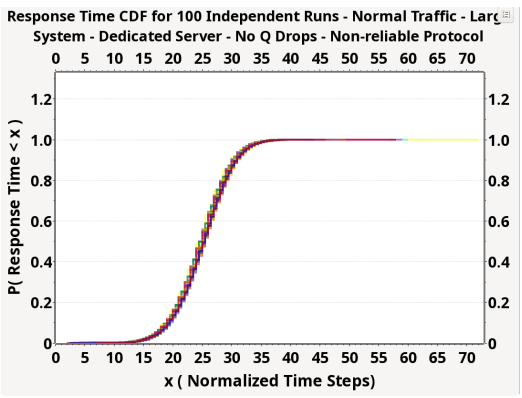
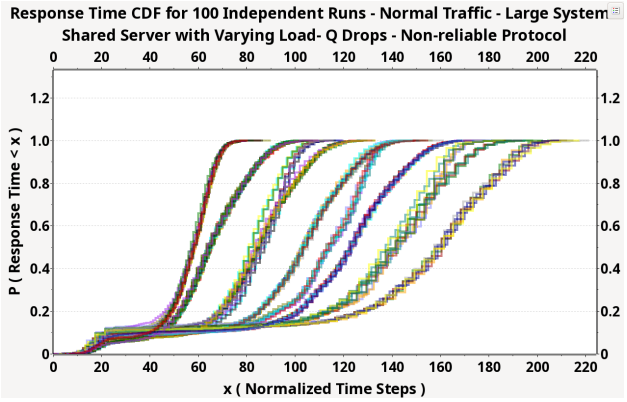


Figure 6.60: S6-E4 Application layer response time distributions. [L:P:F:F:UDP:D]



(a) S1-E4 ten hours of simulation. [L:P:Q:I:UDP:S]

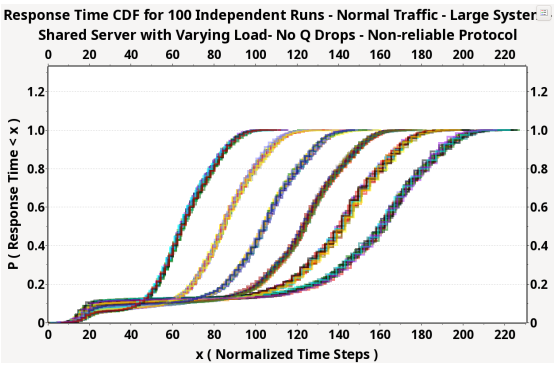


(b) S6-E4 ten hours of simulation. [L:P:F:F:UDP:D]

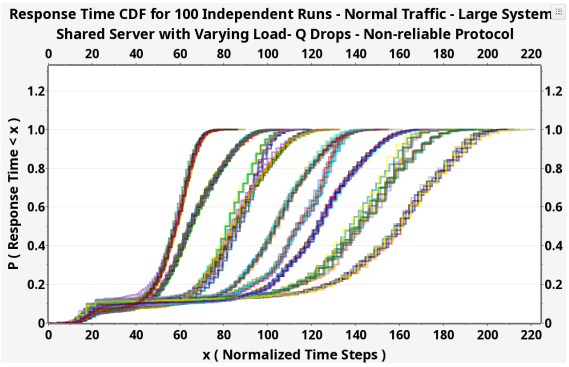
Figure 6.61: The effect of queue drops and shared servers with varying load on performance predictability of the larger system.

6.3.7.5 Scenario 6 Results Summary

The failure rates vary significantly across different experiments, with UDP protocols (S6-E2 and S6-E4) showing higher maximum failure rates compared to TCP protocols (S6-E1 and S6-E3) which is expected as the UDP protocol in our simulations drop the request if there is no response in 128 seconds from any internal server. UDP protocols

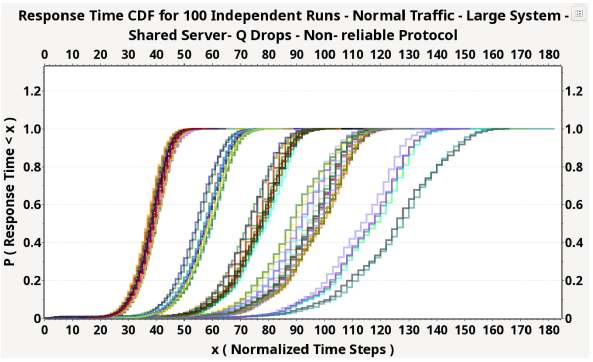


(a) S5-E4 ten hours of simulation. [L:P:F:I:UDP:D]

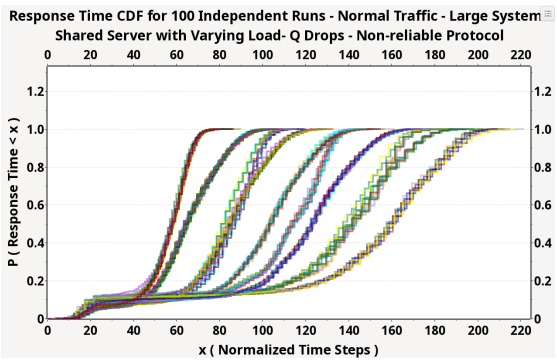


(b) S6-E4 ten hours of simulation. [L:P:F:F:UDP:D]

Figure 6.62: The effect of queue drops on performance predictability of shared servers with varying load for the larger system.



(a) S4-E4 ten hours of simulation. [L:P:F:F:UDP:S]



(b) S6-E4 ten hours of simulation. [L:P:F:F:UDP:D]

Figure 6.63: The effect of varying load on performance predictability of shared servers for the larger system.

generally have better response times at all percentiles but they are less reliable and can have higher failure rates. Large system simulations (S6-E3 and S6-E4) show higher response times compared to smaller systems (S6-E1 and S6-E2), reflecting the impact of system size on performance variations.

Table 6.49 compares the simulations conducted in Scenario 6 according to their compliance with the four Software Engineering Design Rules formally derived from Chapter 5 analyses and summarized in Section8.1.1.

Table 6.48: Scenario 6 experiments results comparison.

Ensemble Code	Experiment Code	Failure Rate [Min. , Max.]	Response Time (NTS) Percentiles [Min. , Max.]		
			90^{th}	95^{th}	99.99^{th}
S6-E1(100)	S:P:F:F:TCP:D	[0.0% , 0.7%]	[28.8 , 83.2]	[36.0 , 134.4]	[68.8 , 451.2]
S6-E2(100)	S:P:F:F:UDP:D	[0.0% , 17.6%]	[17.6 , 60.8]	[20.8 , 70.4]	[36.5 , 131.2]
S6-E3(100)	L:P:F:F:TCP:D	[0.0% , 4.4%]	[87.0 , 259.5]	[88.5 , 319.5]	[106.0 , 632.0]
S6-E4(100)	L:P:F:F:UDP:D	[0.0% , 39.0%]	[67.5 , 186.0]	[70.0 , 193.5]	[86.5 , 217.5]

Table 6.49: Comparison of simulations conducted in Scenario 6 according to compliance with software engineering design rules.

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	Experiment Code	Software Eng. Design Rules				BET Compliant	cdfs
		No. 1	No. 2	No. 3	No. 4		
S6-E1 (100)	S:P:F:F:TCP:D	✗	✗	✗	*T*	✗	6.53
S6-E2 (100)	S:P:F:F:UDP:D	✗	✓	✗	*T*	✗	6.56
S6-E3 (100)	L:P:F:F:TCP:D	✗	✗	✗	*T*	✗	6.58
S6-E4 (100)	L:P:F:F:UDP:D	✗	✓	✗	*T*	✗	6.60

6.4 Summary

In Chapter 6 we presented the detailed results of OMNet++ based Monte Carlo simulations that explore the statistical effects that factors such as queue drops, reliable protocols and different OS scheduling strategies have on software system run-time performance predictability. Real-world relevance is addressed by focusing the behaviors generated via the simulation of a real-world industry-held in-production Large-scale Distributed Software System (LDSS). The impacts of mentioned factors were contrasted through simulating the same LDSS under statistically identical incoming workloads and deployment regimes.

In this manner, it is shown that ongoing queue drops and their recovery actions through reliable protocols results in a substantial decrease in the LDSS's runtime performance predictability and cause increased violations of Chapter 5's Software Engineering Design Rules 1 and 2. This somewhat counter-intuitive result arises due to reliable protocols inherently generating additional workloads to be serviced into LDSS systems already experiencing the onsets of overload conditions. On the other hand, The presented work quantifies these impacts while highlighting the ease with

which the Chapter 5 Software Engineering Design Rules can be measured within operational systems and settings.

More particularly, outside of the explored quiescent cases, it was shown that the impacts of violating the Chapter 5 derived Software Engineering Design Rules are additive, with increasingly broad families of cdfs arising as more of the rules were triggered. Additionally, as per the Chapter 5 DST results and insights, it can be concluded that further increasing the size of the Monte Carlo ensembles will simply lead to the production of wider cdf families. were the size of these families is driven by the power set over the produced wandering sets and the power set over the possible variations in the event execution completion times. The DST of Chapter 5 highlights that neither of these BET non-compliant effects are bounded, a conclusion supported by the Chapter 6 experimental validations. Tables 6.50 and 6.51 show the summary results of all experiments conducted in Chapter 6.

Table 6.50: Experiments Results Comparison

Ensemble Code	Experiment Code	Failure Rate [Min.,Max.]	Response Time (NTS) Percentiles [Min.,Max.]		
			90^{th}	95^{th}	99.99^{th}
S1-E1(100)	S:P:Q:I:TCP:S	[0% , 0%]	[11.2 , 12.0]	[14.0 , 14.5]	[28.5 , 43.2]
S1-E2(100)	S:P:Q:I:UDP:S	[0% , 0%]	[11.2 , 12.0]	[13.6 , 14.0]	[27.0 , 45.0]
S1-E3(100)	L:P:Q:I:TCP:S	[0% , 0%]	[30.0 , 31.0]	[31.5 , 32.5]	[39.0 , 61.5]
S1-E4(100)	L:P:Q:I:UDP:S	[0% , 0%]	[30.0 , 31.0]	[31.5 , 32.5]	[39.0 , 63.0]
S1-E5(100)	S:B:Q:I:TCP:S	[0% , 0%]	[120.0 , 135.0]	[230.0 , 238.0]	[510.0 , 590.0]
S1-E6(100)	S:B:Q:I:UDP:S	[0% , 0%]	[110.0 , 114.0]	[122.0 , 131.0]	[205.0 , 270.0]
S1-E7(100)	L:B:Q:I:TCP:S	[0% , 0%]	[630.0 , 860.0]	[710.0 , 920.0]	[900.0 , 1260.0]
S1-E8(100)	L:B:Q:I:UDP:S	[0% , 0%]	[200.0 , 204.0]	[205.0 , 214.0]	[260.0 , 285.0]
S2-E1(100)	S:P:Q:F:TCP:S	[0.0% , 0.2%]	[4.0 , 11.0]	[6.0 , 33.5]	[33.0 , 354.0]
S2-E2(100)	S:P:Q:F:UDP:S	[0.1% , 1.6%]	[5.0 , 12.0]	[7.0 , 15.0]	[11.0 , 58.0]
S2-E3(100)	L:P:Q:F:TCP:S	[0.0% , 1.0%]	[20.0 , 50.6]	[29.4 , 100.0]	[146.0 , 497.0]
S2-E4(100)	L:P:Q:F:UDP:S	[0.0% , 4.0%]	[23.0 , 31.0]	[22.0 , 33.8]	[28.0 , 70.5]
S2-E5(100)	S:B:Q:F:TCP:S	[0.0% , 0.1%]	[140.2 , 190.0]	[170.5 , 240.0]	[242.5 , 470.3]
S2-E6(100)	S:B:Q:F:UDP:S	[0.2% , 15.7%]	[51.0 , 89.8]	[60.4 , 100.8]	[81.0 , 156.5]
S2-E7(100)	L:B:Q:F:TCP:S	[0.0% , 1.0%]	[560.5 , 770.0]	[590.0 , 800.3]	[625.0 , 932.8]
S2-E8(100)	L:B:Q:F:UDP:S	[0.5% , 64.0%]	[126.0 , 209.0]	[130.0 , 215.0]	[139.0 , 264.0]
S3-E1(100)	S:P:F:I:TCP:S	[0.0% , 0.0%]	[17.0 , 130.0]	[21.0 , 155.5]	[38.5 , 285.0]
S3-E2(100)	S:P:F:I:UDP:S	[0.0% , 0.0%]	[17.0 , 62.0]	[20.0 , 76.0]	[37.5 , 138.0]
S3-E3(100)	L:P:F:I:TCP:S	[0.0% , 0.0%]	[44.0 , 292.0]	[46.0 , 295.0]	[53.0 , 325.0]
S3-E4(100)	L:P:F:I:UDP:S	[0.0% , 0.0%]	[44.0 , 150.0]	[46.5 , 154.0]	[53.5 , 190.0]
S4-E1(100)	S:P:F:F:TCP:S	[0.0% , 0.08%]	[17.5 , 80.0]	[21.0 , 123.0]	[34.5 , 516.0]
S4-E2(100)	S:P:F:F:UDP:S	[0.0% , 24.0%]	[16.8 , 58.0]	[20.0 , 68.0]	[34.4 , 112.0]
S4-E3(100)	L:P:F:F:TCP:S	[0.0% , 7.2%]	[44.0 , 230.0]	[46.0 , 290.0]	[63.0 , 680.0]
S4-E4(100)	L:P:F:F:UDP:S	[0.0% , 42.0%]	[43.0 , 146.0]	[45.5 , 150.0]	[60.0 , 162.0]
S5-E1(100)	S:P:F:I:TCP:D	[0.0% , 0.0%]	[30.4 , 224.0]	[36.8 , 326.4]	[62.4 , 928.0]
S5-E2(100)	S:P:F:I:UDP:D	[0.0% , 0.0%]	[30.4 , 74.4]	[37.2 , 86.4]	[65.0 , 147.2]
S5-E3(100)	L:P:F:I:TCP:D	[0.0% , 0.0%]	[85.0 , 594.0]	[89.8 , 739.5]	[104.5 , 980.5]
S5-E4(100)	L:P:F:I:UDP:D	[0.0% , 0.0%]	[83.5 , 190.5]	[89.0 , 198.0]	[104.5 , 219.5]
S6-E1(100)	S:P:F:F:TCP:D	[0.0% , 0.7%]	[28.8 , 83.2]	[36.0 , 134.4]	[68.8 , 451.2]
S6-E2(100)	S:P:F:F:UDP:D	[0.0% , 17.6%]	[17.6 , 60.8]	[20.8 , 70.4]	[36.5 , 131.2]
S6-E3(100)	L:P:F:F:TCP:D	[0.0% , 4.4%]	[87.0 , 259.5]	[88.5 , 319.5]	[106.0 , 632.0]
S6-E4(100)	L:P:F:F:UDP:D	[0.0% , 39.0%]	[67.5 , 186.0]	[70.0 , 193.5]	[86.5 , 217.5]

Table 6.51: Comparison of simulation scenarios and experiments according to compliance with software engineering design rules (summarized in 8.1.1).

Conducted Simulations (Sorted from Simple to Complex)							
Ensemble Code	**Experiment Code**	**Software Eng. Design Rules**				**BET Compliant**	**cdfs**
		No. 1	**No. 2**	**No. 3**	**No. 4**		
S1-E1 (100)	S:P:Q:I:TCP:S	✓	✓	✓	✓	✓	6.8
S1-E2 (100)	S:P:Q:I:UDP:S	✓	✓	✓	✓	✓	6.9
S1-E3 (100)	L:P:Q:I:TCP:S	✓	✓	✓	✓	✓	6.10
S1-E4 (100)	L:P:Q:I:UDP:S	✓	✓	✓	✓	✓	6.11
S1-E5 (100)	S:B:Q:I:TCP:S	✓	✗	✓	✗	✗	6.12
S1-E6 (100)	S:B:Q:I:UDP:S	✓	✓	✓	✓	✓	6.13
S1-E7 (100)	L:B:Q:I:TCP:S	✓	✗	✓	✗	✗	6.14
S1-E8 (100)	L:B:Q:I:UDP:S	✓	✓	✓	✓	✓	6.15
S2-E1 (100)	S:P:Q:F:TCP:S	✗	✗	✓	✓	✗	6.16
S2-E2 (100)	S:P:Q:F:UDP:S	✗	✓	✓	✓	✗	6.17
S2-E3 (100)	L:P:Q:F:TCP:S	✗	✗	✓	✓	✗	6.18
S2-E4 (100)	L:P:Q:F:UDP:S	✗	✓	✓	✓	✗	6.19
S2-E5 (100)	S:B:Q:F:TCP:S	✗	✗	✓	✗	✗	6.20
S2-E6 (100)	S:B:Q:F:UDP:S	✗	✗	✓	✓	✗	6.21
S2-E7 (100)	L:B:Q:F:TCP:S	✗	✗	✓	✗	✗	6.22
S2-E8 (100)	L:B:Q:F:UDP:S	✗	✗	✓	✓	✗	6.23
S3-E1 (100)	S:P:F:I:TCP:S	✓	✗	✗	*T*	✗	6.25
S3-E2 (100)	S:P:F:I:UDP:S	✓	✓	✗	*T*	✗	6.27
S3-E3 (100)	L:P:F:I:TCP:S	✓	✗	✗	*T*	✗	6.28
S3-E4 (100)	L:P:F:I:UDP:S	✓	✓	✗	*T*	✗	6.30
S4-E1 (100)	S:P:F:F:TCP:S	✗	✗	✗	*T*	✗	6.32
S4-E2 (100)	S:P:F:F:UDP:S	✗	✓	✗	*T*	✗	6.34
S4-E3 (100)	L:P:F:F:TCP:S	✗	✗	✗	*T*	✗	6.37
S4-E4 (100)	L:P:F:F:UDP:S	✗	✓	✗	*T*	✗	6.39
S5-E1 (100)	S:P:F:I:TCP:D	✓	✗	✗	*T*	✗	6.42
S5-E2 (100)	S:P:F:I:UDP:D	✓	✓	✗	*T*	✗	6.44
S5-E3 (100)	L:P:F:I:TCP:D	✓	✗	✗	*T*	✗	6.47
S5-E4 (100)	L:P:F:I:UDP:D	✓	✓	✗	*T*	✗	6.50
S6-E1 (100)	S:P:F:F:TCP:D	✗	✗	✗	*T*	✗	6.53
S6-E2 (100)	S:P:F:F:UDP:D	✗	✓	✗	*T*	✗	6.56
S6-E3 (100)	L:P:F:F:TCP:D	✗	✗	✗	*T*	✗	6.58
S6-E4 (100)	L:P:F:F:UDP:D	✗	✓	✗	*T*	✗	6.60

Chapter 7

Assessing Industry Trends

The global cloud computing software industry actively evolves new approaches, frameworks, and deployment regimes to mitigate, address, and, ideally, improve software run-time performance predictability issues. This Chapter applies the insights and understandings of Chapters 5 and 6 to assess why the emerging and evolving industry trends of dynamic resource allocation and RTOS-based scheduling are not just "nice-to-have" features but, instead, necessary requirements to build software systems and solutions that behave predictably.

More directly, the complementary industry-developed approaches that explored are: (i) Apache Spark's dynamic resource allocation approaches to elastic scaling [278], and (ii) Xen's introduction of soft real-time OS scheduling as a configurable hypervisor instantiation option [279; 280]. We apply the Chapter 6 simulation framework to demonstrate and quantify how both approaches lead to statistically better-behaved systems, with these observed improvements driven by different and complementary mechanisms formally explained by the Software Engineering Design Rules and DST analysis of Chapter 5. Additionally, we show that although the industry (i) and (ii) methods substantially improve the system's observed behaviors, they are in themselves insufficient to bring the system back to being fully BET-compliant. This arises as violation in the Chapter 5 Software Engineering Design Rules still occur albeit at substantially lower rates than when (i) and (ii) are not applied.

