Analysis and Load Testing of a Real-World Cloud Deployed Distributed System

by

Robert O'Dwyer
B.A.Sc., University of British Columbia, 2012

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

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Analysis and Load Testing of a Real-World Cloud Deployed Distributed System

by

Robert O'Dwyer
B.A.Sc., University of British Columbia, 2012

Supervisory Committee

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

Dr. Thomas E. Darcie, Co-Supervisor
(Department of Electrical and Computer Engineering)

ABSTRACT

This thesis uses data from a real-world distributed system to develop a model for realistic load tests, and analyzes the results of several different workload scenarios on a test deployment. The research focused on characterizing the workload of the real-world Pretio system using logs captured from the production deployment, modelling a workload from those logs, and analyzing the impact on a test deployment of the system of a series of scenarios providing different parameters to the model. The results were evaluated by testing the response time distributions across multiple test runs for statistical similarity.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements x

Dedication xi

1 Introduction 1

1.1 Real-world Distributed System 2

1.2 Problem Scope 2

1.3 Problem Statement and Thesis Contributions 3

1.4 Thesis Outline 3

2 Literature Review 4

2.1 Workload Generation in Industry 4

2.2 Flaws in Industry Approaches 5

2.3 Generating Representative Workloads 5

2.4 Chapter Summary 6

3 Methodology 7

3.1 Workload Modeling 7

3.1.1 Characterizing the system's workload 8

3.1.2 The ON-OFF Model 8

3.1.3 Self-similar and Long-Range Dependent workloads 9
3.1.4 Modeling Request Inter-arrivals 9
3.1.5 Testing for exponential distribution 9
3.1.6 Peak load vs. time-dependent load 12
3.1.7 Segmenting the model by request parameters 15
3.1.8 Overall traffic model 15
3.2 System Architecture 16
3.2.1 System Under Test 16
3.2.2 Test Harness 19
3.2.3 Data Collection 19
3.2.4 Deployment 20
3.3 Workload generation 20
3.3.1 Generating scenarios from traffic logs 21
3.3.2 Scenarios and parameters 21
3.3.3 Baseline scenario 22
3.3.4 Increased workload source scenario 22
3.3.5 Increased Duration scenario 22
3.3.6 Scalability scenario 22
3.3.7 Increased Session Length scenario 22
3.3.8 Balancing completeness with testing cost 23
3.4 Analysis Methodology 23
3.4.1 Empirical Latency Distibutions 24
3.4.2 Kolmogorov-Smirnov Test 24
3.4.3 Finding Cliques 26

4 Results **27**
4.1 Result dataset 27
4.1.1 Baseline Traffic Scenario 27
4.1.2 Increased Traffic Source Scenario 28
4.1.3 Increased Duration Traffic Scenario 29
4.1.4 Scalability Traffic Scenario 29
4.1.5 Increased Session Length Traffic Scenario 30
4.2 Distribution of Response Time 31
4.3 Testing Distributions for Similarity 37
4.3.1 Baseline Scenario 37

4.3.2 Increased Traffic Source Scenario 41
4.3.3 Increased Duration Scenario 45
4.3.4 Scalability Scenario 47
4.3.5 Increased Session Length Scenario 49

5 Conclusions **53**
5.1 Evaluation of the Results 53
5.2 Limitations 54
5.3 Future Work 55
5.4 Implications 56

A Tsung Configuration Template **57**

B Load Test Parameters **64**

C Software Used **66**

Bibliography **67**

List of Tables

Table 4.1 Baseline Arrival Rates 27
Table 4.2 Increased Traffic Source Arrival Rates 28
Table 4.3 Increased Duration Arrival Rates 29
Table 4.4 Scalability Arrival Rates 29
Table 4.5 Increased Session Length Arrival Rates 30

Table B.1 Scenario Parameters 64
Table B.2 Session and Event Probabilities 65

List of Figures

Figure 3.1 Distribution of inter-arrival times vs. PDF of exponential distribution 10
Figure 3.2 Calculating the KS test statistic 11
Figure 3.3 Autocorrelation function over a range of time windows 13
Figure 3.4 Visual testing for self-similarity 14
Figure 3.5 Markov chain diagram for a user session (Transition probabilities are an example from Table B.2) 16
Figure 3.6 System Architecture 17
Figure 3.7 System Deployment 21
Figure 3.8 Passing KS-test on two ECDFs 25
Figure 3.9 Example of Graph Representation with a single clique 26

Figure 4.1 Baseline Scenario Arrival Rate 28
Figure 4.2 Increased Traffic Source Scenario Arrival Rate 28
Figure 4.3 Increased Duration Scenario Arrival Rate 29
Figure 4.4 Scalability Scenario Arrival Rate 30
Figure 4.5 Increased Session Length Scenario Arrival Rate 30
Figure 4.6 Baseline Traffic Scenario Response Time Distribution 32
Figure 4.7 Increased Traffic Source Scenario Response Time Distribution . . . 33
Figure 4.8 Increased Duration Scenario Response Time Distribution 34
Figure 4.9 Scalability Scenario Response Time Distribution 35
Figure 4.10 Increased Session Length Scenario Response Time Distribution . . 36
Figure 4.11 Baseline Scenario Cliques 38
Figure 4.12 Baseline Scenario Clique Distributions (KS-test) 39
Figure 4.13 Baseline Scenario Clique Distributions (AD-test) 40
Figure 4.14 Increased Traffic Source Scenario Cliques 42
Figure 4.15 Increased Traffic Source Scenario Clique Distributions (KS-test) . . 43
Figure 4.16 Increased Traffic Source Scenario Clique Distributions (AD-test) . 44
Figure 4.17 Increased Duration Scenario Cliques 46

Figure 4.18 Scalability Scenario Cliques 47
Figure 4.19 Scalability Scenario Clique Distributions (KS-test) 48
Figure 4.20 Scalability Scenario Clique Distributions (AD-test) 48
Figure 4.21 Increased Session Length Scenario Cliques 50
Figure 4.22 Increased Session Length Scenario Clique Distributions (KS-test) . 51
Figure 4.23 Increased Session Length Scenario Clique Distributions (AD-test) . 52

ACKNOWLEDGEMENTS

I would like to thank:

my family and friends, for their continued support and encouragement while working on this project.

the team at Pretio Interactive, for providing the opportunity to test against this system and the time and space to complete my work.

Dr. Stephen Neville, for all of his advice and patience.

DEDICATION

To Sarah, who always believed I could finish this.

Chapter 1

Introduction

Modern distributed systems need to handle much larger loads, deal with unpredictable traffic patterns, and have much more strict requirements for availability than ever before. For many systems, such as web services, both the response time and availability are critical system attributes due to their integration into other companies' systems, often with strict Service-Level Agreements that must be guaranteed.

The effects of poor response times are well known - a study at Amazon in 2007 found a 1% loss of sales with every 100ms of additional page loading time [13]. A study from 2004 on the limits of tolerable waiting time also found that the upper limit for displaying web information to a user is approximately 2 seconds [18]. More recently, many large-scale, heavily marketed video game launches have suffered from availability issues by failing to anticipate or properly test for the influx of players on launch day, such as Pokémon Go [7]. This problem can be compounded by the additional tiers created by businesses that build on top of these services.

With long-running operation, systems may also suffer from memory effects due to complex user behaviours, network effects, the underlying datastore(s), or distributed protocols that cause degraded performance or even downtime. Managing these issues is a difficult task even for a pre-existing system with fixed requirements, but adding new features and continual improvements to a distributed system can cause new, unexpected performance problems as well, particularly when new developers may be unfamiliar with building applications at scale, or unaware of subtle performance issues they introduce.

A realistic and repeatable method of testing the long-running performance of these distributed systems is needed to address these issues. Ideally, such a system would run as automated infrastructure, and regularly assess the performance of a system while highlighting any regressions or failure modes. A parallel can be drawn to the Agile software

development practices of automated testing and continuous integration [16], where software tests are automatically run after code is checked in to source control and before deployments. These automations serve as a valuable quality assurance tool, however, they typically only verify code correctness, and fall short of testing performance or detecting performance regressions.