More specifically, we highlight that these run-time performance predictability improvements were likely masked within prior simulation-based approaches, as Monte Carlo analyses are generally performed under the presumption that BET holds. Our analysis differs in that we do not assume BET holds. Instead, we provide the simulation results in terms of the empirical cdfs produced for each Monte Carlo ensem-

ble run, presenting the full cdf results rather than just the averaged cdf computed over the ensemble. This approach allows us to observe richer behavioral modalities and complexities than previously reported, leading to a better understanding of why the approaches (i) and (ii) produce improved run-time behavior predictability within cloud-deployed LDSSes and to more fully quantify these improvements. The question explored in this Chapter is therefore "By how much comparatively do these approaches improve predictability?", given the solutions do not have the same deployment costs or complexities.

Our contribution, relative to this Chapter, is not to claim that industry has not or is not actively evolving improvements in software system deployment regimes and run-time operations. Instead, it can be observed that such industry improvements are driven primarily through knowledgeable but largely ad hoc processes and methodologies. Our work seeks to place these industry-driven advancements onto the firm theoretical foundation provided via DST and our Chapter 5 analyses. Such deeper more formal analyses are required if modern large-scale software systems are to be brought fully back into being BET-compliant systems and, hence, systems producing statistically predictable run-time performance measures.

7.1 Assessing Apache Resource Scaling

Apache Spark dynamically allocates resources to scale with the workload and maintains fault tolerance through Resilient Distributed Datasets (RDDs), which can rebuild lost data. Furthermore, Spark's integration with cluster managers like Hadoop YARN, Apache Mesos, and Kubernetes enables flexible deployment and resource management [278]. Apache Storm is designed for real-time data processing, enabling rapid data ingestion and response through dynamic task distribution with stream grouping strategies. While Storm does not natively support dynamic resource scaling, it can integrate with systems like Apache Mesos for this functionality. Its fault-tolerant architecture and back-pressure mechanisms prevent system overloads [281].

In this Section, we use Monte Carlo simulation to quantitatively analyze the effects of resource overload prevention, as implemented by Apache, on the ergodicity and therefore BET-compliance state of the system through run-time performance predictability of LDSSes.

7.1.1 Monitor and Control Resource Utilization

Based on Chapter 5's DST-driven insights for BET-compliance, it has been established that on-going queue drops produce wandering sets of non-zero measure. This then renders the impacted run-time performance measures statistically unpredictable, in the sense of Eq. 5.14. This Chapter 5 insight was validated in Chapter 6's Scenario 2 where the effect of queue drops were isolated. As was highlighted in Chapter 5 and 6, under reliable protocols queue drop events then trigger event regenerations, which further load the system, which is already experiencing the onset of overloading. This further worsens the run-time predictability as both Software Engineering Design Rules 1 and 2 are not violated. Due to the hierarchical nature of QNs and modern computer hardware, software, and networks, queue drops can occur in any layer and/or part of the system or its deployment environment. For example, if the limits of available (or allocated) memory are reached and swapping is required, then under the QN model this equates to a queue drop event as the event's execution is stalled and then restarted at some future time [282].

To restore the system to BET-compliance and predictability, controlling queue drops becomes imperative as formally derived with the Chapter 5 analyses. The sotware industry employs various strategies for this purpose. One approach involves configuring limits on resources, especially memory. For example, on the Apache web server, administrators can set directives such as the number of server processes, threads per process, maximum simultaneous connections, and limits on the number of requests a server process will handle before being terminated and replaced. These settings can help manage memory leaks [282].

Another strategy is elastic resource management, which enables systems to automatically scale resources based on real-time demand. When scaling up with more resources available, load balancers distribute incoming requests evenly across all servers in a pool, preventing any single server from bearing an excessive load. This prevents servers from reaching their maximum capacity, thereby reducing the likelihood of queue drops. For instance, Apache can be configured to operate in a load-balanced environment using modules like "mod_proxy_balancer", distributing incoming requests across a pool of back-end servers to optimize resource usage and improve response times. Moreover, platforms offering serverless computing services, such as AWS Lambda, Google Cloud Functions, and Azure Functions, employ this

technique, automatically managing infrastructure resources by scaling up or down instantly based on request volume.

7.1.2 Simulation Details

The impacts of an Apache Spark-like dynamic resource allocation methodology were assessed as follows. An ensemble of 100 Monte Carlo simulation runs was produced, via our Chapter 6 cloud deployment simulation framework, where each run spanned 10 hours of simulated time and involved the exemplary LDSS in 6.2 executing concurrently with a selected set of 12 to 18 other cloud clients so as to model modern commercial cloud environments. Our modeled LDSS remains active throughout the 10 hour period processing in-coming statistically generated workloads which model the real-Poisson workloads of the original industry-held system explained in Chapter 6. In total, 10 workload generators were simulated to produce the cloud-incoming workloads to be processed by the modeled systems. All incoming traffic was modelled as sessionless "*http*" requests in accordance with the real-world industry-held system. The incoming workloads themselves were BET-compliant and, hence, were not the source of any non-BET compliant issues observed.

Similar to Chapter 6 simulations, session time, measured as the total response time, was recorded as the run-time performance measure of interest for each "*http*" request to subsequent LDSS response interaction. The think time between user requests and the idle time between sessions follow Poisson distribution with parameters detailed in Table 7.1.

Table 7.1: Simulation Parameters

Simulation Parameter	**Value**
Performance measure	Response time
No. of independent runs	100
Idle time distribution	$p(x) = \lambda_{idle} e^{-\lambda_{idle} x}; \lambda_{idle} = 1/100$ ms
Think time distribution	$p(x) = \lambda_{think} e^{-\lambda_{think} x}$; $\lambda_{think} = 1/10$ ms
Service time per server	$p(x) = \lambda_{service} e^{-\lambda_{service} x}$; $\lambda_{service} = 1/300$ ms
Simulation time	36×10^6 ms (10h)
Queue length	Infinite

7.1.3 Fair Scheduling

Figure 7.1 displays the cdfs for response times across 100 independent runs of our exemplar LDSS running on a hypervisor employing fair OS scheduling without any load balancing as a benchmark to assess load balancing scenarios. These runs were subjected to a statistically identical Poisson workloads on a single server, which was experiencing time-varying loads, as per commercial cloud regimes. The spread of the cdf curves illustrates the inconsistency in response times occurring between Monte Carlo similar runs. This aligns with the observations reported in the original industry study of [4] as well as the theory and validation insights of Chapters 5 and 6.. This variability in performance, as reflected by the distinct cdf trajectories highlights a core lack of LDSS run-time predictability. Moreover, a real-world deployed system will not simply be described by one of the potential cdfs within this family of possible cdfs. Instead, it would actively transition across this family of possible cdfs stochastically as was executing, highlighting a far more complex performance predictability problem and challenge.

Furthermore, Figure 7.1 highlights the combined impact of fair OS scheduling, reliable protocol usage, and physical hosts (PHs) workloads on an LDSS ability to provide consistent and predictable run-time performance. More particularly, a single server is shown to be able to efficiently handle a certain amount of requests but then performance predictability quickly degrades as load increases. This is an expected result based on Chapter 5 DST-based insights, but Figure 7.1 quantifies the impacts. Importantly, the server transition threshold would generally not be a priori computable except in quite trivial workload cases.

Moving to the right with Figure 7.1, the increasingly wider spread and more gradual ascent within the observed cdfs is indicative of server resource saturation. This suggests that there are points where the server was unable to efficiently handle the load, leading to increasing and more variable response times. We should note that concurrently with load increases would have been the triggering of reliable protocol actions to recover from load-induced "dropped" events. This would have had the effect of placing even more load on a server that is already entering into an overloaded condition driving the further decreases in the run-time performance predictability, as denoted by observable spread in the produced response time cdfs.

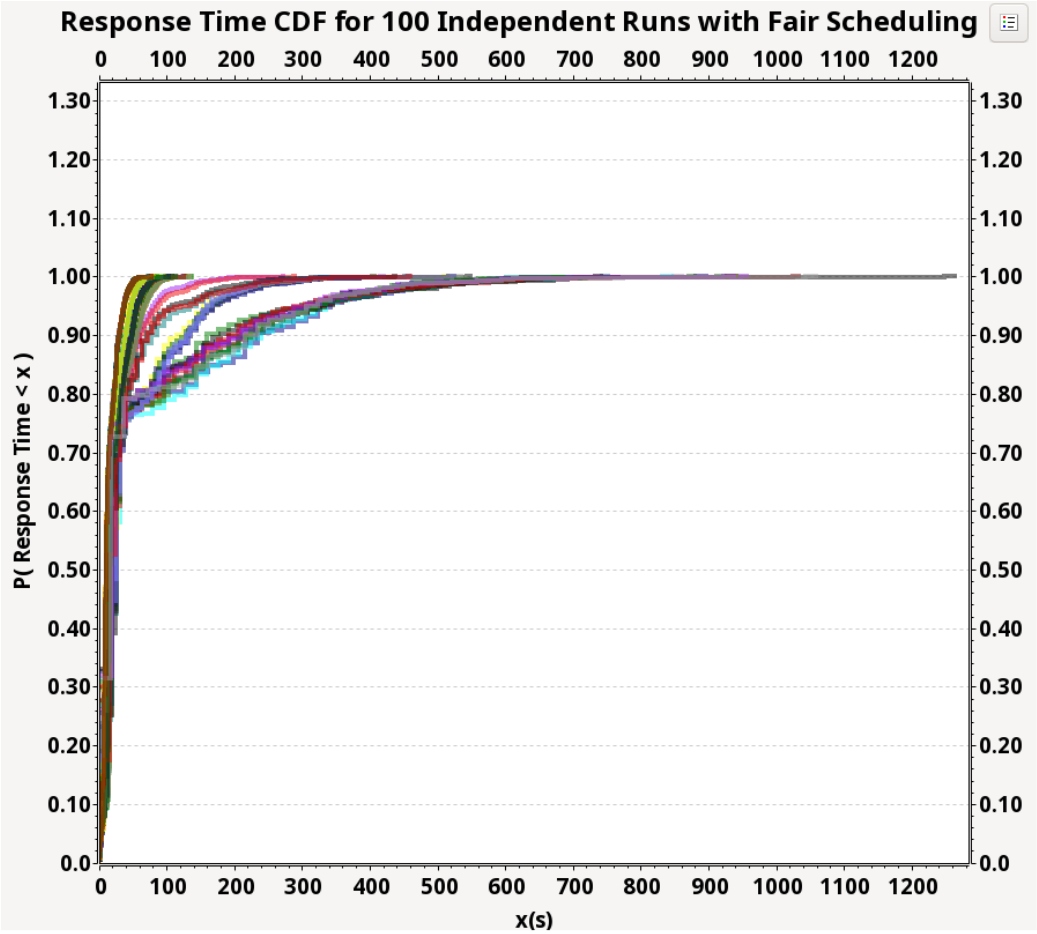
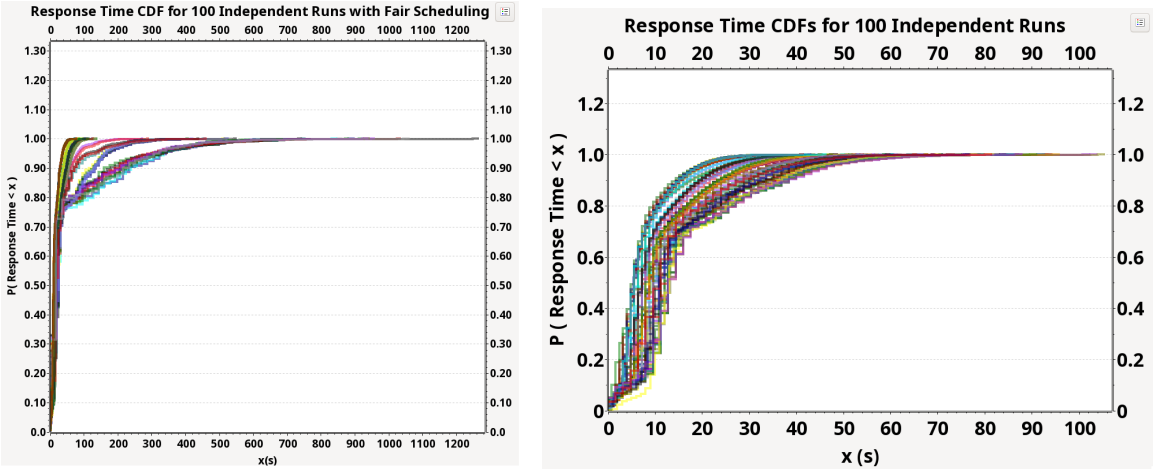


Figure 7.1: Response time cdfs for 100 independent runs of the system for the fair OS scheduling regime.

7.1.4 Fair OS Scheduling with Apache-like Resource Scaling

In this experiment, we investigated the effects of incremental server scaling on system performance, modeling the approach implemented within Apache Spark and Storm. We systematically introduced additional servers in pairs, starting with a two-server configuration and progressing to configurations with four, six, eight, and finally ten servers. At each stage, we simulated identical workloads, mirroring the conditions of the initial single-server environment. By employing fair OS scheduling strategies alongside Apache-like auto-scaling capabilities, we observed the impacts on LDSS response times and their cdf consistencies. Gradually scaling was used to monitor when and where breakpoints in run-time performance predictability occurred, allowing us to understand both the benefits and overheads of increasing server counts.

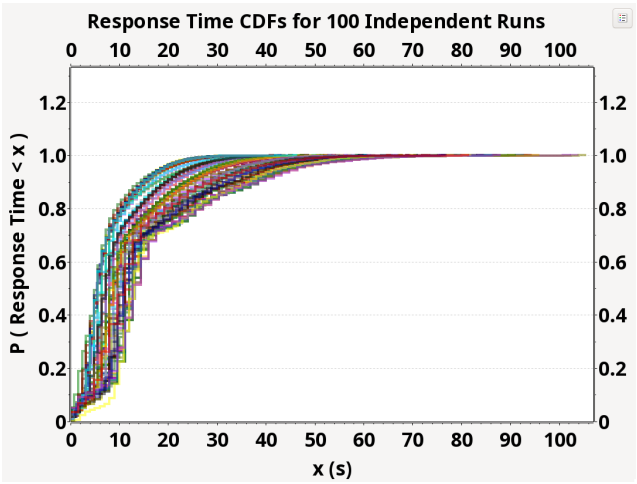
As shown in Figure 7.2, transitioning from a single server to a two-server system shows a significant improvement in response time cdf behaviors as a consequence of server overloads being prevented. This, in turn, reduces the need for reliable protocol actions, thereby creating a smaller family of generated cdfs. These improvements continue as the cloud environment is extended to four and six servers as depicted in Figure 7.3 but clearly with diminishing returns. The produced cdf families show increasingly less impacts with the cdfs' upper regions. As the available resources are further increased to eight and to ten servers more nuanced effects arise. The cdf families remain steep initially, indicating most events (or customers) receive prompt service but the upper region of the cdfs highlight more marginal improvements, particularly relative to the higher costs that would accumulate through these higher server count usages. Costing in commercial cloud deployments is of critical industry concern. Hence, the trade-offs and breakpoints between LDSS increased performance predictability and the resulting cost increases is a critical issue. Figure 7.4 highlights that this issue is a LDSS and cloud-regime dependent mapping, which in turn would necessitate significant of real-world LDSS and deployment regime instrumentation and analysis to effectively and optimally manage the available resources.



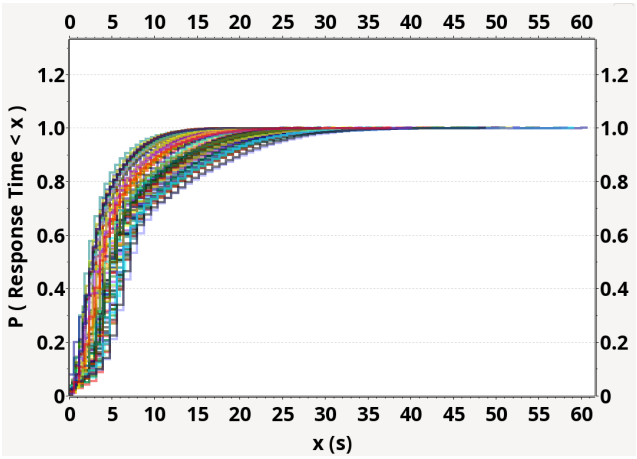
(a) No active resource management with one server does all the jobs.

(b) Apache-like resource management with 2 servers

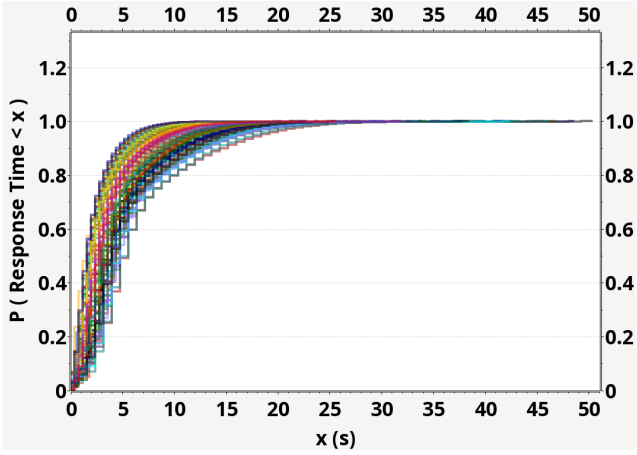
Figure 7.2: Response time cdfs comparison for 100 independent LDSS runs applying fair OS scheduling with one server and 2-servers Apache-like active resource management. Note the x-axis scaling range decreases from 1300s to 100s.



(a) Apache-like resource management with 2 servers

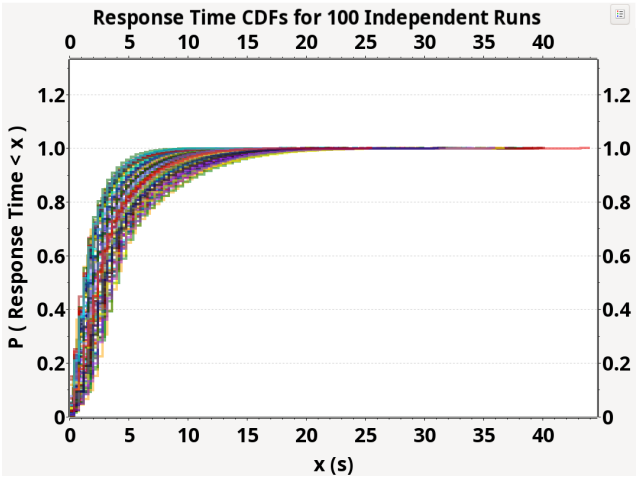


(b) Apache-like resource management with 4 servers

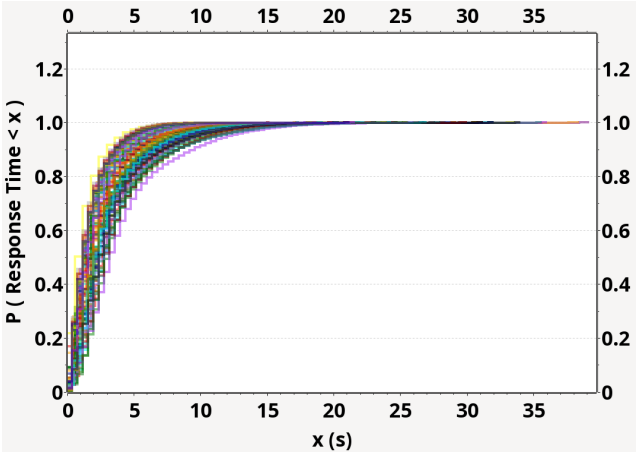


(c) Apache-like resource management with 6 servers

Figure 7.3: Response time cdfs for 100 independent LDSS runs applying Apache-like active resource management against varying server counts of 2, 4 and 6. Note the x-axis scaling range decreases from subplots (a) to (c).



(a) Apache-likeresource management with 8 servers



(b) Apache-likeresource management with 10 servers

Figure 7.4: Response time cdfs for 100 independent LDSS runs applying Apache-like active resource management against varying server counts of 8 and 10.

To better assess the Figure 7.4 results, the 90^{th} percentile response times threshold is used as this provides a good comparative measure of performance under load and, specifically, for the region of the cdf that would lead to SLO/SLA violations. These results are shown in Table 7.2. In the one-server configuration, the 90^{th} percentile range is from $[31.1, 300.0]$ seconds with the longest response time at approximately 300 seconds. This highlights a significant tail latency, resulting from peak load conditions when the server itself becomes a bottleneck due to overload.