1.1 Real-world Distributed System

Pretio Interactive provides an web service to serve advertising creatives on demand to its partner companies, mainly publishing networks, mobile applications, and other advertising networks. This system periodically sustains heavy and somewhat unpredictable loads, and has well-defined quality of service (QoS) requirements in terms of response times and availability. The system must be able to choose and serve advertising content to a large audience, and gracefully degrade without losing availability or causing errors. The scope of this thesis is focused specifically on the application of a series of automated load tests on this example distributed system, and the analysis of the results with regard to performance.

Pretio's advertising network is comprised of mobile applications, websites, and other advertising networks, all referred to generically as *publishers*. These publishers integrate directly with the Pretio web service, which provides them with real-time advertisement choices, automatic tracking of user actions that result in revenue, and optimization of the choice of advertisement for a given user. Revenue for publishers is typically calculated as either a percentage share of the total revenue from user engagement with advertisements, or using a fixed rate based on the number of advertisements shown. The advertisements typically click through to a landing page where Pretio will be paid if the user takes a certain action, such as signing up to a service, or downloading a mobile app.

1.2 Problem Scope

Load and performance testing of distributed systems is a complex and error-prone task, and is not currently well handled by automated tools. Repeatable and long-running tests should be run against a system on a regular basis to ensure that performance has not been degraded by new features, and that expectations of QoS and availability are being met. Most simple load-testing and benchmarking tools do not emulate realistic traffic, and are

lacking in the information they return to developers. Standard statistical methodologies can be used to gain deeper insights into the distributions of the relevant metrics of the system such as response time, especially over long-running operation.

1.3 Problem Statement and Thesis Contributions

The goal of this thesis is to develop a model for realistic load testing of a real-world distributed system and analyze the results of several different workload scenarios. The contribution of this thesis can be summarized as follows:

1. Develop a model for simulating the workload based on traffic traces from the real-world distributed system;
2. Analyze the results of running this system against a real-world distributed system, a web service developed by Pretio Interactive;
3. Suggest improvements to the model and ways this work could be used as part of a development team workflow.

1.4 Thesis Outline

- Chapter 2 discusses existing research, best practices and tools currently used in the industry for load-testing of distributed systems, and the areas where they are still lacking.
- Chapter 3 outlines the methodology of testing the system and analyzing the results.
- Chapter 4 presents and analyzes the results of the tests in several different scenarios and configurations of the system under test.
- Chapter 5 summarizes the contribution of this work, outlines potential future work, and offers recommendations as to the use of this research in industry.

Chapter 2

Literature Review

Load testing and simulation of distributed system workloads is an area of active research and industry development. This chapter will detail some of the prior research upon which this thesis builds.

2.1 Workload Generation in Industry

In the web software industry, there are many projects aimed at solving the problems of workload generation and load testing of distributed systems. Commercial services such as BlazeMeter [4] and HP LoadRunner [11] provide geographically and structurally distributed load testing of a web service. Open-source tools such as Apache JMeter [2] and Gatling [15] allow for in-house deployment of load tests, but do not provide the infrastructure to run the tests in a distributed fashion. Although these tools are highly configurable and can produce massive workloads when deployed correctly, they are primarily designed to solve the engineering problem of benchmarking a service or piece of software, not creating a representative workload that is statistically similar to the real workload. They provide summary statistics on the resulting latency and throughput of the system under test, instead of the complete latency distribution or timings of individual requests. These tools all use optimizations to improve their throughput which result in a less realistic modelling of real traffic from end users, such as fixed-size thread pools producing the requests [22]. A notable exception is Tsung [19], another open-source load testing tool which models users explicitly as individual threads with exponentially distributed think times between requests.

2.2 Flaws in Industry Approaches

One issue in standard industry workload generation approaches is raised by the work of Paxson et al. [20], that using a Poisson process to model packet arrival rates fails to accurately reproduce certain kinds of real world workloads. Traffic exhibiting self-similarity will be underestimated due to bursts in the arrival rates that do not average out at larger scales. There is evidence that web traffic can show self-similarity characteristics which are caused by the underlying distributions of user think times, file sizes, popularity of items, and caching [8]. Pruthi et al. [21] showed that these self-similarity effects can be simulated with an aggregate ON/OFF model where one or both of the ON and OFF periods are drawn from a heavy-tailed distribution. To more accurately simulate these behaviours, load test tools can simulate individual user sessions independently (or connections) and model their inter-arrival and think times according to heavy-tailed distributions, instead of an exponential distribution [10]. Many of the simpler benchmarking tools in industry are simply generating a constant rate of requests, but even the ones using an Poisson model are insufficient for realistic modeling of these workloads.

2.3 Generating Representative Workloads

The following papers propose approaches to generating traffic that more accurately reflect the behaviour of users of a distributed system.

An early and influential implementation, SURGE [3], generates load based on aggregate measurements of request properties from real web server logs. It simulates individual user sessions as threads with a ON/OFF model, and can generate self-similar traffic by using a heavy-tailed distribution of the think times between requests.

GEIST [12] is a workload generator that generates aggregate transactional traffic for the purpose of testing web sites and e-commerce services. It models the aggregate traffic based on tunable parameters or log traces, rather than individual users. It is capable of producing self-similar traffic using statistical models of the aggregate request arrival times. However, it does not explore modeling individual user sessions in order to simulate more complex inter-request dependencies.

Modeling inter-request dependencies is explored in the work of Krishnamurthy et al. [14], with a workload generator for e-commerce systems. Using request logs from the system under test, the requests are analyzed to discover dependencies, and a synthetic workload is generated that maintains those dependencies. This methodology is differ-

entiated from previous approaches, which used Markov chains to model the probability of requests being made in each user session. Instead, a subset of sequences containing inter-related requests are selected from the server logs and used directly in the generated workload. They tested this generator against a standardized testbed that simulates a generic e-commerce system (TPC-W), and showed that modeling the inter-request dependencies resulted in a more realistic workload for the system. They also analyzed the impact of varying session length, think times between requests, and request service times on the system. This work is specific to session-based e-commerce workloads, and does not investigate the impact of this kind of load testing against other types of real-world distributed systems. Only the mean and 95th percentile of the system's response times were analyzed, not their full distributions.

In the work of Vögele et al. [23], an intermediate language is proposed for integrating probabilistic workload specifications (e.g. from request logs gathered from real production systems) with models describing the user behaviour and request inter-dependencies. User sessions in [23] are modeled using Markov chains with probabilistic transitions between requests within a session. The resulting workload is tested against a standard benchmark system (SPECjEnterprise2010), but only the properties of the workload are analyzed, not its effect on the system.

2.4 Chapter Summary

A number of tools exist in industry for performing workload generation and load testing of distributed systems, but these tools are not suitable for realistic modeling of traffic. Previous research in this area has investigated various models of workload generation that result in more realistic usage patterns and workloads for a system. These works have presented the results of testing with these more accurate models against real systems. These findings support the motivation of this thesis to produce a more accurate workload model for a real-world distributed system, while analyzing the resulting impact on the system under test. The primary differences between the previous studies and the work presented in this thesis is in the more detailed analysis of the resulting latency distributions, and the specific workload of the real-world Pretio system.

Chapter 3

Methodology

This chapter provides the research methodology of modeling the actual workload of the distributed system under study in order to assess the impacts of different load scenarios. The processes of analyzing the operational logs, characterizing the requests made to the system, and fitting a statistical distribution to the inter-arrival times of particular classes of request are discussed. The process of using the model to perform multiple load test scenarios with different parameters is also discussed. The architecture of the system under test and the load testing apparatus is described, along with the parameters of the workload scenarios that were run.

3.1 Workload Modeling

Workload modeling is an attempt to create a generalized model that can be used to generate synthetic workloads, using measured data from the real system [10]. The primary purpose of workload modeling is to assess performance of a system, but it also can be used to detect failure conditions or optimizations for particular workloads. Although traces from logs can be replayed directly, modeling the workload has several advantages:

1. The model can be adjusted to test for hypothetical scenarios while preserving statistical properties of the workload.

2. A workload with the same statistical properties can be repeated with different initial conditions (e.g. a random seed), allowing multiple distributions of response time and other metrics to be compared instead of a single run.

3. Noise from the system, such as abnormal or changing conditions, can be excluded from the model to avoid corrupting the results of the test.

3.1.1 Characterizing the system's workload

The Pretio system's primary workload consists of HTTP requests that fit into three categories:

Type 1: Requests for an advertisement to show;

Type 2: Event tracking requests that result from a user's actions once the advertisement is displayed;

Type 3: Requests for resources required to display an advertisement, including static images and scripts.

Based on the known usage patterns of this system, Type 1 requests represent a user arriving at the system for an individual session. As shown by Paxson and Floyd [20], the arrival of user-initiated sessions can be modeled as a Poisson process, thus the arrival rates of this type of request are expected to follow a Poisson distribution. Type 2 requests are expected within each session to follow a binomial distribution, where these probabilities are already measured in detail by the Pretio system. A binomial distribution arises because each request is part of a chain where a subsequent event can arise only if its preceding event took place. Type 3 requests are handled separately by an external system, such as a Content Delivery Network, and are thus not considered in this work. As a result, the Pretio per-user workload should be modellable as Poisson distributed Type 1 requests that then trigger binomially distributed Type 2 requests.

3.1.2 The ON-OFF Model

ON-OFF models are commonly used to simulate these forms of user session-based workload [10][3][14]. In this model, each user's session is represented in terms of ON and OFF states, each with its own duration. For modeling independent web traffic where each session starts with the same request, the ON time is a constant which represents the time taken to service the request. If the start of a user session is initiated by the user, The OFF times are expected to be exponentially distributed [20]. Aggregating a large number of these simulated processes results in their inter-arrival times following a Poisson process model.

3.1.3 Self-similar and Long-Range Dependent workloads

If the ON or OFF times (or both) in the aggregate traffic model follow a heavy-tailed distribution, such as Pareto or Weibull, then the resulting traffic will be self-similar [8], where bursts of increased activity appear at many different time scales, and thus appears similar to itself at different resolutions. According to Paxson and Floyd [20], simulations of internet traffic that fail to accurately reproduce self-similarity effects will significantly misestimate network performance of a system under study. A system requires a higher capacity to handle traffic bursts and a realistic load test would need to model these effects accurately to predict whether the system can handle this load. Hence, statistical tests are required to confirm that the Pretio traffic can be accurately represented by a Poisson model, or whether a self-similar model is required.

3.1.4 Modeling Request Inter-arrivals

All requests to the production version of the Pretio system are recorded in web server logs and copied to external storage. Logs for a period of several hours were collected and analyzed to extract the source and arrival time of the request. This data was then used to validate that the user session inter-arrivals conform to a Poisson process. Since the ON and OFF times of events of Poisson processes are exponentially distributed and independent [10], this model was validated by checking the two properties individually. The Kolmogorov-Smirnov Test [17] was used to measure the degree of fit of the data to an exponential distribution with the autocorrelation of the request times assessed to ensure the arrival times were independent.

3.1.5 Testing for exponential distribution

To test that the request inter-arrivals fit a exponential distribution, the request arrival times from the logs were first translated into inter-arrival durations measured in seconds. The distribution of these durations is plotted as a histogram in Figure 3.1, which appears to follow an exponential distribution. The best-fit exponential distribution with a λ parameter of 0.040 (calculated as the mean of the inter-arrivals over the measurement period) is plotted for comparison.

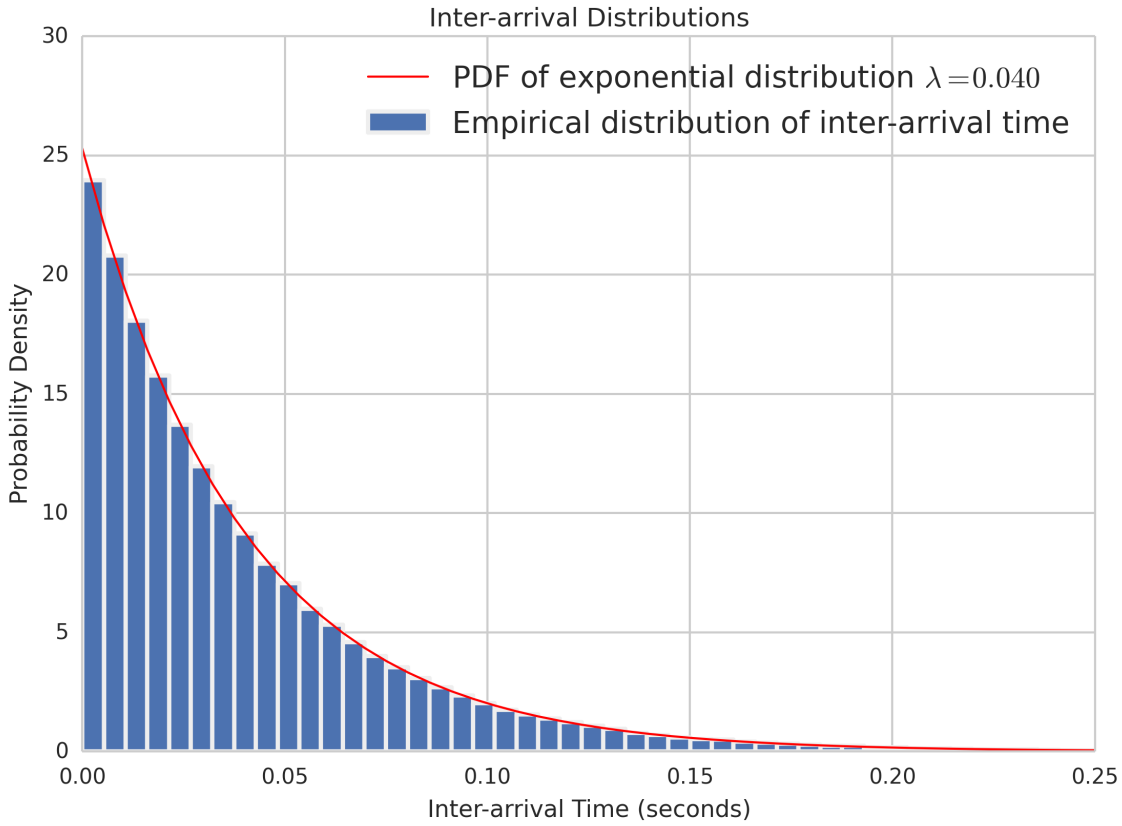


Figure 3.1: Distribution of inter-arrival times vs. PDF of exponential distribution

3.1.5.1 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov goodness-of-fit test was used to compare the two distributions. This involves measuring the maximal absolute difference between the empirical cumulative distribution function from the dataset, and the cumulative distribution function of the theoretical distribution (in this case an exponential distribution based on the mean inter-arrival time), and checking that it falls within a defined confidence level [17].

$$D_n = max\{|F(x) - F_n(x)|\} \tag{3.1}$$

In Equation 3.1, $F(x)$ is the cumulative distribution function for the exponential distribution, and $F_n(x)$ is the empirical cumulative distribution function over the sampled values. The threshold is determined from the desired confidence level α and number of samples N:

$$d_\alpha(N) = \sqrt{\frac{-0.5 \cdot ln\left(\frac{\alpha}{2}\right)}{N}} \tag{3.2}$$

Equation 3.2 was used to determine the minimum threshold required to reject the null hypothesis that the distributions are the same. For a sample size of 1000 inter-arrival times and a confidence level of 0.05, the test statistic D_n must be less than 0.04. The test statistic value of 0.014 was significantly less than the critical value, therefore there is no evidence to suggest the empirical samples were drawn from an underlying distribution that was significantly different than an exponential distribution.