As more servers are added, the 90^{th} percentile range contracts. With two servers, the range is $[11.8, 40.0]$ seconds, with an upper response time of approximately 100 seconds, a significant improvement over the one-server case. Doubling the count to four

Table 7.2: Performance improvement rate comparison.

Simulation	**Improvement Rate [Min., Max.]**	**90^{th} Percentile Response Time(s) [Min., Max.]**
1-Server	-	[31.1 , 300.0]
2-Server	[61.2% , 86.6%]	[11.8 , 40.0]
4-Server	[77.4% , 92.9%]	[6.7 , 21.2]
6-Server	[83.8% , 95.0%]	[4.6 , 14.9]
8-Server	[87.0% , 96.2%]	[3.5 , 11.3]
10-Server	[87.0% , 96.7%]	[3.5 , 9.8]

servers produces more modest improvements, with the range reducing to $[6.7, 21.2]$ seconds and an upper response time of approximately 60 seconds, indicating a doubling of cost with slightly less than a doubling of performance. At six servers, the range is $[4.6, 14.9]$ seconds with an upper response time of approximately 55 seconds. Now, costs have tripled compared to the two-server case, but this increased cost is not reflected in the observed performance improvements. With eight to ten servers, the range tightens to $[3.5, 11.3]$ and $[3.5, 9.8]$ seconds, respectively, with upper response times of approximately 44 and 39 seconds. Clearly, the respective $4x$ to $5x$ cost increase over the two-server case is not worth the observed performance improvement, even though the produced family of cdfs has significantly less variability and, therefore, the LDSS has increased run-time performance predictability.

These comparisons quantify the reductions in upper bound of response times with increasing resourcing and the ensuing marginal gains at higher resourcing levels. Clearly, the impacts are non-linear, LDSS dependent, and cloud regime dependent. This highlights the need to carefully and in detail instrument real-world LDSS so as to be able to extract the information required to achieve optimal cost-effective resource allocation and performance predictability levels. Moreover, it is clear that simple linear optimizations will not suffice, given the modeled and studied scenario is far simpler than that of actual real-world cloud-deployed LDSS contexts.

7.2 Assessing Modern Cloud Auto-scaling

Modern cloud computing regimes have enabled an alternative approach of auto-scaling so as to dynamically adjusting the number of resources (such as virtual machines or containers) allocated to an application as it workload changes. Several modern cloud

frameworks and technologies incorporate auto-scaling for efficient resource management because the general consensus is that auto-scaling helps to maintain performance predictability. Amazon Web Services (AWS) offers Auto Scaling, which automatically adjusts the number of EC2 instances in response to workload changes, allowing users to define scaling policies based on various metrics [283]. Similarly, Google Cloud Platform (GCP) provides an Autoscaler service that adjusts the number of Virtual Machine (VM) instances in managed instance groups based on load metrics or other signals like Stackdriver Monitoring metrics [284]. Microsoft Azure offers Autoscale capabilities for dynamically adjusting the number of virtual machines in scale sets based on predefined conditions such as CPU utilization or custom metrics [285].

In containerized environments, Kubernetes includes the Horizontal Pod Autoscaler (HPA), enabling automatic scaling of pods based on CPU utilization or custom metrics [286]. Serverless computing platforms such as AWS Lambda[267], Azure Functions[287], and Google Cloud Functions[288] inherently support auto-scaling, adjusting resources allocated to individual functions based on incoming requests or events. Moreover, modern Application Performance Monitoring (APM) tools like Datadog [289], New Relic [290], and AppDynamics [291] include auto-scaling features that dynamically adjust resources based on application performance metrics, optimizing system performance and resource utilization. Importantly, auto-scaling directly leads to core formal theory questions as to the applicability of control theory regimes and approaches, wherein these traditional theories require ergodic regimes so as to ensure a control rule indeed exists to be able to move the system from any operational state x into any desired state x', i.e., ergodicity is the formal requirement that ensures all x to x' paths exist.

For the sake of comparison, we shall note that classical load balancing distributes tasks across multiple servers using static methods such as round-robin or least connections, without seeking to adapt in real-time to changes in workload volumes. This makes load balancing simple to implement but less efficient for time varying loads. Dynamic load balancing, on the other hand, continuously monitors resource states and adjusts task re-allocation in real-time using complex algorithms such as weighted round-robin and dynamic hashing. This enhances scalability and performance, provided the sufficient required resources have been pre-allocated. Auto-scaling goes a step further by automatically adjusting the number of active resources based on real-time demand and predefined metrics, providing elasticity while seeking to optimize resource usage and minimize deployment costs. As such, it has become a common

feature of modern cloud environments and deployment frameworks as it is designed to address fluctuating workloads.

To be consistent, we apply Monte Carlo simulation to compare and quantify the effectiveness of classical load balancing and auto-scaling in their ability to address run-time performance predictability issues arising in cloud computing deployment regimes, More specifically, our focus is on assessing the methodologies' ability to address the losses in predictability(ergodicity) in the non-BET compliant regimes that are induced by queue drop events. We applied load balancing and auto-scaling techniques on S6-E3 ([L:P:F:F:TCP:D]) as it closely resembles real-world scenarios with the most complex performance behavior. In load balancing scenario as of the previous section, the incoming traffic is distributed across the shared pre-allocated cloud servers, with each modeled VM supporting between 4 to 12 containers so as to mimic the multi-tasking environments of real-world clouds. This Section introduces auto-scaling, where additional VM and containers are deployed only if the response time surpasses a client-side specified threshold. In the load balancing strategy, the end-goal is to prevent queue drops through having sufficient pre-allocated resourcing, whereas the auto-scaling strategy seeks to dynamically control the rate at which queue drops occur.

All scenarios are then tested against both Poisson and bursty LDSS incoming workloads, with Monte Carlo ensembles of 100 runs conducted for each simulated experiment scenario for each workload type. All parameters and statistical workloads, background VM loads, etc. are identical across all simulations so as to ensure consistency. This enables a fair comparison of the modeled LDSS performance under the different scenarios and workload behaviors, as required to quantitatively assess the resulting impacts within the resulting non-BET compliant operational regimes.

7.2.1 Scenario 1: No Resource Scaling

The baseline no scaling scenario was simulated as follows. The exemplar LDSS was deployed on one VM hosted on a PH, which also co-hosted additional non-LDSS VMs that themselves possessed time varying loads. The PH is simulated to have hypervisor employing fair scheduling with TCP used as the communication protocol. As per real-world deployments, queues within the hypervisors, VMs, and containers "drop" requests from the queue ends once the queue is full, i.e., a first-come-first-serve queuing policy is employed. Events were deemed as "timed-out" and then resent if

they failed to transition any given system component within a simulated 32 seconds, where this was set to be an intentionally long time-out period. At the per-event level, this "time-out and resend" process was repeated for a maximum set of 8 retries. If still no response was received after the full 8 retries then the given event was marked as "failed" with no further retries attempted.

The modeled and simulated LDSS therefore follows a very standard time-out and resend approach but with quite generous thresholds. Such thresholds are useful to highlight that the observed variability is cloud-induced and does not arise due to the impacts of tight thresholds. For all simulated scenarios, the fulls set (or family) of cdfs produced for the ensemble of 100 Monte Carlo runs is plotted, under both Poisson and bursty workload conditions. This is done as the observed results highlight non-BET compliant behaviors, rendering just plotting the ensemble average cdf as an uninformative approach. The bursty workload generated to have a peak workload of 10 times the normal workload volume with a statistical on/off pattern with 300 second maximums. The Poisson and bursty workload are illustrated in Figures 7.5 and 7.6.

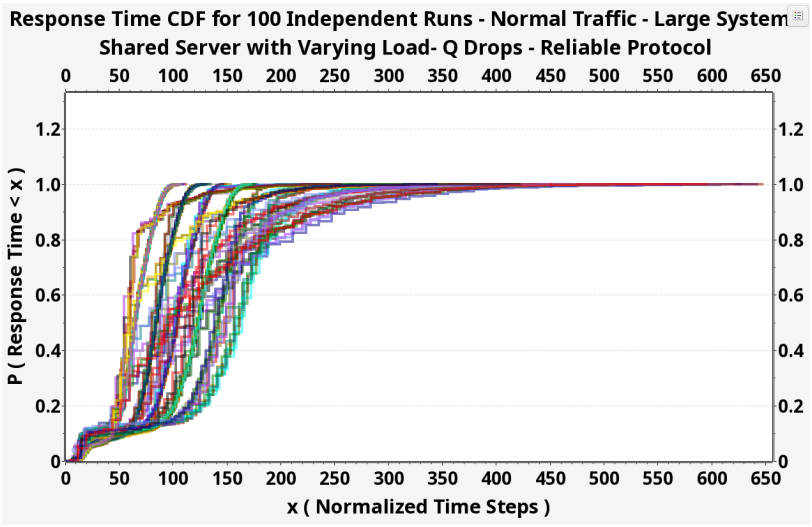


Figure 7.5: No scaling scenario for Poisson traffic.

7.2.2 Scenario 2: Classical Load Balancing

In Scenario 2, traditional load balancing was employed with the LDSS load balanced across four VMs hosted on different PHs, with each PH retaining the time varying background VM workloads as in Scenario 1. Four load balanced servers was chosen

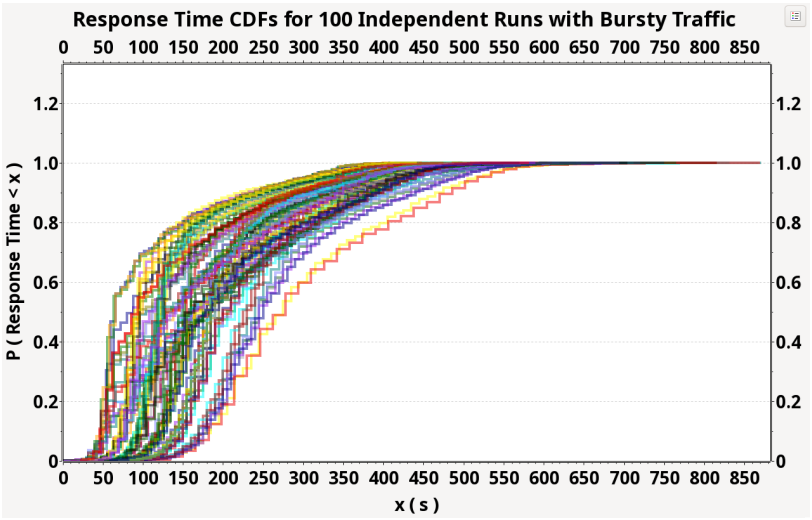


Figure 7.6: No scaling scenario for bursty traffic.

as the result of experiments in the previous section, which assessed that for the given LDSS and workload traffic, queue drop rates where minimized at minimum resource cost at four load balanced systems. Load balancing is performed via a standard round-robin load balancing strategy, which is selected given the homogeneous nature of the modeled LDSS deployment. As in Scenario 1, TCP is used as the communication protocol and the same timeout and resend processes and thresholds are employed. The cdf families for the 100 Monte Carlo runs under Poisson and bursty traffic are illustrated in Figures 7.7 and 7.8.

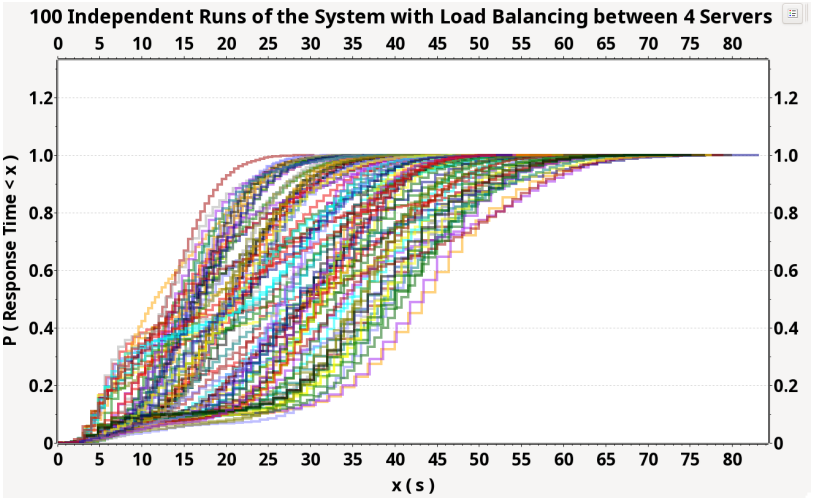


Figure 7.7: Load balancing scenario for Poisson traffic.

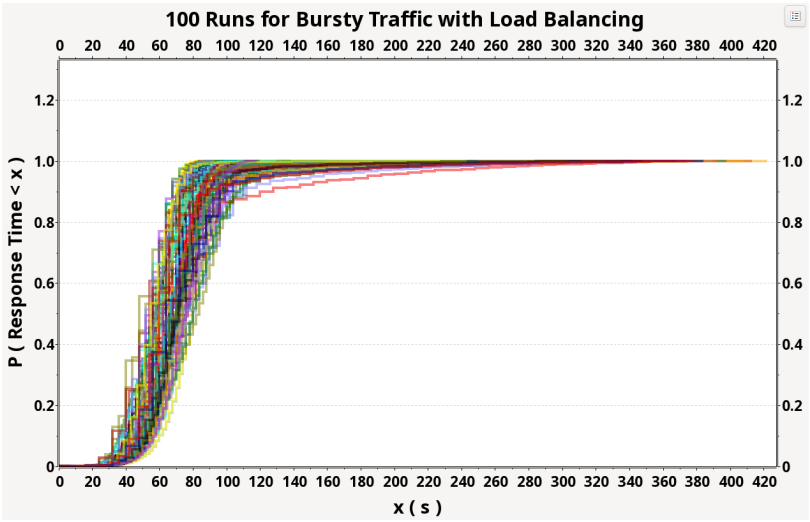


Figure 7.8: Load balancing scenario for bursty traffic.

7.2.3 Scenario 3: Auto-scaling

In scenario 3, the more recent approach of cloud-based auto-scaling was employed, as commonly underlies elastic services within commercial cloud platforms. Under the simulated auto-scaling, new VMs were launched based on predefined maximum per-event response time thresholds. For the simulated scenarios, the 2 client-side defined thresholds explored were set at 16 and 32 seconds. When the response time threshold was exceeded then auto-scaling was triggered with the system then initiating the deployment of an additional VM to help service the workload of each identified overloaded VM. In this manner, the system dynamically adapts its resource footprint adding VM resourcing only where and when overloads arise. More directly, overload events can be identified through either the exceeding of the maximum response time thresholds or by observing the occurrence of queue drop events, where in general the former is likely to precede the latter. Again, the 16 and 32 second maximum response time thresholds are intentionally set to be generous and result in overall maximum possible observable event response times of 112 to 224 seconds for the given LDSS. Identical timeout and resend policies to those of Scenarios 1 and 2 were used.

The full families of cdfs produced under Scenario 3 for the conducted 100 Monte Carlo runs under both Poisson and bursty traffic conditions are shown in Figures 7.9, 7.10 and 7.11. Of more interest in the auto-scaling scenario are the bursty workload results given that bursty workloads arise commonly within real-world industry Internet-based cloud-deployed LDSSes and elastic services. Auto-scaling provides the

promise of enabling more cost and performance effective deployment regimes over the more traditional load balancing methods. What constitute optimal auto-scaling or elastic service regimes though remain open research problems.

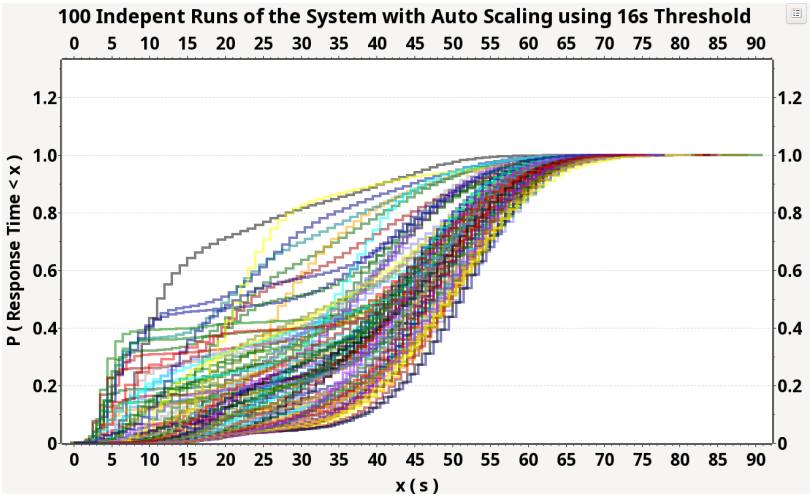


Figure 7.9: Auto-scaling scenario applying 16s threshold for Poisson traffic.

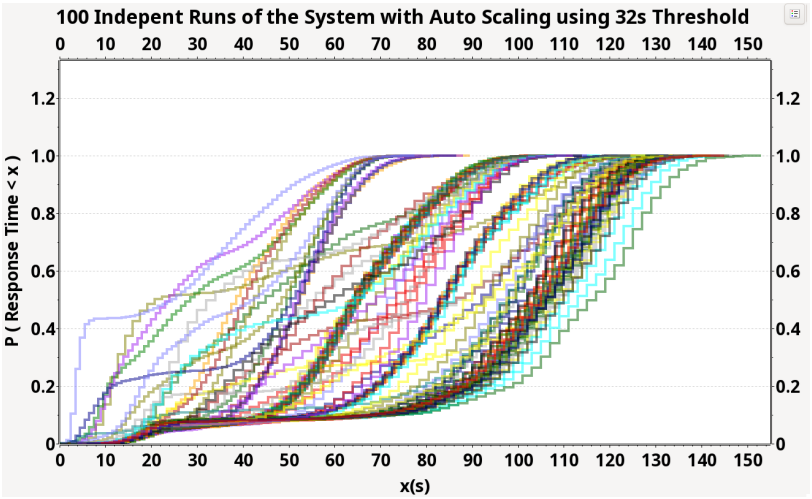


Figure 7.10: Auto-scaling scenario applying 32s threshold for Poisson traffic.

7.3 Observations and Insights

Queue drop events clearly can significantly increase the overall latency of a system as under reliable communication regimes the "lost" events must be resubmitted and re-processed. As more queue drops occur, more re-submissions arise. This significantly

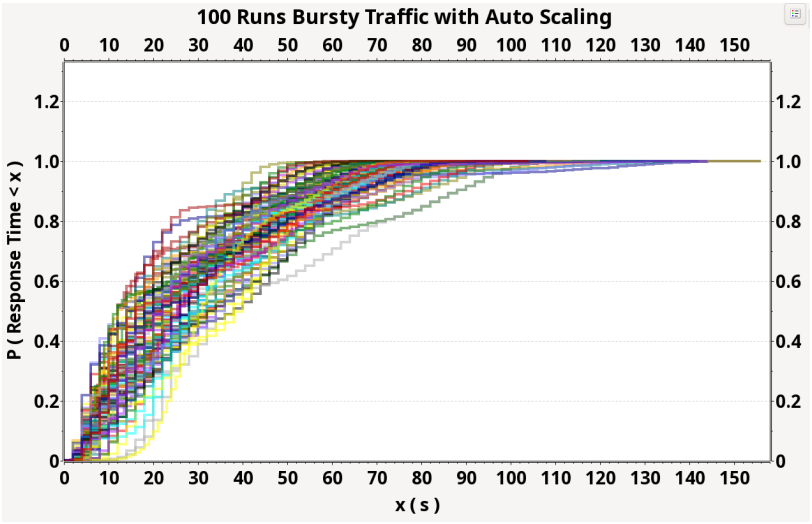


Figure 7.11: Auto-scaling scenario applying 32s threshold for bursty traffic.

increases the workload on a system already experiencing the onsets of overload, given that queue drop events are beginning to occur. This effect can be clearly seen for Scenario 1 in Figure 7.5 where a broad family of themselves quite broad cdfs results, i.e. the system lacks overall behavioral predictability and exhibits clear non-BET compliant behaviors. The Scenario 1 system has no ability to secure additional resources. Hence, high workloads lead to queue drops which lead to even higher workloads due to the reliable protocol triggered resend events.

Moreover, within all the provided cdf figures, only data arising from successful requests are plotted. All "fail" requests, as defined above, are not shown as they by definition do not produce defined response times. Table 6.50 provides the failure rates for each of the conducted simulation scenarios. As can be seen, maximum failure rates across all scenarios ensembles of 100 Monte Carlo runs are generally quite minimal, falling below a 1.6% level. The only exception is Scenario 1 under bursty workload where failure rates peak at 69.1%, as bursty traffic peaks and overloads the LDSS, which cannot acquire any additional resourcing to service these workload peaks.

Table 7.3: Failure rate and response time(s) comparison.

Simulation (Auto Scaling(A.S.))	**Failure Rate [Min. , Max.]**	**Response Time (s) Percentiles [Min. , Max.]**		
		90^{th}	**95^{th}**	**99.99^{th}**
Scenario 1 (Poisson)	[0.0% , 0.7%]	[84.2 , 270.6]	[89.1 , 320.0]	[108.3 , 632.1]
Scenario 1 (bursty)	[1.8% , 69.1%]	[222.2 , 486.9]	[288.1 , 534.7]	[384.7 , 869.8]
Scenario 2 (Poisson)	[0.0% , 0.0%]	[19.5 , 58.0]	[21.4 , 62.4]	[28.7 , 81.1]
Scenario 2 (bursty)	[0.0% , 1.0%]	[66.9 , 128.1]	[72.4 , 191.8]	[83.7 , 408.9]
Scenario 3 (Poisson-16s)	[0.0% , 1.0%]	[41.0 , 61.9]	[47.4 , 65.9]	[62.6 , 85.4]
Scenario 3 (Poisson-32s)	[0.0% , 0.2%]	[49.9 , 131.0]	[56.4 , 138.7]	[66.1 , 153.2]
Scenario 3 (bursty-32s)	[0.6% , 1.6%]	[38.6 , 86.9]	[42.1 , 97.6]	[54.3 , 142.8]

Table 7.4: BET-compliance of simulations based on the 3 scenarios.

Simulation	**Software Eng. Design Rules**				**Formally BET-compliant**	**cdfs**
	No. 1	**No. 2**	**No. 3**	**No. 4**		
Scenario 1 (Poisson)	✗	✗	✗	✗	✗	7.5
Scenario 1 (bursty)	✗	✗	✗	✗	✗	7.6
Scenario 2 (Poisson)	✗	✗	✗	✓	✗	7.7
Scenario 2 (bursty)	✗	✗	✗	✗	✗	7.8
Scenario 3 (Poisson-16s)	***T***	***T***	***T***	***T***	***T***	7.9
Scenario 3 (Poisson-32s)	***T***	***T***	***T***	***T***	***T***	7.10
Scenario 3 (bursty-32s)	***T***	***T***	***T***	***T***	***T***	7.11

In Table 7.4, "Transitional" (T) refers to situations where an LDSS deployment can move back and forth across the BET-compliance boundary due to changes in its run-time deployment and/or dynamic variations in its service workloads, such as those seen in bursty workload scenarios.

7.3.1 Comparative Analysis

The baseline Scenario 1 with static resourcing shows very high vulnerability to bursty traffic, as expected, and significantly higher response times at all percentiles. Scenario 2 is classical load balancing and with the four VM load balanced set demonstrates substantial improvements over Scenario 1. The failure rates drop to 0.0% under Poisson traffic and remains low even under bursty workloads at a 0.0% to 1.0% for

the 100 Monte Carlo runs. The observed response times are also far lower and more consistent than those of Scenario 1. Additionally, the families of produced cdfs, although still exhibiting non-BET compliance, are themselves narrower and more consistent highlighting a significant gain in the LDSS's overall run-time performance predictability. The Scenario 2 load balancing approach though requires that full four VMs are always available, irrespective if the current workload requires them at any given time. Hence, the deployment cost of Scenario 2 comes at a constant level of 4× that of Scenario 1, where the gained performance improvement can be seen to generally be less than a 4× improvement.

Table 7.5: Consumed Resources Comparison

Strategy	**Workload**	**Average Number of VMs**
No Scaling	Poisson	1
No Scaling	Bursty	1
Load Balancing	Poisson	4
Load Balancing	Bursty	4
Auto Scaling	Poisson-16s	3.1
Auto Scaling	Poisson-32s	1.4
Auto Scaling	Bursty-32s	9.2

Scenario 3 addresses the constant cost limitations of Scenario 2 via introducing auto-scaling. Enabling auto-scaling results in low failure rates and consistent response times even under bursty traffic conditions. The failure rates for Poisson workloads remain low, in the 0.0% to 1.0% range. Bursty traffic experiences slightly higher failure rates, in the 0.6% to 1.6% range, which is of the same order of improvement over Scenario 1 as with load balancing. But, the average number of utilized VMs varies significantly from that of Scenario 2, with 3.1 average VMs being used for Poisson workloads with a 16 second threshold, and only 1.4 average VMs used when the threshold is relaxed to 32 seconds. Under bursty traffic, the auto-scaled system more aggressively scales up to service the peak workloads, averaging 9.2 VMs. This adaptive scaling, better optimizes resource utilization and cost by deploying additional VMs only when necessary. The auto-scaling also results in significantly reduced variations (or spread) in the family of produced cdfs, although significant spread still exist. Hence, auto-scaling by itself is not sufficient to bring the LDSS into BET-compliant run-time behaviors. But, the observed variation can be seen to be substantially less than in both the Scenario 1 and Scenario 2 cases. Hence, the auto-scaling approach can be seen as a pathway to both maximize performance impacts at minimal costs

while also bringing the LDSS towards more predictable run-time behavioral regimes (or BET-compliance).

7.3.2 How Load Balancing and Auto Scaling Help with Performance Bounds

In the Scenario 1 with no-scaling approach, under the DST analyses of Chapter 5, no upper bound exists on the possible observed response times. Pragmatically, such bounds will depend on the reliable protocol thresholds or the users' patience. The response time is highly dependent on the host server's utilization rate, scheduling, queue drops, and the reliability of the protocol used. For the Scenario 1 experiments, the maximum possible response time is $32 \times 8 \times 7 = 1792$ seconds. Hence, any performance measure $x_k(t)$ of interest lacks any useful upper performance bound. Additionally, the $x_k(t)$ measures will be highly susceptible to both cloud-induced run-time variability and in-coming workload variability, i.e., a highly sensitive system will result.

In the load balancing scenario, additional resources are pre-allocated such that load can be distributed as a preventive measure for mitigating server overloads. While this reduces the likelihood of excessive delays, there is still no formal upper bound on observable performance. The response times depend on the incoming workloads patterns, the PH time varying utilization and scheduling processes, as well as a number of other factors. The additional resources allow for significant reductions in failure rates over those of Scenario 1, as well as more stable response times for Poisson workloads. However, very long and variable response times remain for bursty traffic, as detailed in Table 6.50. Moreover, the families of produced cdfs still exhibit considerable spread, even though this spread is a significant improvement over the cdf spread observed for Scenario 1.

By comparison, auto-scaling is the only strategy that allows for a formal and predefined upper maximum response time. For example, assumed client-defined limits of 32 or 16 seconds. In auto-scaling the system dynamically adjusts resource allocations to maintain these specified performance targets, irrespective of cloud- and workload-induced variations. Hence, auto-scaling provides an approach that admits formal upper bounds on the $x_k(t)$ performance measures. More specifically, such a constraint on the maximum provided that cloud has available the requested auto-scaling resources. Table 7.5 illustrates that auto-scaling improves resource utilization while maintaining

low failure rates and response times, even under bursty traffic conditions, thereby providing a more robust solution for managing performance variability. Moreover, as can be observed, the families of produced cdfs are again even narrower in spread that the Scenario 2 load balancing case. Hence, auto-scaling also acts to significantly improve the run-time performance predictability of an auto-scaling managed LDSS in manners not achievable under load balancing. It should be noted that although the auto-scaling improvement to cdf family spread are significant, they still are insufficient to bring the run-time LDSS behaviors back to being fully BET-compliant.

As highlighted through Chapters 5 and 6, the above Monte Carlo experiment results are fully explainable through the developed Software Engineering Design Rules and their violations, particularly Rules 1 and 2. Importantly, the underlying Chapter 5 DST theory analysis highlights that no bounds exist on the region (or neighborhood) with the produced cdfs can span or the "shapes" they can exhibit. More particularly, the furthest left cdf can be pragmatically viewed as denoting the "quiescent system" cdf in many cases, but the Chapter 5 theory highlights that no right boundary "worse case" is generally derivable. Additionally, the Chapter 5 theory highlights that a very wide variety of cdf "shapes" are possible as these are relative to the vast number of per-instance and specific ways the Software Engineering Design Rules 1-4 can be violated during a system's run-time operation.

7.4 Operating System Scheduling Algorithms

The scheduling algorithms implemented in operating systems or hypervisors play a critical role in determining BET-compliance within a system as observed in [239] and investigated in Chapter 5, producing Software Engineering Design Rule 3, which was then validated in the Chapter 6 simulations. In practical scenarios, hypervisors often allow for over-commitment onto physical servers, enabling the provisioning of more virtual machines (VMs) than the available physical resources can fully support, i.e., substantially more active VMs than the available CPU cores. In such cases, effective scheduling decisions become critical in managing resource sharing and mitigating resource contention. As explained in Chapter 5, with fair scheduling, we cannot guarantee BET-compliance, as per Software Engineering Design Rule 3 and Theorem 6.

In this Section, we focus on the scheduling mechanisms of two prominent hypervisors, Hyper-V as a Classical fair scheduler and Xen for RTOS, and quantifying their impacts on seeking to improve run-time performance predictability.

7.4.1 Hyper-V Scheduler

Hyper-V [240], developed by Microsoft, offers several modes of scheduler logic that determine how virtual processors are scheduled on underlying logical processors. These modes include the classic scheduler, the core scheduler, and the root scheduler.

7.4.1.1 Classic Scheduler

The classic scheduler has been the default choice for the Windows Hyper-V hypervisor since its introduction, including Windows Server 2016 Hyper-V. It offers a fair share, preemptive, round-robin scheduling model for guest virtual processors [240].

In this way, the classic scheduler ensures that each virtual machine gets its fair share of CPU time, preventing resource contention and promoting efficient utilization of the host's CPU resources. However, it may not be optimized for certain scenarios with specific performance requirements, as it operates on a fixed time-slicing basis without considering the priority or workload characteristics of individual virtual machines.

7.4.1.2 Core Scheduler

The hypervisor core scheduler was introduced as an alternative to the classic scheduler logic in Windows Server 2016 and Windows 10 version 1607. It provides enhanced security and reduced performance variability for workloads running inside virtual machines (VMs) on a simultaneous multi-threading (SMT)-enabled virtualization host. The core scheduler leverages the SMT topology of the virtualization host and allows for running both SMT and non-SMT VMs simultaneously. It schedules groups of guest virtual processors (VPs) from the same VM onto groups of SMT logical processors (LPs) in a symmetric manner. For example, if LPs are organized in pairs, VPs are scheduled in pairs, ensuring that a core is never shared between VMs which reduces vulnerability to side-channel snooping attacks from malicious VMs [240].

However, there are cases where only one VP from a group of VPs can be actively executed at a given time due to contention or other resource constraints. In such a situation, the core scheduler chooses one VP to run while the other VPs in the group

remain idle. This happens because the physical core can only execute one instruction stream at a time, and the other VPs have to wait for their turn. The idling of the remaining VPs in the group represents a potential performance reduction. Since only one VP is executing instructions while others are idle, the overall processing power of the core is not fully utilized. This can lead to decreased throughput and less efficient use of the available CPU resources. It's important to note that this performance impact is a trade-off introduced by the core scheduler's approach of grouping VPs from the same VM together. While it improves security and reduces contention in some cases, it can potentially lead to under-utilization of CPU resources when only one VP can run at a time. We should note also when a VP is scheduled for a VM without SMT enabled, that VP consumes the entire core when it runs [240]. The default scheduler has transitioned to the core scheduler starting from Windows Server 2019, while the classic scheduler remains the default for Windows Server 2016 unless explicitly enabled by the Hyper-V host administrator.

7.4.1.3 Root Scheduler

With the introduction of Windows 10 (version 1803), the root scheduler was introduced as a new type of scheduler. When enabled, it transfers the responsibility of work scheduling from the hypervisor to the root partition. The NT scheduler within the root partition's operating system instance takes charge of workload allocation and distribution across logical processors (LPs). Currently, Microsoft advise against employing the root scheduler with Hyper-V on servers. Its performance attributes have not been thoroughly characterized and optimized to accommodate the diverse array of workloads commonly encountered in server virtualization deployments. [240]

7.4.2 Xen Scheduler

Xen is a bare-metal hypervisor that runs directly on the hardware without the need for a host operating system. The Xen-based hypervisors, widely recognized and adopted in the industry [292], employ efficient scheduling mechanisms for allocating virtual CPUs (vCPUs) to physical CPUs (pCPUs). Utilizing the capabilities of modern CPUs that support multiple threads per core, the Xen hypervisor assigns vCPUs to different threads within a physical core. This approach is enabled by a two-level hierarchical scheduling framework that allocates pCPUs to virtualized hosts, enabling the simultaneous execution of multiple VMs. Within each VM, the assignment of multiple

vCPUs allows for parallel execution of tasks, thereby introducing an additional layer of virtualization and parallelism.

7.4.2.1 Real-Time Deferrable Server(RTDS)

An important scheduler available in the Xen hypervisor is the Real-Time Deferrable Server (RTDS) scheduler. Designed specifically for soft real-time systems with strict response time requirements, the RTDS scheduler ensures that tasks with real-time constraints meet their specified deadlines. By providing guarantees for timely task completion, the RTDS scheduler reduces response time variability and enhances predictability. The rationale behind this, is quite evident as it restores the system to an ergodic state by ensuring measure invariance for macro performance assessment.

7.4.2.2 Soft Real-Time Operating Systems(SRTOS)

Soft real-time design centers around ensuring that individual tasks meet specified time constraints rather than optimizing the total time spent on all tasks. For example, a soft real-time system may strive to complete 99.9 of requests within 10ms, processing 100 requests per second. In contrast, a non-real-time system might aim for 200 requests per second, allowing occasional requests that can block for 50ms or more. Meanwhile, a hard real-time system commits to processing one request every 15ms without exception.

Soft real-time is applied in scenarios where missing a deadline leads to service degradation but isn't performance-critical. Common use cases include multimedia and telephony, where timely rendering of audio or video frames is crucial, and occasional frame skipping is acceptable. In Erlang, reliability took precedence over maintaining continuous connections, accepting occasional call drops to ensure the overall functionality of the telephone exchange.

Conversely, applications like motor control demand software that never misses a deadline to ensure precise control. While this approach involves performance trade-offs and limits the complexity of achievable behaviors, it is indispensable for critical applications. On the other hand, tasks like number crunching prioritize overall performance, emphasizing the speed of operations like matrix multiplication over individual component processing speed.

7.4.3 Assessing Xen RTOS and Apache-like Resource Scaling

Real-Time Operating Systems (RTOS) are designed to manage and operate applications with strict timing constraints. As discussed in Chapter 5, RTOS helps address measure-preserving issues by ensuring that tasks are completed within a specified time frame. This characteristic can be used to define the time-shift transformation T to be larger than this time frame, ensuring predictable behavior within that period. Therefore, RTOS can provide BET-compliant behavior within the specified time frame, contributing to predictable system performance. The only caveat here is the possible high failure rate in case of overloaded servers.

Figure 7.12 and Figure 7.13 show cdfs for response times of 100 independent runs of the exemplar LDSS of Section 7.1 and S6-E3 [L:P:F:F:TCP:D] in a modeled cloud environment employing RTOS scheduling and load balancing across four servers, as informed by the prior section's results. The RTOS scheduling produces a tighter family of cdfs. The simulated RTOS has a hard limit of 8 seconds per process execution, resulting in a hard maximum of 48 seconds in the first system and 56 seconds in the system from S6-E3 experiment as the longest possible observed response time, with any unserviced requests being dropped after the 8-second mark in any part of the system.

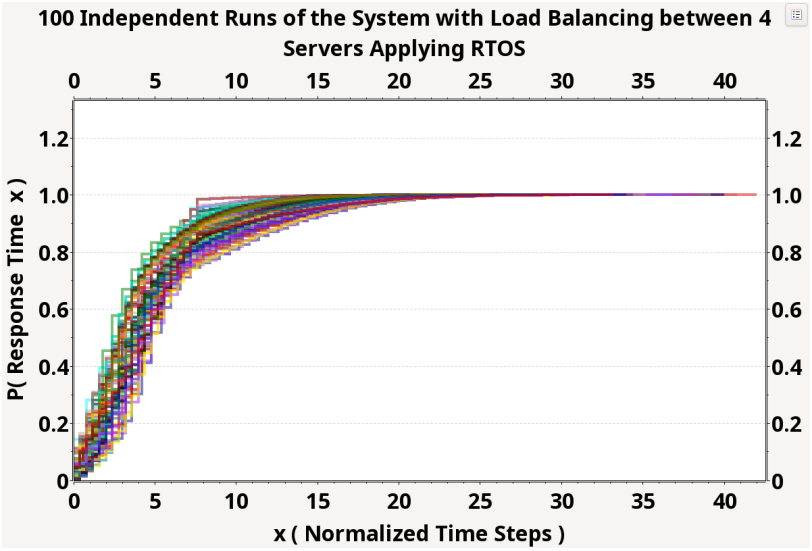


Figure 7.12: Applying RTOS and load balancing on system from Section 7.1 with 6 distributed servers and infinite queues.

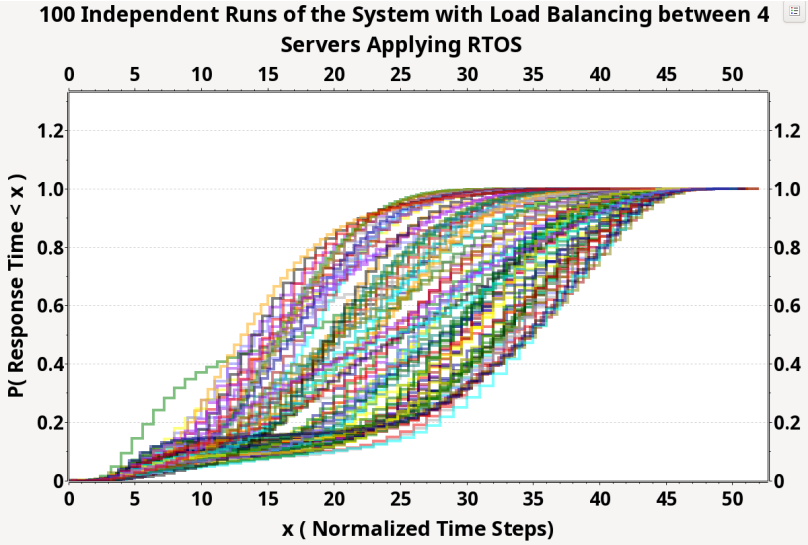


Figure 7.13: Applying RTOS and load balancing on S6-E3 experiment. [L:P:F:F:TCP:D]

7.4.4 Assessing Xen RTOS and Cloud-like Resource Auto-Scaling

RTOS ensures that tasks are completed within defined time frames, providing predictable BET-compliant behavior even under varying loads, while auto-scaling adjusts resources dynamically to meet demand, preventing overloading and maintaining the hard limit of RTOS. Therefore, this integration is the only possible solution to maintain the BET-compliance of the system and leads to improved performance vs cost results.

7.4.5 Comparative Analysis

Table 7.6 illustrates the impact of different configurations—baseline, load balancing (L.B.), auto-scaling (A.S.), and their combinations with RTOS—on the run-time performance predictability of LDSSes using Sc6-E3 experiment of Chapter 6. Starting with the baseline configuration using a single VM, response times are high and failure rates are around 4% in cases where the Physical Host(PH) is overloaded. This configuration is setting a reference point for evaluating other configurations. Load balancing with four VMs significantly reduces response times across all percentiles and eliminates failure rates, showing the effect of PH load on performance predictability.