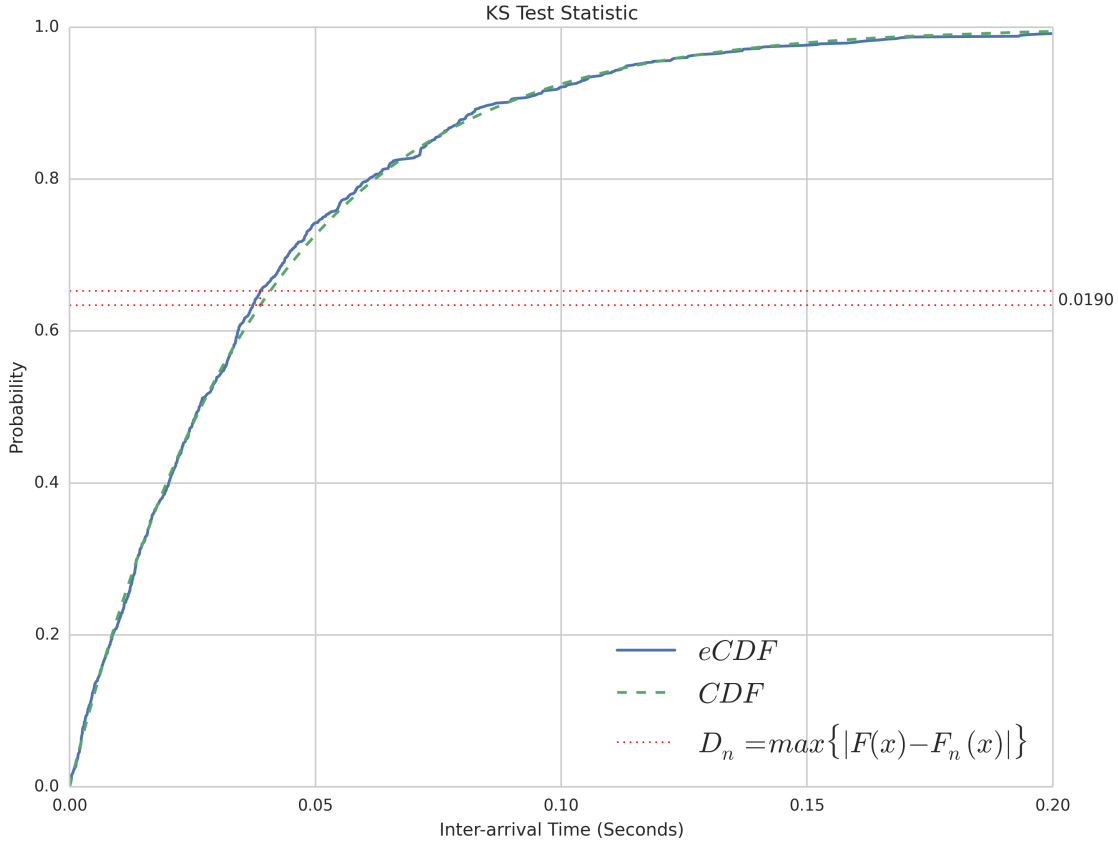


Figure 3.2: Calculating the KS test statistic

3.1.5.2 Autocorrelation test

To check for independence between the arrival rates, the autocorrelation function was calculated over a fixed window size N and a range of time lags k, in seconds, where X_i represents the i_{th} sample in the time series, $\overline{X_{i+k}}$ represents the mean of N samples after

shifting forward in time by k seconds, and σ_{i+k} represents the standard deviation of the shifted samples.

$$R(k) = \frac{\sum_{i=1}^{N}(X_i - \overline{X_i})(X_{i+k} - \overline{X_{i+k}})}{N \cdot \sigma_i \cdot \sigma_{i+k}} \tag{3.3}$$

This function measures the expected correlation of the signal with itself at different points in time, and can be used to find both long-range dependence between requests and repeating patterns such as a daily or hourly cycle. In Figure 3.3, the autocorrelation function is plotted from $k = 0$ up to 30 minutes ($k = 3600$) in sample time across several consecutive 1 hour windows. Multiple windows were analyzed to compensate for the autocorrelation function's sensitivity to non-stationary behaviour.

The result is a mostly flat graph with a peak at $k = 0$, where the lag is zero and the samples are identical. This is consistent with the autocorrelation of a white noise process. In the last chart in Figure 3.3, the same process is applied to samples drawn from a Poisson distribution based on the mean arrival rate, and the resulting autocorrelation graph has the same shape and scale. This indicates that the arrival rates are likely independent, at least over this time window.

3.1.5.3 Visual model testing

Figure 3.4 shows the result of visualizing aggregate arrival rates over longer time periods and lower levels of granularity. For a samples generated by a Poisson process, aggregation over longer time periods is expected to act like a low pass filter operation, smoothing out the traffic. Because the individual requests are independent, the deviations from the mean arrival rate should cancel out on a larger scale [20]. If the request arrival rates were self-similar, the bursts in arrival rate would remain visible at all levels of granularity [10]. The arrival rate is visualized with a bin widths of 1 second, 10 seconds, and 100 seconds. When aggregated over 10 and 100 seconds, the bursts of traffic become progressively less visible, indicating that the requests are likely independent and the arrival rate is not self-similar.

3.1.6 Peak load vs. time-dependent load

A strong time dependence was observed in the logs over longer time periods, and appeared to correspond to predictable daily cycles. Although accurately simulating the system's load over longer periods of time would require this to be part of the model, the focus of the