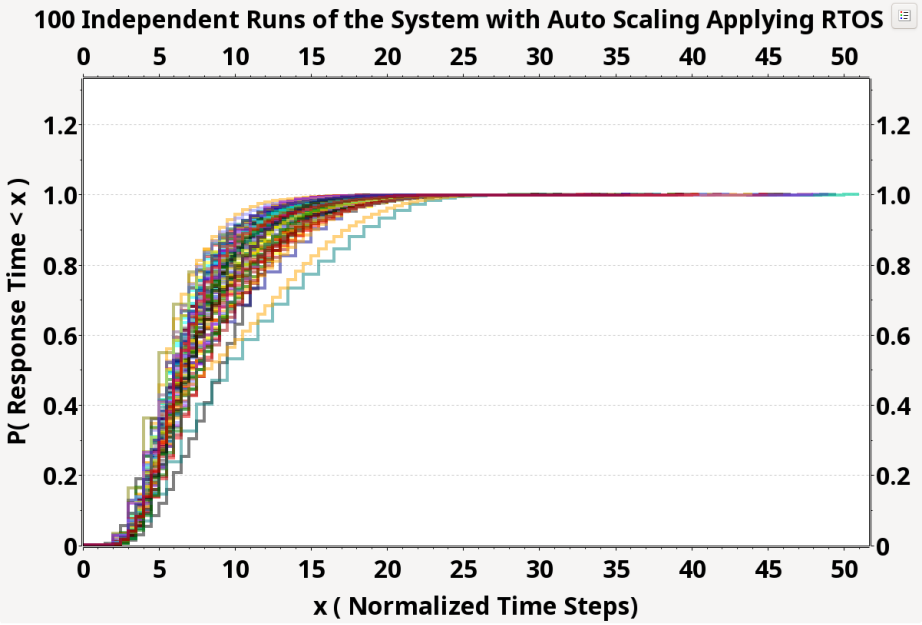


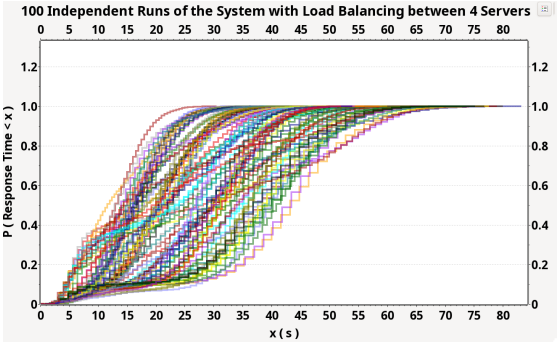
Figure 7.14: Applying RTOS and auto-scaling on S6-E3 experiment. [L:P:F:F:TCP:D]

Adding RTOS to load balancing further improves maximum response times, but introduces a potential for high failure rates because RTOS turning down the requests that can't be serviced in the specified hard limit. auto-scaling with an average of 3.1 VMs provides moderate improvements generally but very similar improvement to load balancing for the maximum response time but also with slightly increased failure rates compared to load balancing.

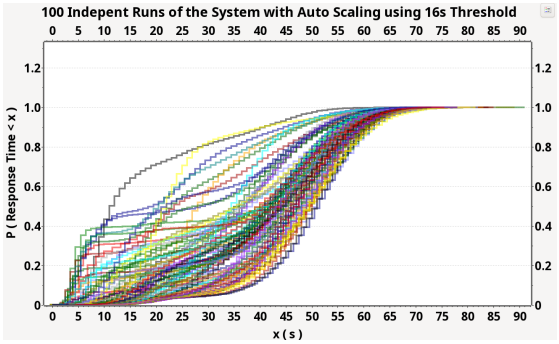
Table 7.6: Performance improvement rate comparison based on Sc6-E3 experiment.

Simulation	**VMs**	**Imp. Rate on [Min. , Max.]**	**Failure Rate [Min. , Max.]**	**Response Time (NTS) Percentiles [Min. , Max.]**		
				90^{th}	**95^{th}**	**99.99^{th*}**
Baseline	1	N/A	[0.0% , 4.4%]	[87.0 , 259.5]	[88.5 , 319.5]	[106.0 , 632.0]
L.B.	4	[74.0% , 87.0%]	[0.0% , 0.0%]	[19.8 , 58.0]	[21.5 , 62.0]	[27.5 , 81.6]
L.B. & RTOS	4	[70.1% , 92.4%]	[0.0% , 85.6%]	[22.0 , 42.4]	[24.6 , 44.8]	[31.6 , 48.8]
A.S.	3.1	[40.9% , 86.4%]	[0.0% , 1.0%]	[41.0 , 61.9]	[47.4 , 65.9]	[62.6 , 85.4]
A.S.& RTOS	4.4	[85.3% , 92.9%]	[0.0% , 0.0%]	[9.0 , 18.5]	[10.5 , 19.0]	[15.5 , 44.5]

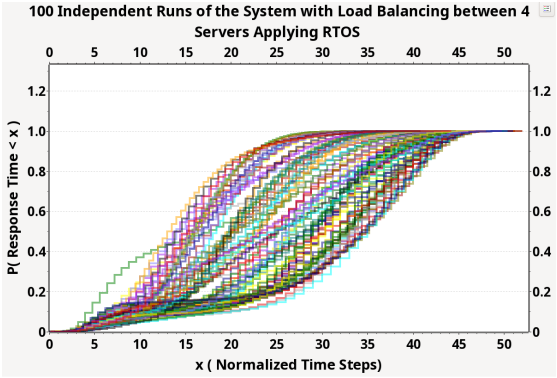
The combination of auto-scaling and RTOS with 4.4 VMs yields the best performance among conducted experiments, showing the lowest response times across all percentiles and zero failure rates. This configuration offers the most predictable and stable performance, making it the optimal choice for achieving reliable run-time



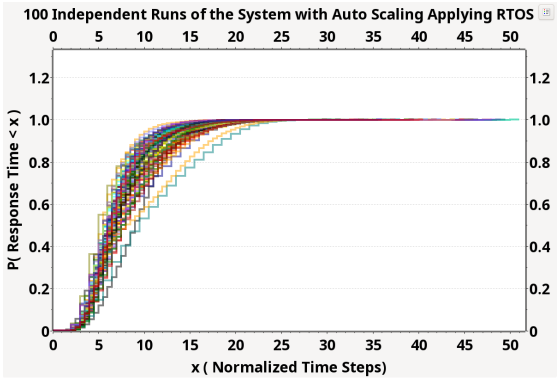
(a) S6-E3 ten hours of simulation with load balancing on 4 servers. [L:P:F:F:TCP:D]



(b) S6-E3 ten hours of simulation with auto-scaling. [L:P:F:F:TCP:D]



(c) S6-E3 ten hours of simulation with load balancing on 4 servers and applying RTOS. [L:P:F:F:TCP:D]



(d) S6-E3 ten hours of simulation with auto-scaling and applying RTOS. [L:P:F:F:TCP:D]

Figure 7.15: RTOS applied on load balancing and auto-scaling.

performance in LDSS cloud environments. However, it is essential to consider the increased cost and complexity associated with RTOS implementations. Based on DST-driven insights from Chapter 5, the combination of auto-scaling and RTOS helps to maintain the system's ergodicity in two ways. First of all, by applying auto-scaling we are preventing the ongoing queue drops and therefore triggering reliable protocol which leads to wandering sets of non-zero measures. Secondly, the application of RTOS helps to have transformation T as a measure-invariant transformation with respect to performance measure in the RTOS hard limit time window.

In Table 7.7, "Transitional" denotes situations when a LDSS deployment can cross back and forth over the BET-compliance boundary due to changes in its run-time deployment and/or dynamic variations in its service workloads, e.g., as per bursty workload scenarios.

Table 7.7: BET-compliance of simulations on S6-E3 experiment.

Simulation	Software Eng. Design Rules				Formally BET-compliant	cdfs
	No. 1	No. 2	No. 3	No. 4		
Baseline	✗	✗	✗	✗	✗	7.5
Load Balancing	✗	✗	✗	✓	✗	7.7
Load Balancing & RTOS	*T*	*T*	*T*	*T*	*T*	7.13
Auto Scaling	*T*	*T*	*T*	*T*	*T*	7.9
Auto Scaling & RTOS	*T*	*T*	*T*	*T*	*T*	7.14

7.5 Insights and Observations

The above simulation results clearly highlight the value of industry advancements such as auto-scaling and the ROTS based hypervisors, particularly over traditional methods such a load balancing. Again, these industry approaches has largely arisen via informed but ad hoc innovation processes. Our work, particularly that of Chapter 5, allows these methodologies to be placed onto solid deep theoretical foundations provided via DST. Importantly, the industry-developed methodologies reduce but do not eliminate violations of the Software Engineering Design Rules developed in Chapter 5. This explains why families of cdfs are still produced, even though their overall span, at least across the conducted 100 Monte Carlo ensembles, is clearly reduced, thereby significantly improving run-time performance predictability. The underlying theory in Chapter 5, however, highlights that establishing bounds on the producible cdfs and their possible characteristics is not feasible. Hence, larger Monte Carlo ensembles will produce richer cdf variations, but with the same overall trend of observing tighter sets of behaviors for systems employing auto-scaling and/or RTOS scheduling.

Importantly, the issue of how to best apply approaches such as auto-scaling and RTOSes to bring run-time software systems back to being fully BET-compliant remains an open question. The DST approach developed in Chapter 5 can be used to develop and assess potential and, ideally, optimal solution approaches.

Additionally, both load balancing and auto-scaling effectively mitigate the impacts of cloud-induced variability. But only auto-scaling is able to formally place a system's performance measures within predefined maximums. Pragmatically, load balancing achieves effective bounds by distributing the load across the sets of pre-allocated resources but often this results in potential over-provisioning. Auto-scaling's dynamic ability to adjust resource allocations to ensure client-defined performance limits are

met can enable better and more predictable performance at lower costs. Notably, these improvements come without increasing the failure rates of the respective systems. An issue previously not highlighted in the literature is the degree to which load balancing and auto-scaling impact the observed richness of the produced cdf families, i.e., the degree to which the systems exhibit non-BET compliant behaviors. As was highlighted, auto-scaling significantly improves these issues over both the baseline and load balancing scenarios, but still not to sufficient degree that the LDSS could be considered to have been brought back to being a BET-compliant system. As such, run-time predictability can still vary significantly for different LDSS instantiations and run-time cloud deployments even under auto-scaling regimes. On the other hand, if we apply RTOS and auto-scaling we can bring back the systems closer to become a BET-compliant system by adhering to software system's design rules of Chapter 5 which derived from formally investigating ergodicity of software systems.

Notably, combining the theory, observations, and insights from Chapters 5 to 7 suggests that it should be possible to construct an optimal adaptive control and management process for run-time software systems. Such a system would actively work to maintain and ensure BET-compliance while achieving maximal resource efficiency, i.e., at minimal resource cost.

This chapter's assessments make it clear that current auto-scaling methodologies help but are also non-optimal. Moreover, the open question remains regarding the nature and characteristics of an approach that would assuredly place bounds on the viable neighborhood in which the cdf families could exist. These bounds might be statistical bounds relative to the cdf families' own underlying distribution of possible cdfs. Both issues are non-trivial but approachable via the theory and Software Engineering Design Rules developed in Chapter 5. Hence, these have been left as areas of future work, which importantly would not be approachable without the theoretical contributions and insights developed in Chapter 5.

Chapter 8

Conclusions, Contributions and Insights, and Future Work

The Section below highlight the conclusions and summary of the conducted research, the main novel contribution and insights gained through the work, and avenues and directions left for future work.

8.1 Conclusions and Summary

This research began with the aim of better understanding the bifurcation between industry and academia regarding their respective views on the nature and complexity of building software-centric systems that exhibit assuredly predictable run-time behavior. The academic literature largely suggests that software system behavioral predictability is a mostly solved problem, whereas industry recognizes it as a difficult and largely unresolved issue outside of very specific contexts. This problem is significant because modern societies increasingly depend on large- to global-scale software systems for various critical operations. Building systems that behave predictably in all likely real-world operational contexts is a fundamental principle and requirement of any engineered system. Therefore, developing the theory and understanding needed to achieve this within modern, large-scale, and highly complex software systems is a core societal expectation of software engineering as an engineering discipline.

This led to a detailed review of current literature and industry perspectives in **Chapter 2**. The analysis highlighted that, due to the inherent limits of academic contexts, it has become commonplace for industry to deploy, manage, and operate

software-centric systems that are orders of magnitude larger in scale and complexity than those typically studied in academic settings. In general, with modern coding frameworks and deployment regimes, even smaller high-tech startups commonly deploy systems serving global client bases with relatively small core software engineering teams. These systems, even at smaller scales, still cost millions to build and hundreds of thousands per year to operate, placing them well outside the scope of all but the most well-funded global academic research labs. Consequently, most academic researchers have transitioned to simulation-based studies and explorations, generally involving Monte Carlo approaches and analysis frameworks.

Markovian-based analysis methods have generally dominated the academic literature, while industry must focus its efforts on at-scale, real-world deployed systems. This distinction allowed for identifying a key and core differentiator between industry and academic approaches that may explain the ongoing bifurcation. In particular, **Chapter 3** highlighted that the vast majority of the published academic literature either implicitly or explicitly is based on the presumption that the systems being studied and modeled are statistically ergodic. However, a detailed search of the literature showed that no general proof of ergodicity holding within modern software-centric systems exists. Moreover, the methodologies by which ergodicity conclusions are supported within classical physical systems do not translate to software-centric system contexts, i.e. standard Hamiltonian and Laplacian descriptions of conservative systems do not apply. This led the research to focus on the need to formally assess ergodicity within the context of modern software-centric systems and their deployment regimes. For this purpose, ergodicity was defined in terms of Birkhoff's Ergodic Theorem (BET) and BET-compliance, as it applies to σ-finite measure spaces.

The statement of BET, however, is within the context of dynamical systems theory (DST). Thus, an approach was required to map software-centric system and their run-time behaviors into the required DST context. More particularly, it was useful to build on the queuing network (QN) and hierarchical QN approaches developed to support Markovian analyses, as these have been previously shown to provide general run-time inclusive models of software systems. **Chapter 4**, therefore, developed the bridge between QN software system models and the previously developed Extended Maurer Model (EMM), to enable this bridge between QN models and DST so as to support the subsequent BET analyses. Moreover, the formal structure of the EMM, as based on Maurer's original Turing-reducible model, enabled the analysis to be limited to the context of σ-finite measure spaces, i.e., the boundary of the spaces over which

probabilities can be formally defined. This is an important consideration as ergodicity is a far more complex concept within the context of infinite measure spaces. Via the EMM, we are able to show that σ-finite measure spaces suffice for modern arbitrary scale and complex software-centric systems, while infinite measure space may or may not be required in the performance study of quantum computing-based systems, the latter being an open research question.

Importantly, to support modern cloud VM and container deployment regimes and fair OS scheduling processes, it was necessary to extend the traditional QN model to incorporate stochastic On/Off switching at the per-queue level. This extension more accurately reflects real-world cloud operations, where commercial cloud servers support more active VMs than CPU cores, leading to time-sliced CPU resources and stochastic On/Off periods for each "active" VM. To our knowledge, this important reality has not been addressed in prior published works.

The QN and EMM-based approach of Chapter 4 allowed for a full formal BET-compliance theoretical analysis to be conducted in **Chapter 5**, proving in the general case the necessary and sufficient conditions for the run-time behaviors of software-centric systems to be BET-compliant, or conversely, to identify when and why a system becomes BET non-compliant. A critical understanding within Chapter 5 was provided via Peter Walters' Ergodic Equivalency Theorem (3), whereby it was observed that Markovian approaches were based on the recurrence requirement denoted by Statement 3 of Walters' theorem, whereas Statements 2 and 4 provide equivalent mathematical conditions. Within the literature, it has proven difficult to formally ensure recurrence holds within systems of arbitrary scales and complexities. Our core insight was that, via the QN and EMM approach, Statements 2 and 4 are relatively easy to prove within scale and complexity-agnostic contexts. This then led to the Chapter 5 theorems and proofs that: i) wandering sets of non-zero measure (Statement 4) are produced by queue drop events and the recovery actions of reliable protocols, and ii) event space variability and the loss of measure invariance are created by time-variant stochastic effects of fair OS scheduling regimes and resource contention and utilization issues.

As the claim in Chapter 5 to a general understanding of ergodicity and the necessary and sufficient requirements for general achievability within arbitrary scale and complexity software-centric systems and deployment regimes is quite strong, **Chapter 6** focused on providing a detailed validation of the Chapter 5 results and insights. A Monte Carlo simulation framework was developed, utilizing the lower-level services

and packet-level capabilities provided through the OMNet++ and INET frameworks. This simulation framework modeled modern container and VM-based cloud computing software system deployments, inclusive of Poisson and bursty incoming workloads, fair and RTOS OS scheduling, background server VM processes, and more.

The simulation framework developed in Chapter 6 was directly informed by the DST analyses and insights from Chapter 5, ensuring that all pertinent aspects highlighted via DST were retained and incorporated into the simulator, as well as the conducted simulation and subsequent analysis methodologies. The analysis was done in the context of the presumption of BET non-compliant regimes. Hence, the full families of produced cdfs were retained, as opposed to the more commonplace approach of simply reporting the ensemble average results, which, by definition, is only meaningful if BET-compliance holds.

To maintain real-world relevance, Chapter 6 then proceeded with simulating the impacts of the four Chapter 5 derived Software Engineering Design Rules for an exemplar Software-as-a-Service (SaaS) system modelled after a real-world industry-held system for which a previous study had explored it cloud deployment run-time performance predictability[4]. As this system was quite a simple system, it exhibited behavioral complexities result from the impacts of the deployment regime(s) as opposed to any inherent complexity within the SaaS system itself. The combination of the fours Software Engineering Design Rules, with testing against Poisson and bursty workloads, and against fair, and fair and RTOS-based OS scheduling, resulting an a total of 96 experimental scenarios, which were all run for Monte Carlo ensembles of 100 runs each. As subset of the 34 most important of these results were presented in Chapter 6, with the results although exhibiting quite complex families of cdf behaviors aligning with the expectations and understandings provided via the Chapter 5 developed theory and design rules, i.e., all of the results of the Chapter 6 experiments were consistent with the expectations and understandings of the Chapter 5 DST results.

Finally, **Chapter 7** applies the Chapter 6 simulation approach to explore and quantify the impacts of two modern software system deployment options of: i) Apache Spark and Storm like auto-scaling, and ii) Xen-like hypervisor soft RTOS scheduling, where these are compared against classical load balancing solutions. Of interest is not whether or not such approaches improve performance, given this was the goal of their development by industry, but instead to quantify the degree to which they comparatively increase run-time performance predictability. Moreover, it was shown

that although (i) and (ii) both decrease the span of the family of cdfs produced via the Monte Carlo simulation, they are insufficient approaches to being the system back to being fully BET-compliant. This result is explainable via the Chapter 5 developed software engineering design rules, given that the (i) and (ii) are insufficient to ensure that none of this design rules are violated. Instead, the rate at which the rules are violated is reduced, but violation still occur thereby leading to the observed BET non-compliance. The contribution of Chapter 7 is to place this industry developed methodologies onto the Chapter 5 developed formal DST analysis foundations, whereas such mathematical foundations have not been previously available. Hence, via Chapter 5 the (i) and (ii) approaches can be determined to be requirement for BET-compliant run-time software system performance measures to exist as opposed to simply being "nice-to-have" features that appear to improve performance predictability.

8.1.1 Software Engineering Design Rules

The DST theory insights and observations led to the construction of four pragmatically easily measurable and real-world implementable Software Engineering Design Rules, which denote the sufficient and necessary conditions required to keep a software-centric system within a BET-compliant run-time regime.

The theoretical derivation of these rules is thoroughly detailed in Chapter 5, where extensive DST analyses and formal proofs are employed to establish their necessity, importance and applicability. These design rules are not merely theoretical constructs; they are mandatory guidelines that software engineers must apply in real-world scenarios to ensure their systems exhibit predictable and reliable behavior according to Eq. 5.14. The effectiveness of applying these rules is validated through the simulations presented in Chapters 6 and 7. Chapter 6 demonstrated how violations of these rules lead to BET non-compliant systems, highlighting the negative consequences of non-adherence. Conversely, Chapter 7 illustrated how following these rules brings systems closer to BET-compliance, providing empirical evidence that adherence to these design rules results in systems that meet the required criteria for predictability.

More particularly, BET compliance was assessed with respect to when LDSS performance measures $x_k(t)$ admit statistically predictable (or bounded) behaviors such that:

$$P[x_k(t) \in [B_{lower}(t), B^{upper}(t)]] \geq 1 - \epsilon \; \forall t \in [0, T] \tag{8.1}$$

for some appropriately small $\epsilon > 0$, i.e., some appropriate confidence level. This results in the following derived BET-compliance softare engineering design rules:

Rule 1: Controlling for Queue Drops

For deployed LDSSes on-going queue drop events produce wandering sets of non-zero measure causing BET-compliance to be violated. Queue drop events therefore must be actively managed and mitigated if all LDSS's run-time performance measures are to be statically predictable.

Rule 2: Impacts of Reliable Protocols

For deployed LDSSes, the actions of reliable protocols produce wandering sets of non-zero measure causing BET-compliance to be violated. Therefore, reliable protocols, although necessary, should not be overused if all LDSS's run-time performance measures are to be statically predictable.

Rule 3: Real-time OS Scheduling

For deployed LDSSes, fair OS scheduling under non-quiescent operational regimes lead to stochastic event completion times causing BET-compliance to be violated. Where possible, RTOS should be used if all LDSS's run-time performance measures are to be statically predictable.

Rule 4: Impacts of Resource Utilization and Contention

For deployed LDSSes, resource contention and utilization issues can produce stochastic event completion times causing BET-compliance to be violated. Therefore, resource contention must be minimized and resource utilization levels carefully monitored if all LDSS's run-time performance measures are to be statically predictable.

It should be noted that a run-time LDSS can concurrently support $x_k(t)$'s that do or do not violate some, all, or none of the rules. Moreover, for given $x_k(t)$ this can change dynamically. Hence, the rules are required if *all* $x_k(t)$ are to be statistically predictable, but they do not preclude that some $x_k(t)$'s can exist that exhibit statistical predictability even when all the rules are violated, i.e., as would generally occur within quiescent systems under fair OS scheduling experiencing rare but non-zero queue drop events. This highlights the behavioral complexity of modern larger-scale LDSSes in that, as per the research's developed insights and understandings, can actively cross back and forth between BET-compliance and non-compliance and at the level of individual $x_k(t)$ run-time performance measures.

8.2 Contributions and Insights

The core novel contribution of this research is the development of a generally applicable, scale- and complexity-agnostic approach to assessing and understanding BET-compliance for the run-time performance measures of software-centric systems. No such methodology currently exists in the literature, as formally assessing recurrence properties within large-scale, highly complex systems has proven difficult. This contribution was achieved by applying the Extended Maurer Model to QN software system modeling approaches common to Markovian analysis, while also incorporating stochastic per-queue level On/Off switching to reflect the realities of modern cloud computing deployment regimes, such as VMs and containers. Statements 2 and 4 of Peter Walters Ergodicity Equivalency theorem (3) for σ-finite measure spaces were then applied and shown to avoid the scale and complexity issues inherent within Statement 3, which denotes the recurrence requirements upon which most Markovian analysis approaches are based.

The developed DST analysis and software-centric BET-compliance theorems allowed as set of four software engineering design rules to be stated, where by following these pragmatically easily measurable and real-world implementable enables the resulting run-time system to maintain BET-compliance. The development of these rules, set on solid DST foundations, as they allow for a more usable understanding of why and when software system performance predictability is lost and/or gained. Importantly, these rules and their underlying DST formalization act as "lines-in-the-sand" in terms of clearly delineating when and why a system may or may not admit

statistically predictable run-time performance measures. Moreover, key and critical insights became available via the developed novel theory analysis.

More particularly, via the combined QN, EMM, and DST analysis software-centric systems could be shown to exits as far more complex dynamical systems than are generally studies within more traditional domains such as Physics and Mechanical Engineering. In particular, software-centric systems were shown to be able to shift back and forth across the mathematical "line-in-the-sand" that denotes BET-complaint systems in response to variations in their run-time conditions, deployment environments, and time varying operational characteristics. Hence, unlike more classical DST regimes, software-centric system analyses require considering the families of possible QNs that can arise for a given software-system instantiation regime, or nominally fixed set of executable code, where this itself is one-to-many with the many not being easily a priori predictable. That fair OS scheduling and resource contention and utilization issues both enact stochastic scatterings of the DST event space is also a novel and problematic aspects of software system behaviors. Such considerations are not required in the analysis of more classical DST problem domains. These theory based observations an insights provide formal foundational arguments as to why the engineering of software-centric systems such that they assuredly behave predictably under all likely real-world feasible scenarios, inclusive of those caused by intelligent adversaries, has proven to be an on-going hard open challenge particularly at the large scales and complexities commonplace to many modern operationally deployed systems. In general, most within most other engineering domains BET-compliance arises as a consequence of conservation laws applying, as first identified via Noether's theorem [256], whereas this research has formally shown and highlighted that a core part of why the engineering of real-world operational software systems is hard is that they can easily cross back and forth between being BET-compliant and non-compliant. This places such systems as being far removed from and far more behaviorally complex than those studied in nearly all other engineering domains.

This important connection between DST and software system run-time performance analysis has only been very lightly explored previously, with no prior works seeking to apply DST in combination with traditional QN modeling so as to formally prove when BET-compliance does or does not hold, inclusive of where and why compliance is lost. Hence, solidifying this connection and formal theory underpinnings to scale and complexity agnostic statistical run-time performance predictability is am important novel contribution to software engineering as an engineering discipline.

Moreover, it should be noted that by following the developed software engineering design rules, since the run-time software system then becomes BET-compliant, prior works and approaches implicitly or explicitly based on requiring BET-compliance to hold now become fully applicable. Additionally, by run-time monitoring the software engineering design rule adherence it can be quickly and easily assessed when statistical predictability has been lost again as well as to usefully quantify the degree to which it has been lost. This allows for example systems to be built that ensure and retain quality of service predictability for high valued and high-cost customers while allowing lower tiered customers to trade-off run-time predictability for accrued cost-savings.

8.3 Future Work

Modern societies are increasingly critically dependent on software-centric systems across many core domains spanning critical control and management, banking and finance, social media and entertainment, eVoting and eGovernment, and advaned healthcare and eHealth, eCommerce, etc. Such systems are rapidly growing in their deployment scales and complexities, with emerging areas of Smart Cities and Buildings, Smart Grids, autonomous vehicles, etc. adding significant human and environmental health and safety concerns and unfolding regulatory landscapes. Hence, it can be expected that modern socities will increasingly require and demand that such large-scale complex software systems be formally engineered to behave predictably within all real-world likely operational scenarios, inclusive of active targeting by intelligent and, potentially, nation-state adversaries.

Within most engineering domains, ergodicity is a fundamental precept of building real-world operational that behave predictably, i.e., ergodicity is a foundation precept within the mathematical models used to design, analyze, and engineer such systems. For software-centric systems, this can be seen in the widespread implicit or explicit presumptions made within Markovian-based software system run-time performance assessment and prediction academic literature that the studied systems are necessarily ergodic. Similarly, BET-compliance is generally presumed to hold within Monte Carlo based simulation studies. Generally, these works make such ergodic presumptions without formally proving that such assumptions will continue to hold within real-world operational contexts. Hence, this becomes a potential explanation of the ongoing industry-academic bifurcation in terms of whether or not run-time software-

centric system behaviors are predictable. The available academic literature strongly suggests they are whereas industries' direct experience shows that they are not, where again the underlying issue is one of industry commonly developing, operating, and maintaining systems of scales and complexities that are orders of magnitude largely than those amenable to most academic research contexts.

By utilizing Peter Walter's Ergodic Equivalency theorem in concert with the extended version of Maurer's computing model, we have been able to derive more general conditions for BET-compliance to hold within software-centric systems of arbitrary scales and complexities. Hence, the resulting work and developed formal DST foundation have applicability to a wide set of possible future research directions and contributions. The following subsections briefly highlight some of these potential areas and directions.

8.3.1 Extensibility of Prior Academic Literature and Insights

As stated previously, in general the prior Markovian works have been based on assuming ergodicity holds and a consequence of recurrence theorems, as required to support standard matrix-based Markov chain-based solution approaches. This leads to open challenges in terms of how to prove recurrence holds within arbitrarily large-scale and complex software systems, an issue we sidestepped by making use of the mathematically equivalence of Theorem 3's Statements 2 and 4. Importantly, though although Theorem 3 Statements 3 denotes a recurrence requirement, it is a requirement that weak recurrence holds and not the stronger recurrence required by the matrix-based Markov chain-based solution approaches. In both cases, any $x \in \Omega$ can be reached via the iterative application of $T : \Omega \rightarrow \Omega$, i.e., recurrence holds. The distinction though is whether each such x can be reached (or passed through) at least once or an infinite number of times, where Markov solution approach generally presuppose the latter.

Our intuition is that the Chapter 5 theory most likely results in all $x \in \Omega$ being passed through an infinite number of times *almost always* with the iterative application of $T[.]$. But, it remains an area of future work to actually develop a formal proof that this is the case. If we can develop such a proof, then all of the prior Markovian based research works on software system run-time behavioral predictability then become fully applicable to describe observed operational real-world system behaviors when the developed four software engineering design rules are met and actively run-

time managed for under. Hence, our work is complementary to this past rich body of literature in that it provides, via the developed software engineering design rules, the run-time mechanisms required to ensure that the real-world industry operational systems admit the prior literature's developed solutions, while also enabling easy measurements of when a run0-time system has left the operational regimes covered by the prior literature.

Moreover, as per the Chapter 7 assessments, current industry auto-scaling and RTOS-based hypervisor approaches appear to be insufficient to fully bring cloud deployed software systems back to being BET-compliant. Hence, a nearer term future research direction is to seek to develop, verify, and experimentally vet an the viability of an optimal auto-scaling solution sufficient to bring cloud-deployed software systems back to BET-compliance at minimal resourcing costs. If and where this is not possible, then the goal would be to develop formal bounds on the family of producible cdf for BET non-compliant scenarios. Ideally, this would include developing formal constraints on the possible shapes of producible operational cdfs, where such bounds may be in terms of confidence intervals on the underlying statistical distribution of producible cdf families. Such future directions though require substantive additional formal analyses past those provided in Chapter 5.

8.3.2 Applications of Control Theory for Cloud Elastics Services

A emerging area of software engineering interest is in the application of formal control theory approaches to the development of optimal and responsive cloud elastic services and auto-scaling solutions. Most generally, the goal of control theory is to move a system from an unwanted operational state $x \in \Omega$ into some desired state $x' \in \Omega$. This directly leads to the question as to whether a path (or control rule) exists by which the system can be moved from operational state x to the desired x'. Ergodicity is important in control theory as recurrence provides the formal guarantee that all such paths exists for any x and x', i.e., no wandering sets of non-zero measure exist into which the system could get trapped and unable to get out of.

Importantly, the Chapter 5 DST analysis highlights that larger-scale software-centric systems, particularly cloud-deployed systems, do not exist as the singular fixed systems common to most other engineering domains. Instead, as resources are added to and removed from the system, so as to maintain performance levels while

minimizing costs, a relatively broad and complex dynamic and time-variant family of software systems results. All system within the family service the same input-to-output functional needs by they do so with potentially very different resource allocations and within potentially very different run-time deployment environments.

The derived software engineering design rules of Chapter 5 though can be used to ensure all members of the produced family of the given software-centric system retain BET-compliance. Hence, paths from any x to x' are guaranteed to exist, but the control rule required to transition along each path changes as the active member of the potential family of instantiated run-time systems changes. Given the nature of software-centric systems, it would seem feasible and possible to make run-time observations of the currently instantiated family members such that these required dynamic and time-varying control rules could be generated and enacted sufficiently quickly to enable and allow for the overall control of the system as a whole. Achieving and formalizing this though would require substantial additional research and developed understanding. Hence, it has been left as an areas of potential future research.

8.3.3 Applications to Emerging Domains of Smart Cities, Buildings, Grids and Autonomous Vehicles

Smart cities, buildings, and grids as well as autonomous vehicles are all rapidly emerging areas where software-centric systems coincide with the need for formally controlled systems so as to manage and mitigate the inherent human health and safety risks. As per the Chapter 5 analyzes, academic research simulation studies of such systems are highly likely to be within BET-compliant regimes, whereas real-world operational-scale deployments of such systems are highly unlikely to be. Moreover, such systems innately will tend to require hard real-time observe, orient, decide, and act (OODA) processes[293] due to their safety critical nature. This in turn requires that the the systems necessarily maintain BET-compliance in their operation. Hence, this works developed theory, insights, and observations developed may have a core role to play within these emerging areas.

8.3.4 Applications of AI/ML Approaches to Software System Prediction and Control

As discussed in earlier Chapters, research works are beginning to appears that seek to apply modern machine learning (ML) and artificial intelligence (AI) approaches to the problem of run-time prediction and management (or control) of software-system behaviors. AI/ML approaches that are based on training data consumption and analyses though inherently make the presumption that the training data contains information that is representative of both the test data and, subsequent, operational data. Moreover, the need to address cases where this does hold has lead to the emergence of the concern over what is being termed "concept drift" within the modern AI/ML communities[294; 295]. It can be observed though that the well-developed mathematically foundation provided via ergodic theory and, specifically, BET provide a rich set of already existing and mature mathematical tools by which to identify, address, and, potentially, mitigate such issues, or at least identify when re-training must occur.

As such, the Chapter 5 derived software engineering design rules can be seen as also identifying where and when AI/ML approaches are likely to be successful in modeling and predicting run-time software-centric system behaviors. Moreover, they are likely also usable to provide useful measures of when the AI/ML system "tracking" is degrading and becoming lost such that the AI/ML approach requires re-training. In particular, the scope of the generated wandering sets is quite easily measured as in the scope of how event completions are being stochastically "shattered" forward in time. Hence, there are important future applications primarily of the Chapter 5 DST and analysis to how to most appropriately and successfully apply AI/ML approach to the run-time control and management of software-centric systems.

8.3.5 Relevance to Quantum Computing

Clearly, quantum computing is also a rapidly emerging field. At this point though more work would be required to assess whether or not the Chapter 5 σ-finite measure space based DST-formalism continue to apply to quantum computing systems, or whether such system are more appropriately described via dynamical systems defined over infinite measure spaces. The issue is important as BET (Theorem 1) and Peter Walter's Ergodic Equivalence Theorem (Theorem 3) are both defined with respect to DST on σ-finite measure spaces. Within infinite measure space, ergodicity is a more

complex concept which broaden out into a multiplicity of ways in which a dynamical system on an infinite measure space may or may not be ergodic, i.e., for infinite measure space ergodicity itself then arises in multiple "favors".

Currently, many quantum computing theory analyses appear to be based on σ-finite measure space ergodicty presumptions, but significant future work would be required to assess the degree to which the Chapter 5 developed results may (or may not) apply to quantum computing based systems.

8.3.6 Relevance to Cyber-security Issues and Concerns

Finally, the presented work and research also have applications to issues and concerns within cyber-security areas as detecting malicious events, activities, and/or intrusions within BET non-compliant normal operating regimes is itself an extremely challenging proposition. Moreover, a core approach to the development of anomaly detection based cyber-security methods is to developed well-functioning models of normal behaviors and then detect attacks and, potentially, novel attacks as deviations from normal. Such approaches follow on from quite roughly similar approaches used within classical fault detection and diagnosis.

Such approaches though are predicated on relatively well-behaved or well-modellable normal behaviors, where the formal theory, insights, and observations of this research can be applied to assess where and when these are likely to occur and when they are not. Hence, as with prior Markovian-based run-time performance prediction works, the discussed research may offer path by which the performance of existing cyber-security solution approaches may be improved via driving BET-compliant normal systems behaviors. This may be particularly useful for cyber-security approaches seeking to exploit AI/ML solution approaches as a consequence of the issues discussed in Section 8.3.4 above.

8.4 Summary

This Chapter provided a brief review of the research work and outcomes provided in this dissertation, inclusive of the motivations for the research directions and undertakings. A summary of the core novel contribution produced by the research were then provided. Finally, a set of potential future research directions and emerging re-

search areas for which the dissertation results may have applicability were presented and discussed.

Bibliography

[1] S. F. Piragha. (2016) Containercloudsim: An environment for modeling and simulation of containers in cloud data centers. [Online]. Available: http://www.cloudbus.org/cloudsim/container.html

[2] J. McKim. (2016) Abstracting the back-end with faas. [Online]. Available: https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362

[3] A. Adas, "Traffic models in broadband networks," *IEEE communications Magazine*, vol. 35, no. 7, pp. 82–89, 1997.

[4] R. O'Dwyer, "Analysis and load testing of a real-world cloud deployed distributed system," Master's thesis, University of Victoria, 2016.

[5] M. E. A. Elgamal, "The extended maurer model: Bridging turing-reducibility and measure theory to jointly reason about malware and its detection," Ph.D. dissertation, 2014.

[6] Docker. (2017) Docker engine. [Online]. Available: https://www.docker.com/

[7] google, "Manage a cluster of linux containers as a single system to accelerate dev and simplify ops," 2021.

[8] Knative. Knative, year = 2022, url = https://knative.dev/docs/.

[9] E. Ataie, E. Gianniti, D. Ardagna, and A. Movaghar, "A combined analytical modeling machine learning approach for performance prediction of mapreduce jobs in cloud environment," in *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2016, pp. 431–439.

[10] Y. Zhang, Z. Zheng, and M. R. Lyu, "Real-time performance prediction for cloud components," in *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, 2012, pp. 106–111.

[11] P. Saripalli, G. Kiran, R. R. Shankar, H. Narware, and N. Bindal, "Load prediction and hot spot detection models for autonomic cloud computing," in *2011 fourth IEEE international conference on utility and cloud computing*. IEEE, 2011, pp. 397–402.

[12] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin, "Predicting performance anomalies in software systems at run-time," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[13] R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl, "Integrating run-time observations and design component models for cloud system analysis," 2014.

[14] A. Keshavarzi, A. Toroghi Haghighat, and M. Bohlouli, "Online qos prediction in the cloud environments using hybrid time-series data mining approach," *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, vol. 45, pp. 461–478, 2021.

[15] C. Yin, Z. Xiong, H. Chen, J. Wang, D. Cooper, and B. David, "A literature survey on smart cities." *Sci. China Inf. Sci.*, vol. 58, no. 10, pp. 1–18, 2015.

[16] M. Batty, K. W. Axhausen, F. Giannotti, A. Pozdnoukhov, A. Bazzani, M. Wachowicz, G. Ouzounis, and Y. Portugali, "Smart cities of the future," *The European Physical Journal Special Topics*, vol. 214, pp. 481–518, 2012.

[17] Y. Ji, "Application of fault detection using distributed sensors in smart cities," *Physical Communication*, vol. 46, p. 101182, 2021.

[18] H. Jiang, J. J. Zhang, W. Gao, and Z. Wu, "Fault detection, identification, and location in smart grid based on data-driven computational methods," *IEEE Transactions on Smart Grid*, vol. 5, no. 6, pp. 2947–2956, 2014.

[19] A. E. L. Rivas and T. Abrao, "Faults in smart grid systems: Monitoring, detection and classification," *Electric Power Systems Research*, vol. 189, p. 106602, 2020.

[20] C. A. Andresen, B. N. Torsæter, H. Haugdal, and K. Uhlen, "Fault detection and prediction in smart grids," in *2018 IEEE 9th International Workshop on Applied Measurements for Power Systems (AMPS)*. IEEE, 2018, pp. 1–6.

[21] A. M. Elbir, B. Soner, S. Çöleri, D. Gündüz, and M. Bennis, "Federated learning in vehicular networks," in *2022 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. IEEE, 2022, pp. 72–77.

[22] J. Almeida, J. Rufino, M. Alam, and J. Ferreira, "A survey on fault tolerance techniques for wireless vehicular networks," *Electronics*, vol. 8, no. 11, p. 1358, 2019.

[23] C. Sommer and F. Dressler, *Vehicular networking*. Cambridge University Press, 2014.

[24] H. Ye, L. Liang, G. Y. Li, J. Kim, L. Lu, and M. Wu, "Machine learning for vehicular networks: Recent advances and application examples," *ieee vehicular technology magazine*, vol. 13, no. 2, pp. 94–101, 2018.

[25] V. S. Sharma and K. S. Trivedi, "Quantifying software performance, reliability and security: An architecture-based approach," *Journal of Systems and Software*, vol. 80, no. 4, pp. 493–509, 2007.