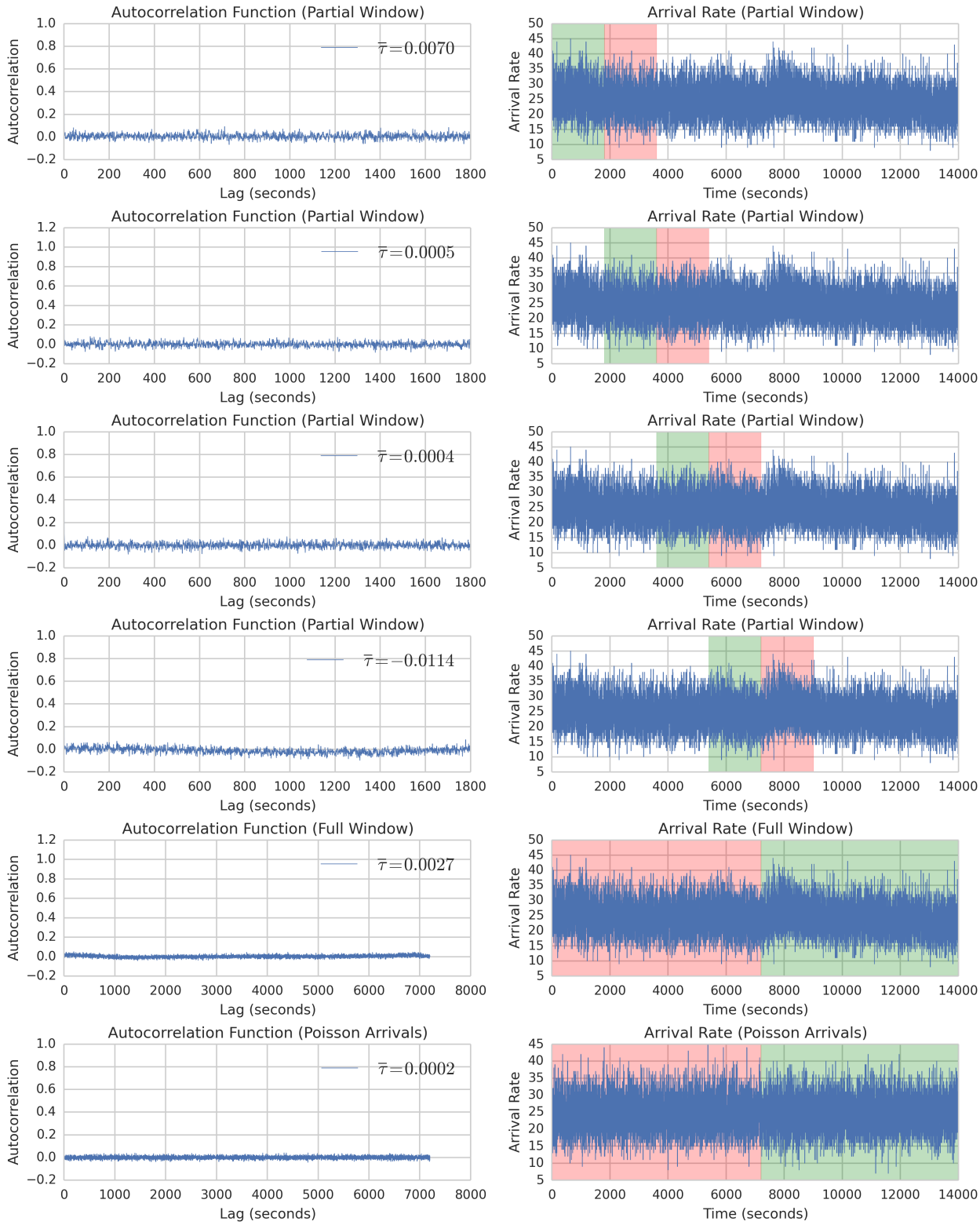


Figure 3.3: Autocorrelation function over a range of time windows

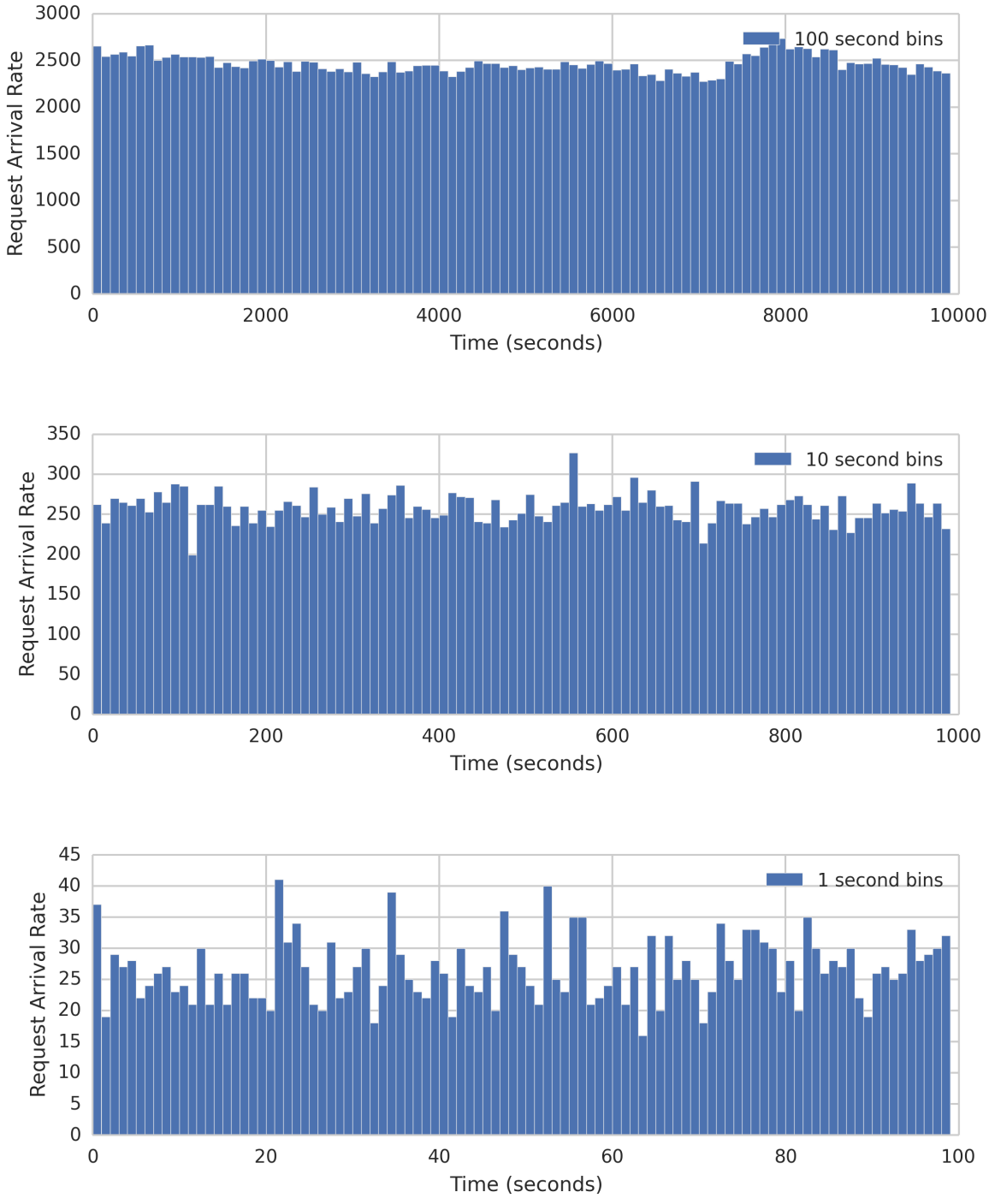


Figure 3.4: Visual testing for self-similarity

research was the system's ability to handle peak load over a sufficiently long test run. The simulation of very long-running scenarios where the effect of daily cycles would become relevant is also likely to be impractical for the business, since it would effectively require doubling the infrastructure costs to continually test a identical version of the system. As a result, the simulation of this property of the system's load is outside the scope of this work. It appears given the regular nature of the daily cycle and the exponential distribution shown in the steady-state request times, it should be possible to model the daily cycles with a piecewise stationary Poisson process, where the arrival rate λ changes as a function of time [10].

3.1.7 Segmenting the model by request parameters

A significant part of the motivation for this research was the desire to model the impact of new sources of traffic (publishers) on the system and assessing the potential impacts of scaling up an existing publisher's traffic. To achieve this, the model was segmented by the unique identifier passed by each publisher. Traffic simulation was done separately on a per-publisher basis. A separate Poisson traffic model was developed for each publisher. This allowed sample users and their workloads to be modeled differently for each publisher, in order to run experiments where the arrival rates could be modified for individual groups.

3.1.8 Overall traffic model

To accurately model the load on the production system, Type 2 traffic must also be simulated. This was done using per-publisher aggregate statistics extracted from Pretio's analytics system, and simulated on a per-user basis. Each session is simulated as a Markov chain, with a state representing a request and subsequent think time. The probability of transitioning to the next state is calculated from the probability of that event occurring in the production system. Each state either transitions to the next state in the chain, or the final state, which ends the session.

The code simulating each user session first makes an initial HTTP request, and then if the conditions for subsequent event requests are met, the next request is made if a Bernoulli trial based on the probability of that event occurring succeeds. If any trial fails, none of the subsequent requests are made, as a failed event breaks the chain of sequential events. The think times between event requests are randomly generated using an exponential distribution, based on aggregate measurements extracted from the analytics system. This method of simulating a user's behaviour follows the Markovian approach

of modelling real people's arrivals to interactive sessions as exponential distributions. A Markov chain diagram showing the state progression of an sample session can be seen in Figure 3.5. Each state represents a request made by the user, with the exception of the start and end states. The transition probabilities in Figure 3.5 are can be found in Table B.2, and are dependent on the publisher (the source of the request).