[26] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[27] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, and A. Koziolek, *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.

[28] G. A. Spedicato, "Discrete time markov chains with r." *R J.*, vol. 9, no. 2, p. 84, 2017.

[29] M. Gribaudo, M. Iacono, and M. Kiran, "A performance modeling framework for lambda architecture based applications," *Future Generation Computer Systems*, vol. 86, pp. 1032–1041, 2018.

[30] Z. Nikdel, B. Gao, and S. W. Neville, "Dockersim: Full-stack simulation of container-based software-as-a-service (saas) cloud deployments and environments," in *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 2017, pp. 1–6.

[31] omnet. Networksimulator, year = 2022, url = https://omnetpp.org/.

[32] C. Z. Mooney, *Monte carlo simulation*. Sage, 1997, no. 116.

[33] R. M. Gray and R. Gray, *Probability, random processes, and ergodic properties*. Springer, 2009, vol. 1.

[34] G. Apostolopoulos and C. Hassapis, "V-em: A cluster of virtual machines for robust, detailed, and high-performance network emulation," in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*. IEEE, 2006, pp. 117–126.

[35] V. Looga, Z. Ou, Y. Deng, and A. Ylä-Jääski, "Mammoth: A massive-scale emulation platform for internet of things," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 3. IEEE, 2012, pp. 1235–1239.

[36] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *2014 IFIP Networking Conference*. IEEE, 2014, pp. 1–9.

[37] J. Hasenburg, M. Grambow, E. Grünewald, S. Huk, and D. Bermbach, "Mockfog: Emulating fog computing infrastructure in the cloud," in *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2019, pp. 144–152.

[38] A. company. (2023) Simplified solutions for network performance testing: Network emulation & traffic generation. [Online]. Available: https://www.apposite-tech.com/?creative=643786993670&keyword=applicationperformancetest&matchtype=p&network=g&device=c&source=googlead&medium=cpc

[39] Amazon. (2017) Aws. [Online]. Available: https://aws.amazon.com/

[40] Google. Load testing best practices, year = 2023, url = https://cloud.google.com/run/docs/about-load-testing.

[41] IBM. Distributed load testing on aws, year = 2023, url = https://www.ibm.com/docs/en/rpt/9.2.0?topic=overview-rational-performance-tester.

[42] AWS. Rational performance tester overview, year = 2023, url = https://aws.amazon.com/solutions/implementations/distributed-load-testing-on-aws/.

[43] W. E. Lewis, *Software testing and continuous quality improvement*. CRC press, 2017.

[44] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 85–103.

[45] A. Filieri, M. Maggio, K. Angelopoulos, N. d'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein *et al.*, "Software engineering meets control theory," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015, pp. 71–82.

[46] W.-P. Wang, D. Tipper, and S. Banerjee, "A simple approximation for modeling nonstationary queues," in *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, vol. 1. IEEE, 1996, pp. 255–262.

[47] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 341–350.

[48] J. Young and T. Barth, "Akamai online retail performance report: Milliseconds are critical," 2017.

[49] W. Stadnik and Z. Nowak, "The impact of web pages' load time on the conversion rate of an e-commerce platform," in *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology–ISAT 2017: Part I*. Springer, 2018, pp. 336–345.

[50] D. Di Fatta, D. Patton, and G. Viglia, "The determinants of conversion rates in sme e-commerce websites," *Journal of Retailing and Consumer Services*, vol. 41, pp. 161–168, 2018.

[51] B. Elliott, "Anything is possible: Managing feature creep in an innovation rich environment," in *2007 IEEE International Engineering Management Conference*. IEEE, 2007, pp. 304–307.

[52] F. D. Davis and V. Venkatesh, "Toward preprototype user acceptance testing of new information systems: implications for software project management," *IEEE Transactions on Engineering management*, vol. 51, no. 1, pp. 31–46, 2004.

[53] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2011, pp. 104–113.

[54] A. Keshavarzi, A. Toroghi Haghighat, and M. Bohlouli, "Enhanced time-aware qos prediction in multi-cloud: a hybrid k-medoids and lazy learning approach (qopc)," *Computing*, vol. 102, no. 4, pp. 923–949, 2020.

[55] Y. Pan, S. Wang, L. Wu, Y. Xia, W. Zheng, S. Pang, Z. Zeng, P. Chen, and Y. Li, "A novel approach to scheduling workflows upon cloud resources with fluctuating performance," *Mobile Networks and Applications*, vol. 25, pp. 690–700, 2020.

[56] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, pp. 1–23, 2016.

[57] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed systems*, vol. 22, no. 6, pp. 931–945, 2011.

[58] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[59] D. Cerotti, M. Gribaudo, P. Piazzolla, and G. Serazzi, "Flexible cpu provisioning in clouds: A new source of performance unpredictability," in *2012 Ninth International Conference on Quantitative Evaluation of Systems*. IEEE, 2012, pp. 230–237.

[60] Alibaba. Alibaba cluster trace program, year = 2023, url = https://github.com/alibaba/clusterdata/.

[61] CNBC. Aws outage, year = 2021, url = https://www.cnbc.com/2021/12/10/aws-explains-outage-and-will-make-it-easier-to-track-future-ones.html.

[62] IBM. Coldstart, year = 2022, url = https://developer.ibm.com/articles/reducing-cold-start-times-in-knative/.

[63] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 289–302.

[64] S.-M. Park and M. Humphrey, "Predictable high-performance computing using feedback control and admission control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 3, pp. 396–411, 2010.

[65] H. Khojasteh and J. Mišić, "Task admission control policy in cloud server pools based on task arrival dynamics," *Wireless Communications and Mobile Computing*, vol. 16, no. 11, pp. 1363–1376, 2016.

[66] N. Leontiou, D. Dechouniotis, and S. Denazis, "Adaptive admission control of distributed cloud services," in *2010 International Conference on Network and Service Management*. IEEE, 2010, pp. 318–321.

[67] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 204–212.

[68] Y. He, J. Huang, Q. Duan, Z. Xiong, J. Lv, and Y. Liu, "A novel admission control model in cloud computing," *arXiv preprint arXiv:1401.4716*, 2014.

[69] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE, 2007, pp. 27–27.

[70] K. Ye, Y. Kou, C. Lu, Y. Wang, and C.-Z. Xu, "Modeling application performance in docker containers using machine learning techniques," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 1–6.

[71] J. Kumar and A. K. Singh, "Workload prediction in cloud using artificial neural network and adaptive differential evolution," *Future Generation Computer Systems*, vol. 81, pp. 41–52, 2018.

[72] W. Zhang, B. Li, D. Zhao, F. Gong, and Q. Lu, "Workload prediction for cloud cluster using a recurrent neural network," in *2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*. IEEE, 2016, pp. 104–109.

[73] Q. Zhang, L. T. Yang, Z. Yan, Z. Chen, and P. Li, "An efficient deep learning model to predict cloud workload for industry informatics," *IEEE transactions on industrial informatics*, vol. 14, no. 7, pp. 3170–3178, 2018.

[74] Y. Yu, V. Jindal, F. Bastani, F. Li, and I.-L. Yen, "Improving the smartness of cloud management via machine learning based workload prediction," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2018, pp. 38–44.

[75] P. Nawrocki and P. Osypanka, "Cloud resource demand prediction using machine learning in the context of qos parameters," *Journal of Grid Computing*, vol. 19, no. 2, pp. 1–20, 2021.

[76] H. Moradi, W. Wang, and D. Zhu, "Online performance modeling and prediction for single-vm applications in multi-tenant clouds," *IEEE Transactions on Cloud Computing*, 2021.

[77] H. Khazaei, J. Misic, and V. B. Mi˘sic, "Performance analysis of cloud centers under burst arrivals and total rejection policy," in *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011*. IEEE, 2011, pp. 1–6.

[78] V. Goswami, S. S. Patra, and G. B. Mund, "Performance analysis of cloud with queue-dependent virtual machines," in *2012 1st International Conference on Recent Advances in Information Technology (RAIT)*. IEEE, 2012, pp. 357–362.

[79] H. Khazaei, N. Mahmoudi, C. Barna, and M. Litoiu, "Performance modeling of microservice platforms," *IEEE Transactions on Cloud Computing*, 2020.

[80] H. Khazaei, J. Mišić, V. B. Mišić, and S. Rashwand, "Analysis of a pool management scheme for cloud computing centers," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 5, pp. 849–861, 2012.

[81] X. Liu, W. Tong, X. Zhi, F. ZhiRen, and L. WenZhao, "Performance analysis of cloud computing services considering resources sharing among virtual machines," *The Journal of Supercomputing*, vol. 69, no. 1, pp. 357–374, 2014.

[82] W. Ellens, J. Akkerboom, R. Litjens, H. Van Den Berg *et al.*, "Performance of cloud computing centers with multiple priority classes," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 245–252.

[83] H. Khazaei, J. Mišić, and V. B. Mišić, "Performance analysis of cloud computing centers," in *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer, 2010, pp. 251–264.

[84] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[85] J. Beda, "Containers at scale: the google cloud platform and beyond," 2015.

[86] R. Zhou, Z. Li, and C. Wu, "Scheduling frameworks for cloud container services," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 436–450, 2018.

[87] S. R. Poojara, V. B. Ghule, M. N. Birje, and N. V. Dharwadkar, "Performance analysis of linux container and hypervisor for application deployment on clouds," in *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*. IEEE, 2018, pp. 24–29.

[88] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *2016 international conference on computing, communication and automation (iccca)*. IEEE, 2016, pp. 1204–1210.

[89] H. A. Akpan and B. Vadhanam, "A survey on quality of service in cloud computing," *International Journal of Computer Trends and Technology*, vol. 27, no. 1, pp. 58–63, 2015.

[90] Y. O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady, "Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis," in *2010 IEEE 3rd International Conference on Cloud Computing*. Ieee, 2010, pp. 91–98.

[91] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[92] J. Anselmi, E. Amaldi, and P. Cremonesi, "Service consolidation with end-to-end response time constraints," in *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. IEEE, 2008, pp. 345–352.

[93] G. Baranwal and D. P. Vidyarthi, "Admission control in cloud computing using game theory," *The Journal of Supercomputing*, vol. 72, no. 1, pp. 317–346, 2016.

[94] B. Khargharia, S. Hariri, F. Szidarovszky, M. Houri, H. El-Rewini, S. U. Khan, I. Ahmad, and M. S. Yousif, "Autonomic power & performance management for large-scale data centers," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[95] S. U. Khan and C. Ardil, "Energy efficient resource allocation in distributed computing systems," in *International conference on distributed, high-performance and grid computing*, 2009, pp. 667–673.

[96] A. Nezarat and G. Dastghaibifard, "Efficient nash equilibrium resource allocation based on game theory mechanism in cloud computing by using auction," *PloS one*, vol. 10, no. 10, p. e0138424, 2015.

[97] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan *et al.*, "Ultra-large-scale systems: The software challenge of the future," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 2006.

[98] Amazon. Aws service-level-agreements, year = 2023, url = https://aws.amazon.com/legal/service-level-agreements/?aws-sla-cards.sort-by=item.additionalFields.serviceNameLower&aws-sla-cards.sort-order=asc&awsf.tech-category-filter=*all.

[99] J. C. Doyle, B. A. Francis, and A. R. Tannenbaum, *Feedback control theory*. Courier Corporation, 2013.

[100] K. Xiong and H. Perros, "Service performance and analysis in cloud computing," in *2009 Congress on Services-I*. IEEE, 2009, pp. 693–700.

[101] J. Vilaplana, F. Solsona, I. Teixidó, J. Mateo, F. Abella, and J. Rius, "A queuing theory model for cloud computing," *The Journal of Supercomputing*, vol. 69, pp. 492–507, 2014.

[102] Y. Tay and R. Suri, "Error bounds for performance prediction in queuing networks," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 3, pp. 227–254, 1985.

[103] D. Arcelli, "Exploiting queuing networks to model and assess the performance of self-adaptive software systems: a survey," *Procedia Computer Science*, vol. 170, pp. 498–505, 2020.

[104] S. Balsamo, V. D. N. Personè, and P. Inverardi, "A review on queueing network models with finite capacity queues for software architectures performance prediction," *Performance Evaluation*, vol. 51, no. 2-4, pp. 269–288, 2003.

[105] M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple qos constrained scheduling strategy of multiple workflows for cloud computing," in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2009, pp. 629–634.

[106] A. Gambi, G. Toffetti, and S. Comai, "Model-driven web engineering performance prediction with layered queue networks," in *ICWE Workshops*, 2010, pp. 25–36.

[107] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[108] J.-B. Durand and O. Gaudoin, "Software reliability modelling and prediction with hidden markov chains," *Statistical Modelling*, vol. 5, no. 1, pp. 75–93, 2005.

[109] T. Ahmad, D. Truscan, and I. Porres, "Identifying worst-case user scenarios for performance testing of web applications using markov-chain workload models," *Future Generation Computer Systems*, vol. 87, pp. 910–920, 2018.

[110] D. G. Feitelson, "Workload modeling for performance evaluation," in *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*. Springer, 2002, pp. 114–141.

[111] ——, *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[112] M. Calzarossa and G. Serazzi, "Workload characterization: A survey," *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1136–1150, 1993.

[113] M. C. Calzarossa, L. Massari, and D. Tessera, "Workload characterization: A survey revisited," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–43, 2016.

[114] W. Cirne and F. Berman, "A comprehensive model of the supercomputer workload," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 140–148.

[115] M. F. Arlitt and C. L. Williamson, "Internet web servers: Workload characterization and performance implications," *IEEE/ACM Transactions on networking*, vol. 5, no. 5, pp. 631–645, 1997.

[116] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 2007, pp. 171–180.

[117] G. Cloud. Containers in cloud, year = 2023, url = https://cloud.google.com/containers.

[118] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[119] M. Van Steen and A. S. Tanenbaum, *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[120] M. Kumar, S. C. Sharma, A. Goel, and S. P. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *Journal of Network and Computer Applications*, vol. 143, pp. 1–33, 2019.

[121] A. Arunarani, D. Manjula, and V. Sugumaran, "Task scheduling techniques in cloud computing: A literature survey," *Future Generation Computer Systems*, vol. 91, pp. 407–415, 2019.

[122] N. Kasenides and N. Paspallis, "A systematic mapping study of mmog backend architectures," *Information*, vol. 10, no. 9, p. 264, 2019.

[123] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[124] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[125] Google. (2017) app engine. [Online]. Available: https://appengine.google.com/

[126] G. cloud. (2022) Java 11/17 runtime environment. [Online]. Available: https://cloud.google.com/appengine/docs/standard/java-gen2/runtime

[127] R. Koty and B. Kannan. (2017) The rise of container-as-a-service (caas) for faster application delivery. [Online]. Available: https://blog.equinix.com/blog/author/ramchandra-koty-and-balasubramaniyan-kannan/

[128] Flexera. (2022) Flexera 2022 state of the cloud report. [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud?id=ELQ-Redirect

[129] Amazon. (2022) Containers at aws. [Online]. Available: https://aws.amazon.com/containers/

[130] S. Fotuhi Piraghaj, "Energy-efficient management of resources in container-based clouds," Ph.D. dissertation, 2016.

[131] Amazon. (2022) Category deep dive containers. [Online]. Available: https://aws.amazon.com/getting-started/deep-dive-containers/#:

~:text=Containers%20provide%20a%20standard%20way,consistent%20deployments%2C%20regardless%20of%20environment.

[132] opencontainers. (2015) About the open container initiative. [Online]. Available: https://opencontainers.org/about/overview/

[133] R. P. Padhy. (2018) Serverless architecture: A detailed comparison of faas platforms. [Online]. Available: https://www.linkedin.com/pulse/serverless-architecture-detailed-comparison-faas-prasad-padhy/

[134] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.

[135] B. Javadi, J. Abawajy, and R. Buyya, "Failure-aware resource provisioning for hybrid cloud infrastructure," *Journal of parallel and distributed computing*, vol. 72, no. 10, pp. 1318–1331, 2012.

[136] S. Alnajdi, M. Dogan, and E. Al-Qahtani, "A survey on resource allocation in cloud computing," *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, vol. 6, no. 5, 2016.

[137] A. Belgacem, "Dynamic resource allocation in cloud computing: analysis and taxonomies," *Computing*, vol. 104, no. 3, pp. 681–710, 2022.

[138] H. A. Kholidy, "An intelligent swarm based prediction approach for predicting cloud computing user resource needs," *Computer Communications*, vol. 151, pp. 133–144, 2020.

[139] L. M. Haji, S. Zeebaree, O. M. Ahmed, A. B. Sallow, K. Jacksi, and R. R. Zeabri, "Dynamic resource allocation for distributed systems and cloud computing," *TEST Engineering & Management*, vol. 83, no. May/June 2020, pp. 22 417–22 426, 2020.

[140] Y. Zhang, J. Yao, and H. Guan, "Intelligent cloud resource management with deep reinforcement learning," *IEEE Cloud Computing*, vol. 4, no. 6, pp. 60–69, 2017.

[141] D. Ferrari, "Workload charaterization and selection in computer performance measurement," *Computer*, vol. 5, no. 4, pp. 18–24, 1972.

[142] M. C. Calzarossa, M. L. Della Vedova, L. Massari, D. Petcu, M. I. Tabash, and D. Tessera, "Workloads in the clouds," in *Principles of Performance and Reliability Modeling and Evaluation*. Springer, 2016, pp. 525–550.

[143] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998, pp. 151–160.

[144] A. B. Downey and D. G. Feitelson, "The elusive goal of workload characterization," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 4, pp. 14–29, 1999.

[145] M. Calzarossa, L. Massari, and D. Tessera, "Workload characterization issues and methodologies," in *Performance Evaluation: Origins and Directions*. Springer, 2000, pp. 459–482.

[146] M. Ghorbani, Y. Wang, Y. Xue, M. Pedram, and P. Bogdan, "Prediction and control of bursty cloud workloads: a fractal framework," in *Proceedings of the 2014 international conference on hardware/software codesign and system synthesis*, 2014, pp. 1–9.

[147] M. E. Crovella, M. Harchol-Balter, and C. D. Murta, "Task assignment in a distributed system (extended abstract) improving performance by unbalancing load," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998, pp. 268–269.

[148] W. Leland and T. J. Ott, "Load-balancing heuristics and process behavior," in *Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation*, 1986, pp. 54–69.

[149] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: Evidence and possible causes," *IEEE/ACM Transactions on networking*, vol. 5, no. 6, pp. 835–846, 1997.

[150] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 3, pp. 253–285, 1997.

[151] V. Paxson, "Empirically derived analytic models of wide-area tcp connections," *IEEE/ACM transactions on Networking*, vol. 2, no. 4, pp. 316–336, 1994.

[152] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson, "Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level," *IEEE/ACM Transactions on networking*, vol. 5, no. 1, pp. 71–86, 1997.

[153] M. E. Crovella, "Performance evaluation with heavy tailed distributions," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2000, pp. 1–9.

[154] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "Towards understanding cloud performance tradeoffs using statistical workload analysis and replay," *University of California at Berkeley, Technical Report No. UCB/EECS-2010-81*, 2010.

[155] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[156] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 2010, pp. 87–92.

[157] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini, "Statistical profiling-based techniques for effective power provisioning in data centers," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 317–330.

[158] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2007, pp. 119–128.

[159] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari, "Server workload analysis for power minimization using consolidation," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009, pp. 28–28.

[160] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak, "A capacity management service for resource pools," in *Proceedings of the 5th international workshop on software and performance*, 2005, pp. 229–237.

[161] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services." in *NSDI*, vol. 8, 2008, pp. 337–350.

[162] P. Wang, H. Wang, and W. Wang, "Finding semantics in time series," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 385–396.

[163] V. S. Frost and B. Melamed, "Traffic modeling for telecommunications networks," *IEEE Communications Magazine*, vol. 32, no. 3, pp. 70–81, 1994.

[164] D. Krishnamurthy, J. A. Rolia, and S. Majumdar, "A synthetic workload generation technique for stress testing session-based systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, p. 868, 2006.

[165] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," *IEEE Transactions on Cloud Computing*, vol. 2, no. 2, pp. 208–221, 2014.

[166] D.-C. Juan, L. Li, H.-K. Peng, D. Marculescu, and C. Faloutsos, "Beyond poisson: Modeling inter-arrival time of requests in a datacenter," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2014, pp. 198–209.