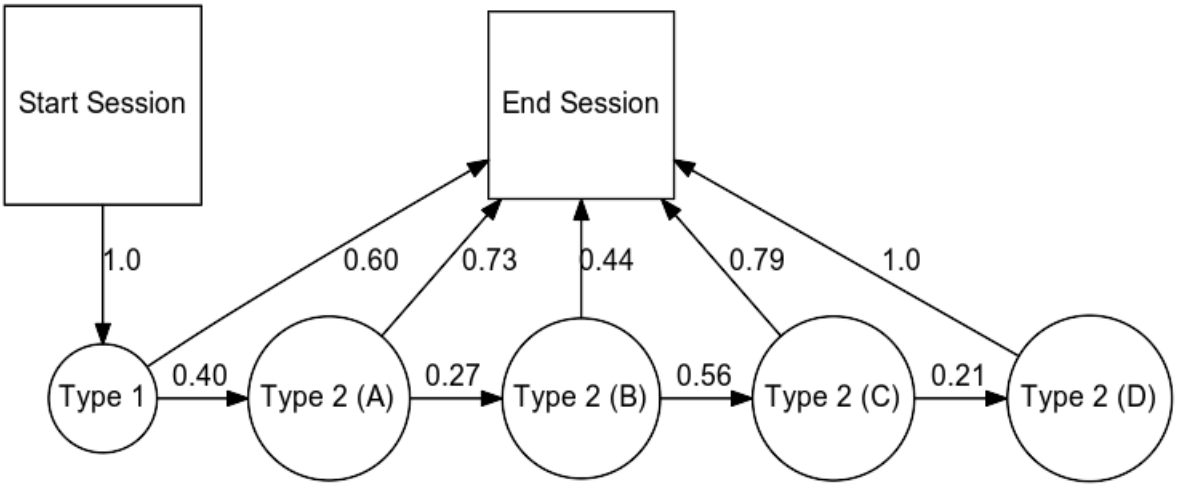


Figure 3.5: Markov chain diagram for a user session (Transition probabilities are an example from Table B.2)

3.2 System Architecture

This section outlines the architecture of the Pretio system and the types of requests it receives, the test harness used to generate the workload, and the data collection system.

3.2.1 System Under Test

The Pretio system consists of a web service served by a load balancer and a pool of web servers (Figure 3.6). These web servers communicate with a primary relational database and a global cache server, both of which run on dedicated machines.

3.2.1.1 Load balancer

The front-end load balancer that handles all inbound HTTP requests is an Elastic Load Balancer, a managed server provided by Amazon Web Services. This service distributes traffic in a round-robin pattern to the back-end application servers. There are small

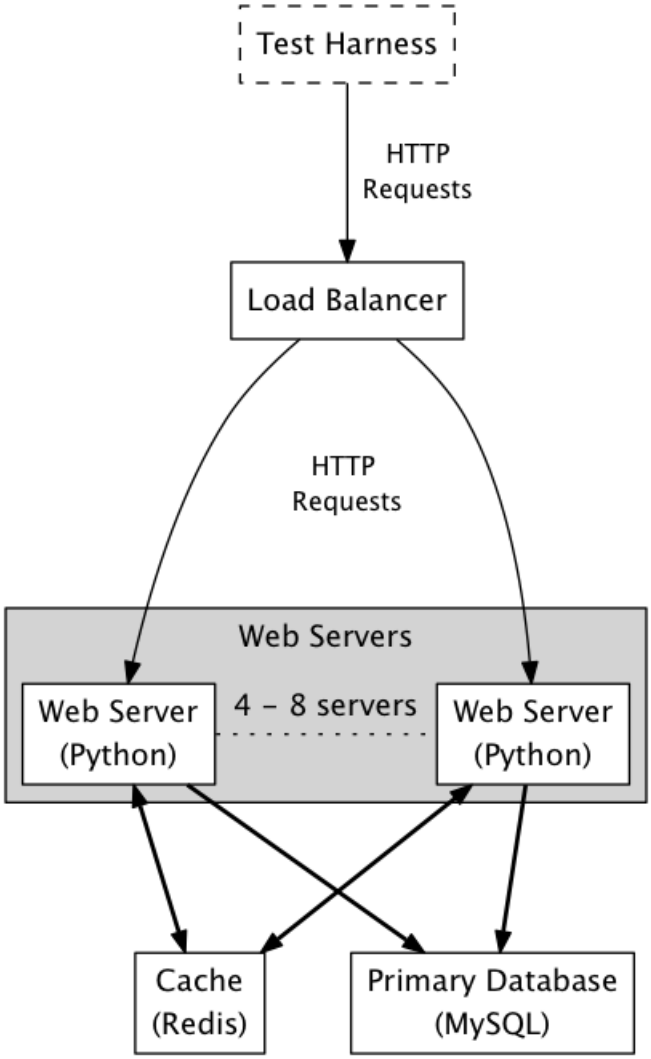


Figure 3.6: System Architecture

configuration differences between this service and the equivalent in the production environment, but for the purposes of this research it is assumed to be identical with respect to performance.

3.2.1.2 Application Servers

The application servers are written in Python. They handle the bulk of the application logic related to parsing requests, choosing advertisements to serve, rendering the correct output for the client and storing event records in the databases. Compared to the other components in the system, this application is much less efficient and is known to be a bottleneck. However, because the servers are stateless and deployed behind the load

balancer, the number of servers can easily be scaled up. This makes their performance more of an efficiency and cost-effectiveness problem, rather than an immediate scalability issue.

3.2.1.3 Database

The primary database is a MySQL server that acts as the write master for several replicas. As much of the heavy read-only queries as possible are delegated to replica servers to avoid unnecessary load, but all of the writes still go through the primary server. This makes it a single point of failure and a contentious resource across the system. Most of the write load involves storing immutable event log records to track user activity, and aggregated counts of those user activities. At some unknown level of load, this component is also suspected to be a bottleneck for request throughput. For the purposes of the load test, only a single primary database server was run, because most of the replica usage only occurs in background processes that are not part of the testing scenarios.

3.2.1.4 Global Cache

A single Redis server is used as a global cache. The results of cached database queries and temporary counters used in ad serving algorithms are stored in the global cache. It also stores several more complex data structures used to optimize ad serving algorithms, such as sorted sets of advertisement identifiers.

3.2.1.5 Request Types

There are two types of requests coming into the system: the ad serving requests, which were modeled as described in Chapter 3 using an exponential distribution of inter-arrival times, and the event tracking requests, which were modeled as a optional session starting after the initial ad serving request, using simple probabilities extracted from the production system's database.

3.2.1.6 Ad Serving API Requests

The ad serving requests are a HTTP GET request with all parameters passed through the query string. The response can be encoded in a number of formats (JSON, HTML, JavaScript) based on different types of integrations and settings, but the same underlying system is used to service the request. The primary task for each request is to select an

ad to display to an individual user, based on past performance and campaign restrictions. This system relies heavily on cached data to operate at a low latency, and as such can be negatively impacted by a large request load with uncached identifiers (e.g. new traffic or incorrect parameters). Due to an architectural limitation in the system, these requests must synchronously write several database records on each request. This makes the performance of this component dependent on the primary database server's load.

3.2.1.7 Event Tracking

The event tracking requests are also simple HTTP GET or POST requests. Each request stores a record in the database, associated with the original identifier provided in the response to an ad request. It also updates counters in the global cache that are used to track performance and enforce rate limiting.

3.2.2 Test Harness

The open-source Tsung distributed load testing tool [19] was used to generate the load for each scenario that was tested. This software was chosen for several reasons. It is high performance and able to horizontally distribute the load generation, due to its use of the Erlang Virtual Machine. It models traffic on a per-session basis with probabilistic arrival and think times, allowing a direct mapping between the model generated for Pretio's production traffic and the actual configuration for the tool. The configuration also allows more complex conditional logic to be expressed, which made it possible to generate an accurate version of the model explained in Section 3.1. Tsung reports latency information for every request instead of the summary statistics reported by most HTTP benchmarking tools, which was a necessary feature for in-depth analysis of the results.