[167] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 1287–1294.

[168] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011, pp. 500–507.

[169] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE transactions on cloud computing*, vol. 3, no. 4, pp. 449–458, 2014.

[170] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[171] J. Yang, C. Liu, Y. Shang, B. Cheng, Z. Mao, C. Liu, L. Niu, and J. Chen, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Information Systems Frontiers*, vol. 16, no. 1, pp. 7–18, 2014.

[172] Y.-C. Chang, R.-S. Chang, and F.-W. Chuang, "A predictive method for workload forecasting in the cloud environment," in *Advanced Technologies, Embedded and Multimedia for Human-Centric Computing*. Springer, 2014, pp. 577–585.

[173] M. S. Yoon, A. E. Kamal, and Z. Zhu, "Adaptive data center activation with user request prediction," *Computer Networks*, vol. 122, pp. 191–204, 2017.

[174] V. Debusschere, S. Bacha *et al.*, "Hourly server workload forecasting up to 168 hours ahead using seasonal arima model," in *2012 IEEE international conference on industrial technology*. IEEE, 2012, pp. 1127–1131.

[175] C. Liu, C. Liu, Y. Shang, S. Chen, B. Cheng, and J. Chen, "An adaptive prediction approach based on workload pattern discrimination in the cloud," *Journal of Network and Computer Applications*, vol. 80, pp. 35–44, 2017.

[176] M. C. Calzarossa, M. L. Della Vedova, L. Massari, G. Nebbione, and D. Tessera, "Modeling and predicting dynamics of heterogeneous workloads for cloud environments," in *2019 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2019, pp. 1–7.

[177] P. Pruthi and A. Erramilli, "Heavy-tailed on/off source behavior and self-similar traffic," in *Proceedings IEEE International Conference on Communications ICC'95*, vol. 1. IEEE, 1995, pp. 445–450.

[178] J. Yin, X. Lu, H. Chen, X. Zhao, and N. N. Xiong, "System resource utilization analysis and prediction for cloud based applications under bursty workloads," *Information Sciences*, vol. 279, pp. 338–357, 2014.

[179] S. Pacheco-Sanchez, G. Casale, B. Scotney, S. McClean, G. Parr, and S. Dawson, "Markovian workload characterization for qos prediction in the cloud," in *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011, pp. 147–154.

[180] S. Chen, M. Ghorbani, Y. Wang, P. Bogdan, and M. Pedram, "Trace-based analysis and prediction of cloud computing user behavior using the fractal mod-

eling technique," in *2014 IEEE International Congress on Big Data*. IEEE, 2014, pp. 733–739.

[181] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 5, pp. 936–943, 2011.

[182] R. Urgaonkar, U. C. Kozat, K. Igarashi, and M. J. Neely, "Dynamic resource allocation and power management in virtualized data centers," in *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE, 2010, pp. 479–486.

[183] Z. Feldman, M. Masin, A. N. Tantawi, D. Arroyo, and M. Steinder, "Using approximate dynamic programming to optimize admission control in cloud computing environment," in *Proceedings of the 2011 Winter Simulation Conference (WSC)*. IEEE, 2011, pp. 3153–3164.

[184] Q. Duan, "Cloud service performance evaluation: status, challenges, and opportunities–a survey from the system modeling perspective," *Digital Communications and Networks*, vol. 3, no. 2, pp. 101–111, 2017.

[185] R. Ghosh, F. Longo, V. K. Naik, and K. S. Trivedi, "Modeling and performance analysis of large scale iaas clouds," *Future generation computer systems*, vol. 29, no. 5, pp. 1216–1234, 2013.

[186] H. Khazaei, J. Misic, and V. B. Misic, "A fine-grained performance model of cloud computing centers," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 11, pp. 2138–2147, 2012.

[187] Y. Xia, M. Zhou, X. Luo, Q. Zhu, J. Li, and Y. Huang, "Stochastic modeling and quality evaluation of infrastructure-as-a-service clouds," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 1, pp. 162–170, 2013.

[188] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: modeling techniques and their applications," *Journal of Internet Services and Applications*, vol. 5, no. 1, pp. 1–17, 2014.

[189] V. Stantchev and C. Schröpfer, "Negotiating and enforcing qos and slas in grid and cloud computing," in *International Conference on Grid and Pervasive Computing*. Springer, 2009, pp. 25–35.

[190] Amazon. (2017) Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region. [Online]. Available: https://aws.amazon.com/message/41926/

[191] Z. Yuchao, D. Bo, and P. Fuyang, "An adaptive qos-aware cloud," in *2012 International Conference on Cloud Computing Technologies, Applications and Management (ICCCTAM)*. IEEE, 2012, pp. 160–163.

[192] D. Petcu, G. Macariu, S. Panica, and C. Crăciun, "Portable cloud applications—from theory to practice," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1417–1430, 2013.

[193] M. H. Ghahramani, M. Zhou, and C. T. Hon, "Toward cloud computing qos architecture: Analysis of cloud systems and cloud services," *IEEE/CAA Journal of Automatica Sinica*, vol. 4, no. 1, pp. 6–18, 2017.

[194] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[195] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, 2009, pp. 13–18.

[196] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, 2005, pp. 217–228.

[197] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," in *International Workshop on Quality of Service*. Springer, 2003, pp. 381–398.

[198] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, "Dynamic virtual clusters in a grid site manager," in *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*. IEEE, 2003, pp. 90–100.

[199] S.-M. Park and M. Humphrey, "Feedback-controlled resource sharing for predictable escience," in *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008, pp. 1–12.

[200] P. Ruth, P. McGachey, and D. Xu, "Viocluster: Virtualization for dynamic computational domains," in *2005 IEEE International Conference on Cluster Computing*. IEEE, 2005, pp. 1–10.

[201] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 43–52.

[202] M. D. De Assunção, A. Di Costanzo, and R. Buyya, "Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, 2009, pp. 141–150.

[203] M. Mazzucco, D. Dyachuk, and R. Deters, "Maximizing cloud providers' revenues via energy aware allocation policies," in *2010 IEEE 3rd international conference on cloud computing*. IEEE, 2010, pp. 131–138.

[204] I. Goiri, J. Guitart, and J. Torres, "Characterizing cloud federation for enhancing providers' profit," in *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 2010, pp. 123–130.

[205] F. Chang, J. Ren, and R. Viswanathan, "Optimal resource allocation in clouds," in *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 2010, pp. 418–425.

[206] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2011, pp. 1–12.

[207] M. M. Hassan, B. Song, M. S. Hossain, and A. Alamri, "Qos-aware resource provisioning for big data processing in cloud computing environment," in *2014 International Conference on Computational Science and Computational Intelligence*, vol. 2. IEEE, 2014, pp. 107–112.

[208] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 94–103.

[209] P. C. Hershey, S. Rao, C. B. Silio, and A. Narayan, "System of systems for quality-of-service observation and response in cloud computing environments," *IEEE Systems Journal*, vol. 9, no. 1, pp. 212–222, 2014.

[210] M. Salama and A. Shawish, "A qos-oriented inter-cloud federation framework," in *2014 IEEE 38Th Annual Computer Software And Applications Conference*. IEEE, 2014, pp. 642–643.

[211] W. C.-C. Chu, C.-T. Yang, C.-W. Lu, C.-H. Chang, N.-L. Hsueh, T.-C. Hsu, and S. Hung, "An approach of quality of service assurance for enterprise cloud computing (qosaecc)," in *2014 International Conference on Trustworthy Systems and their Applications*. IEEE, 2014, pp. 7–13.

[212] R. Karim, C. Ding, and A. Miri, "An end-to-end qos mapping approach for cloud service selection," in *2013 IEEE ninth world congress on services*. IEEE, 2013, pp. 341–348.

[213] S.-Y. Lee, D. Tang, T. Chen, and W. C.-C. Chu, "A qos assurance middleware model for enterprise cloud computing," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. IEEE, 2012, pp. 322–327.

[214] B. Liao, J. Yu, H. Sun, and M. Nian, "A qos-aware dynamic data replica deletion strategy for distributed storage systems under cloud computing environments," in *2012 Second International Conference on Cloud and Green Computing*. IEEE, 2012, pp. 219–225.

[215] P. Zhang and Z. Yan, "A qos-aware system for mobile cloud computing," in *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*. IEEE, 2011, pp. 518–522.

[216] Y. Xiao, C. Lin, Y. Jiang, X. Chu, and X. Shen, "Reputation-based qos provisioning in cloud computing via dirichlet multinomial model," in *2010 IEEE International Conference on Communications*. IEEE, 2010, pp. 1–5.

[217] P. Billingsley, *Ergodic Theory and Information*. New York: John Wiley & Sons, Inc., 1965, illustrations, tables, Price 64 s.

[218] D. V. Dagaev, "Towards developing of oberon system with specific requirements of ergodicity," *Proceedings of the Institute for System Programming of the RAS*, vol. 32, no. 6, pp. 67–78, 2020.

[219] A. Kopyltsov, "The application of the theory of dynamic systems to software quality estimation," in *Journal of Physics: Conference Series*, vol. 1399, no. 3. IOP Publishing, 2019, p. 033016.

[220] H. C. Tijms, *Stochastic models: an algorithmic approach*. John Wiley & Sons Incorporated, 1994, vol. 303.

[221] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[222] C. Wu and Y. Liu, "Queuing network modeling of driver workload and performance," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, no. 3, pp. 528–537, 2007.

[223] R. G. Gallager, "Discrete stochastic processes," *Journal of the Operational Research Society*, vol. 48, no. 1, pp. 103–103, 1997.

[224] W. K. Grassmann, *Computational probability*. Springer Science & Business Media, 2000, vol. 24.

[225] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, 2010, pp. 87–92.

[226] T. M. Cover, *Elements of information theory*. John Wiley & Sons, 2006.

[227] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-based software performance analysis*. Springer, 2011, vol. 980.

[228] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *The Annals of Mathematical Statistics*, pp. 338–354, 1953.

[229] A. Leon-Garcia, *Probability and random processes for electrical engineering*. Pearson Education India, 1994.

[230] R. O. Onvural, "Survey of closed queueing networks with blocking," *ACM Computing Surveys (CSUR)*, vol. 22, no. 2, pp. 83–121, 1990.

[231] S. Balsamo, "Queueing networks with blocking: Analysis, solution algorithms and properties." *Network performance engineering*, vol. 5233, pp. 233–257, 2011.

[232] R. Bierbooms, I. J. Adan, and M. van Vuuren, "Approximate analysis of single-server tandem queues with finite buffers," *Annals of Operations Research*, vol. 209, pp. 67–84, 2013.

[233] B. D'Auria and S. Kanta, "Pure threshold strategies for a two-node tandem network under partial information," *Operations Research Letters*, vol. 43, no. 5, pp. 467–470, 2015.

[234] S.-S. Ko and R. F. Serfozo, "Response times in m/m/s fork-join networks," *Advances in Applied Probability*, vol. 36, no. 3, pp. 854–871, 2004.

[235] D. Andone and D. Merezeanu, "Tandem queueing systems maximum throughput problem," in *SINTES 10-International Symposium on System Theory*, vol. 1, 2000, p. 4.

[236] N. Gulpinar, P. Harrison, B. Rustem, and L.-F. Pau, "Performance optimization of mean response time in a tandem router network with batch arrivals," in *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. IEEE, 2006, pp. 1–4.

[237] Z. Sahinoglu and S. Tekinay, "On multimedia networks: self-similar traffic and network performance," *IEEE Communications Magazine*, vol. 37, no. 1, pp. 48–52, 1999.

[238] Z. Saffer and W. Yue, "A dual tandem queueing system with gi service time at the first queue," *Journal of Industrial and Management Optimization*, vol. 10, no. 1, pp. 167–192, 2013.

[239] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, "Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 2012, pp. 34–41.

[240] Microsoft. Managing hyper-v hypervisor scheduler types, year = 2023, url = https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types.

[241] Amazon. What is a cloud hypervisor?, year = 2023, url = https://aws.amazon.com/what-is/hypervisor/.

[242] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Benchmarking the performance of microsoft hyper-v server, vmware esxi and xen hypervisors," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 4, no. 12, pp. 922–933, 2013.

[243] J. Kim, A. Dudin, S. Dudin, and C. Kim, "Analysis of a semi-open queueing network with markovian arrival process," *Performance Evaluation*, vol. 120, pp. 1–19, 2018.

[244] C. Kim, S. Dudin, A. Dudin, and K. Samouylov, "Analysis of a semi-open queuing network with a state dependent marked markovian arrival process, customers retrials and impatience," *Mathematics*, vol. 7, no. 8, p. 715, 2019.

[245] H. Pishro-Nik, "Introduction to probability, statistics, and random processes," 2016.

[246] J. Little, "A proof of the theorem l= λ w," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[247] J. Keilson and L. D. Servi, "The distributional form of little's law and the fuhrmann-cooper decomposition," *Operations Research Letters*, vol. 9, no. 4, pp. 239–247, 1990.

[248] S. M. Ross, *Introduction to probability models*. Academic press, 2014.

[249] W. G. Marchal, "An approximate formula for waiting time in single server queues," *AIIE transactions*, vol. 8, no. 4, pp. 473–474, 1976.

[250] J. Medhi, *Stochastic models in queueing theory*. Elsevier, 2002.

[251] V. Gupta, M. Harchol-Balter, J. G. Dai, and B. Zwart, "On the inapproximability of m/g/k: why two moments of job size distribution are not enough," *Queueing Systems*, vol. 64, pp. 5–48, 2010.

[252] A. Kahraman and A. Gosavi, "On the distribution of the number stranded in bulk-arrival, bulk-service queues of the m/g/1 form," *European journal of operational research*, vol. 212, no. 2, pp. 352–360, 2011.

[253] D. V. Lindley, "The theory of queues with a single server," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 48, no. 2. Cambridge University Press, 1952, pp. 277–289.

[254] P. Walters, *An introduction to ergodic theory*. Springer Science & Business Media, 2000, vol. 79.

[255] B. Jahnel and W. König, *Probabilistic Methods in Telecommunications*. Springer, 2020.

[256] E. Noether, "Invariant variation problems," *Transport theory and statistical physics*, vol. 1, no. 3, pp. 186–207, 1971.

[257] AWS. (2024) Aws cloud watch concepts. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html

[258] A. Katok, A. Katok, and B. Hasselblatt, *Introduction to the modern theory of dynamical systems*. Cambridge university press, 1995, no. 54.

[259] W. D. Maurer, "A theory of computer instructions," *Journal of the ACM (JACM)*, vol. 13, no. 2, pp. 226–235, 1966.

[260] L. M. P. van Zelst, "Modeling computer viruses," 2008.

[261] I. Tomek, *The foundations of computer architecture and organization*. WH Freeman & Co., 1990.

[262] D. Hamlet, *Composing Software Components: A Software-testing Perspective*. Springer Science & Business Media, 2010.

[263] P. T. Cox and B. Song, "A formal model for component-based software," in *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No. 01TH8587)*. IEEE, 2001, pp. 304–311.

[264] J. A. Bergstra and C. A. Middelburg, "Simulating turing machines on maurer machines," *Journal of Applied Logic*, vol. 6, no. 1, pp. 1–23, 2008.

[265] A. Turing, "M.(1936) 1965:"on computable numbers-with an application to the entscheidungsproblem"," *Proceedings of the London Mathematical Society*, pp. 1936–37, 1965.

[266] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 1978, pp. 179–196.

[267] "AWS Lambda," https://aws.amazon.com/lambda/, accessed: May 30, 2024.

[268] H. Khazaei, J. Misic, and V. B. Misic, "Modelling of cloud computing centers using m/g/m queues," in *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 2011, pp. 87–92.

[269] H. Poincaré, *The value of science: essential writings of Henri Poincaré*. Modern library, 2012.

[270] A. B. Hajian and S. Kakutani, "Weakly wandering sets and invariant measures," *Transactions of the American Mathematical Society*, vol. 110, no. 1, pp. 136–151, 1964.

[271] S. Müller, "Recurrence for branching markov chains," 2008.

[272] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Leveraging deep learning to improve the performance predictability of cloud microservices," *arXiv preprint arXiv:1905.00968*, 2019.

[273] R. Nillsen, *Randomness and recurrence in dynamical systems: a real analysis approach*. American Mathematical Soc., 2010, vol. 31.

[274] V. András, "The omnet++ discrete event simulation system," in *The European Simulation Multiconference (ESM'2001)*, 2001.

[275] I. Association *et al.*, "Networking and telecommunications: Concepts, methodologies, tools and applications," 2010.

[276] K. Fall, S. Floyd, and T. Henderson, "Ns simulator tests for reno fulltcp," *URL http://www. aciri. org/floyd/papers/fulltcpsims. ps*, 1997.

[277] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[278] "Apache Spark," https://spark.apache.org/, accessed: May 3, 2024.

[279] N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben, "Xen and the art of cluster scheduling," in *First International Workshop on Virtualization Technology in Distributed Computing (VTDC 2006)*. IEEE, 2006, pp. 4–4.

[280] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Proceedings of the ninth ACM international conference on Embedded software*, 2011, pp. 39–48.

[281] "Apache storm," https://storm.apache.org/, accessed: May 3, 2024.

[282] Apache. Apache performance tuning, year = 2024, url = https://httpd.apache.org/docs/2.4/misc/perf-tuning.html.

[283] "Amazon Web Services (AWS) Auto Scaling," https://aws.amazon.com/autoscaling/, accessed: May 30, 2024.

[284] "Google Cloud Platform (GCP) Autoscaler," https://cloud.google.com/compute/docs/autoscaler, accessed: May 30, 2024.

[285] "Microsoft Azure Autoscale," https://azure.microsoft.com/en-us/services/monitor/autoscale/, accessed: May 30, 2024.

[286] "Kubernetes Horizontal Pod Autoscaler (HPA)," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, accessed: May 30, 2024.

[287] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, accessed: May 30, 2024.

[288] "Google Cloud Functions," https://cloud.google.com/functions, accessed: May 30, 2024.

[289] "Datadog," https://www.datadog.com/, accessed: May 30, 2024.

[290] "New Relic," https://newrelic.com/, accessed: May 30, 2024.

[291] AppDynamics, "Appdynamics application performance monitoring," https://www.appdynamics.com, real-time performance monitoring and analytics for web applications.

[292] Xen. Why xen project?, year = 2023, url = https://xenproject.org/users/why-xen/: :text=Amazon

[293] J. R. Boyd *et al.*, *A discourse on winning and losing.* Air University Press Maxwell Air Force Base, AL, 2018, vol. 400.

[294] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine learning*, vol. 23, pp. 69–101, 1996.

[295] L. Poenaru-Olaru, L. Cruz, J. S. Rellermeyer, and A. Van Deursen, "Maintaining and monitoring aiops models against concept drift," in *2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN)*. IEEE, 2023, pp. 98–99.

List of Acronyms

AWS Amazon Web Services, 4

BET Birkhoff Ergodic Theorem, 1

CaaS Container as a Service, 20

cdf cumulative density function, 98

CLT Central Limit Theorem, 12

DTMC Discrete Time Markov chains, 3

DST Dynamical System Theory, 11

EC2 Elastic Compute Cloud, 4

EMM Extended Maurer Model, 11

FaaS Function as a Service, 20

FDI Fault Detection and Identification, 9

GAE Google App Engine, 24

HMM Hidden Markov Model, 40

IaaS Infrastructure as a Service, 6

JRE Java Runtime Environment, 24

LDSS Large-scale Distributed Software Systems, 2

LLN Law of Large Numbers, 12

LP Logical Process, 79

LQN Layered Queuing Network, 6

ML Machine Learning, 92

MMOG Massive Multiplayer Online Games, 1

NIST National Institute of Standards and Technology, 22

OCI Open Container Initiative, 27

OS Operating System, 24

PaaS Platform as a Service, 24

QNM Queuing Network Model, 92

QN Queuing Network, 57

QoE Quality of Experience, 1

QoS Quality of Service, 1

RTDS Real-Time Deferrable Server, 80

RTOS Real-time Operating System, 137

SaaS Software as a Service, 1

SIBPA Swarm Intelligence Based Prediction Approach, 34

SLA Service Level Agreement, 6

SLO Service Level Obligations, 19

VM Virtual Machine, 2