3.2.3 Data Collection

Results were collected from the Tsung load testing tool, which was configured to store local log files with entries for every HTTP request it made during the test. These log files consist of a semicolon-delimited format with the following structure:

```
timestamp;pid;id;http method;host;URL;HTTP status;size;duration;transaction;
    match;error;tag
```

The following fields were used: timestamp, URL, HTTP status, and duration. The URL allowed the requests to be filtered by type. Only the initial advertisement requests were

relevant to the analysis of the system's performance. The other types of requests represent tracking of metrics that do not affect user experience. The HTTP status was checked to ensure that requests in each test were all successful, and that error conditions were not encountered due to misconfiguration or system overload. Finally, the duration and timestamp were used to build a time series of request latency in milliseconds over each individual run. The request latency in these logs was measured from the point of view of the load test client, to provide a measurement of end-to-end response time.

3.2.4 Deployment

The individual components (application servers, primary database, cache) are all deployed on the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) public cloud. Every server used a **m3.medium** instance type (chosen to match the production environment), which has the following specifications:

- 1 CPU (virtualized);
- 3.75 GB RAM;
- 4.0 GB SSD hard disk.

The components were pre-built as images using Docker, a containerization technology, and run using Amazon's EC2 Container Service (ECS) to orchestrate configuring and running the servers for each component. This allowed for rapidly setting up and tearing down the environment between each run to ensure there was no memory between tests. It also provided a reliable way of testing the same configuration locally before deploying to the test environment, because a system of Docker containers can be run and tested locally with only minor differences from the environment they will be deployed to. This matches the deployment configuration used in the real-world production environment.

3.3 Workload generation

This section describes the process for generating the workload scenarios and discusses each scenario and its parameters. The purpose of each scenario is also discussed.

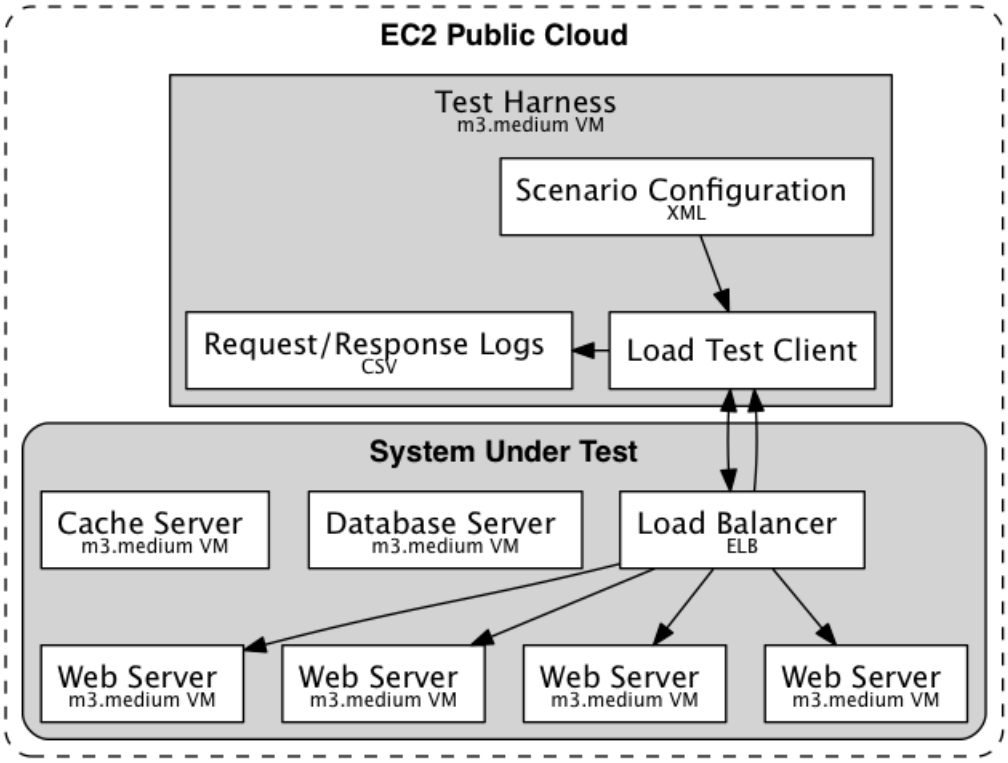


Figure 3.7: System Deployment

3.3.1 Generating scenarios from traffic logs

A simple configuration generator script was written that takes the log files and event probabilities for each step in the simulation, and generates a template of a Tsung configuration file that will run the entire scenario. These templates were modified for specific scenarios, such as increasing the traffic from an individual publisher or extending the run time of the simulation. Each publisher was modeled as a separate type of user session with a fixed set of request parameters and an arrival rate based on the model from Section 3.1. A sample of the output configuration is included in Appendix A.

3.3.2 Scenarios and parameters

The following scenarios were run, after generating a configuration file for the load testing tool and modifying the parameters to fit the particular scenario. Each scenario was run 10 times and a log of every request, the connection time, response time, and total latency in milliseconds, and the HTTP response code was recorded. Before each run, a new set of

servers was provisioned, and a different random seed was provided to the random number generator from which the session arrival times and user think times were retrieved.

3.3.3 Baseline scenario

The first scenario used the unmodified configuration generated from the logs. This served as a ground truth for other simulations, and as a test to ensure that the simulation did not cause errors and generated an appropriate distribution of traffic.

3.3.4 Increased workload source scenario

This scenario involved increasing the arrival rate for the publisher with the largest request volume, while retaining all other traffic at its baseline rates. The purpose of this scenario was to test the impact of planned increases in traffic from a known traffic source.

3.3.5 Increased Duration scenario

This scenario was identical in behaviour to the baseline, but with the run time extended to 5 hours. This time period was chosen so that the resulting single dataset would be an equivalent size to 10 consecutive runs of 30 minutes, like the other scenarios. The purpose was to reveal any potential instability of the system over prolonged steady-state load conditions, and to expose bugs such as memory leaks that could contribute to such behaviours.

3.3.6 Scalability scenario

This scenario scales up both the overall request rate and the number of web servers by a factor of 2. The purpose was to determine whether the system is horizontally scalable to handle at least twice the current traffic level. The hypothesis is that the primary bottleneck at the current traffic level is the performance of the web server code. If performance is limited by the database or cache servers, this scenario should result in poor or inconsistent performance.

3.3.7 Increased Session Length scenario

This scenario modifies the session generation and think times to use a longer average think time of 120 seconds between each event in a user session. The purpose was to

observe the impact on the system when sessions last significantly longer and requests are thus correlated over longer periods of time. The hypothesis was that this change might introduce a limited self-similarity effect in the aggregate request arrival rates, and the resulting bursts of traffic would be more likely to slow down or even overload the system.

3.3.8 Balancing completeness with testing cost

One limitation of the scenarios above is in the number of runs and the duration for each run. Obviously more data and longer test runs would provide greater confidence in the results. An ideal result of this research would be a series of regularly scheduled long-running load tests running nearly continuously to expose issues in the system's performance without manual testing. However, care must be taken to avoid the cost of such testing exploding. Part of the purpose of load testing is to reduce infrastructure costs with effective capacity planning. Running continuous testing against a realistic copy of a production distributed system effectively multiplies the cost, and this must be balanced against the ability to test the system's ability to operate effectively.

3.4 Analysis Methodology

The analysis of the test results consisted of 4 stages:

1. Extract arrival rates and latency of HTTP request service times from logs;
2. Filter each set of samples to remove outliers and other noise;
3. Analyze the distributions of request latency across each scenario;
4. Apply statistical similarity tests to the latency distributions to assess changes in the underlying distributions across different runs.

The first three stages were necessary to obtain the required latency measurements and their distributions, and to evaluate qualitatively whether the QoS requirements were achieved. The filtering stage was used to remove outliers before applying the statistical similarity tests. Since multiple sets of measurements were collected in each test scenario, the resulting distributions were compared against each other in a pairwise fashion using a standard statistical goodness-of-fit test. Passing statistical similarity tests were used to construct the edges in a graph, which was analyzed to find cliques.

3.4.1 Empirical Latency Distibutions

For each test, empirical distributions of request latency were constructed, and visualized as histograms. The shape and properties of these distributions were used to visually assess whether the test successfully achieved the QoS requirements for the Pretio system. The histograms were also used to identify interesting features in the distributions, such as peaks in particular latency values.

3.4.2 Kolmogorov-Smirnov Test

The empirical distributions of response time collected from every run in each scenario were tested pairwise against each other for statistical similarity. Because the underlying distribution of the measurements was not known, statistical tests that assume an underlying distribution could not be used. Two distribution-free tests that could be used are the two sample Kolmogorov-Smirnov test [17] (KS-test) and the Anderson-Darling test [1] (AD-test). The KS-test measures the maximum difference between the two empirical distribution functions and tests whether this value is within a confidence interval. The KS-test was initially chosen because it tests the maximum difference between the cumulative distribution functions, which is relevant when measuring the real-world performance of this system. For example, a significant difference in the median latency is a more important change to highlight than an difference spread across the the latency distribution, and the KS-test would be more likely to fail in the first case. The property that the test statistic can be easily visualized alongside the empirical distribution function (Figure 3.8) is also useful for visually determining why the test failed for a given pair of distributions. However, the AD-test tests the cumulative difference weighted by probability, and thus is more sensitive to differences spread across the latency distribution, especially in the distribution tails. Both tests were performed for each pair of runs, and the results are compared in Chapter 4.

The KS-test is sensitive to outliers in the data, as a single large outlier can dominate the maximal difference between two cumulative distribution functions. To compensate for this, a median filter was used to filter outliers without removing the underlying shape of the distribution. This technique uses a sliding window of samples, and replaces each sample with the median of the surrounding window if it is more than a certain number of standard deviations away from the median. A 1-dimensional window of 20 samples was used, meaning that each sample in the time series was replaced with the median of the window containing itself and the preceding and following samples, if the difference

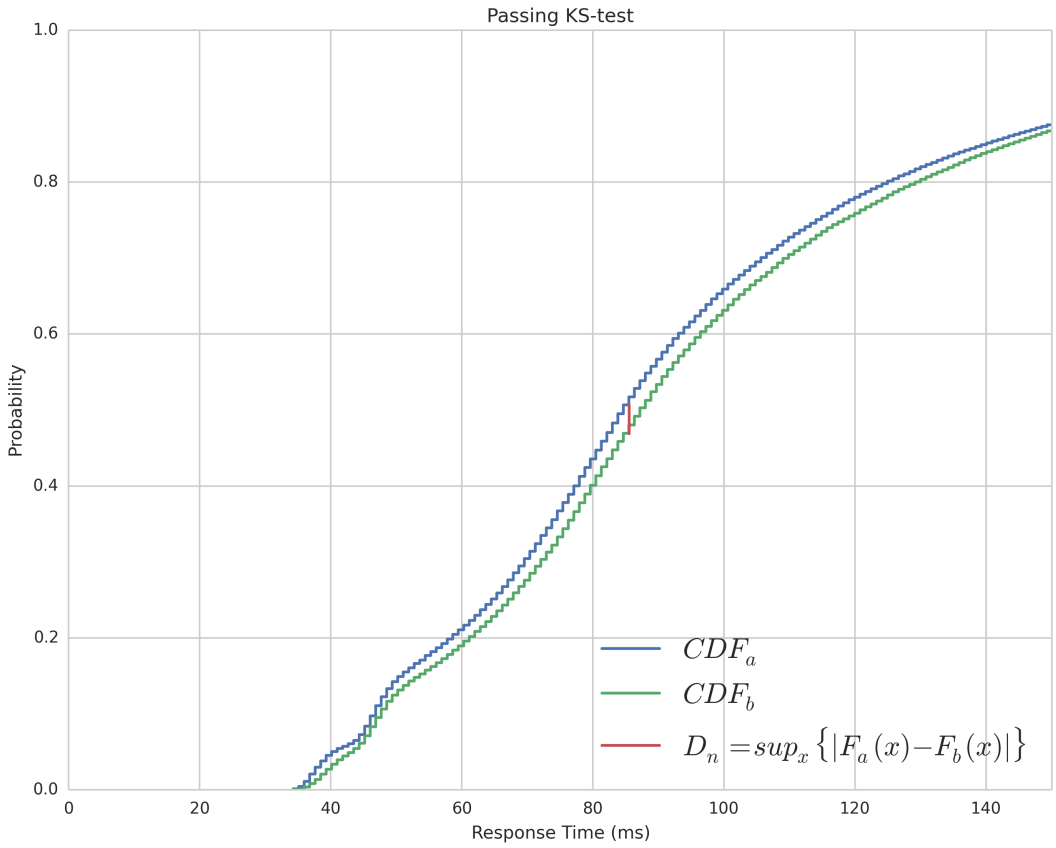


Figure 3.8: Passing KS-test on two ECDFs

between the sample and the median of the window was more than twice the standard deviation.

Since the sampling rate for each test in a pairwise comparison was not equal, the two cumulative distribution functions could not be compared directly. Instead, interpolation was used on the raw samples to produce a smaller set of uniformly distributed samples. These interpolated samples were used as input to the KS-test. This allowed comparing between the two cumulative distribution functions in the KS-test over the same range of latency values. A bin size of 1 millisecond was chosen, because changes in latency below this threshold would not be treated as significantly different in the operation of the production system. For example, failing the statistical similiarity test due to one latency distribution being shifted by less than a millisecond would not be a useful result, as the distributions are still effectively the same. A cubic spline interpolation method was used, which involved taking the empirical CDF and calculating a piecewise polynomial function that could be used to map values in sample space (latency) to an interpolated probability.

The individual samples in the empirical CDFs were then interpolated using this function into a consistent linear domain of 1 millisecond bins.

3.4.3 Finding Cliques

The Kolomogorov-Smirnov and Anderson Darling statistical similarity tests described in Section 3.4.2 were applied to each unique pair of runs in a scenario's ensemble. The results of these tests were used to construct a graph for each scenario, where each run is represented by a node, and the statistical tests that passed at a 95% confidence level were represented by edges between the nodes. Pairs of runs where the statistical similarity test failed were not connected with an edge. A simplified example is provided in Figure 3.9, where the edge between nodes 2 and 3 represent a pass of the statistical similarity test, and the missing edge between nodes 1 and 3 represents a failure.

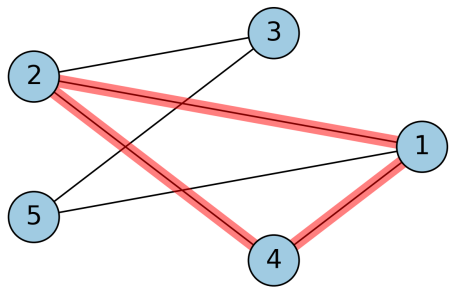


Figure 3.9: Example of Graph Representation with a single clique

To determine whether the system has more than one mode of behaviour in a given scenario, the graphs of the pairwise KS-test results were analyzed to find cliques, defined as sets of completely connected nodes [6]. In Figure 3.9, the single clique containing nodes 1, 2, and 4 is highlighted. Each node represents the set of samples for a single test run, and the edges represent passing KS-tests, indicating that the two sets of samples do not appear to be drawn from different underlying distributions. The Bron-Kerbosch algorithm was used to find all cliques within the graph for each scenario [5]. These cliques can then be interpreted as the sets of runs whose distributions do not appear to be different (in a statistical sense).

Chapter 4

Results

This chapter provides the results of the methodologies described in Chapter 3, described in terms of the response time distributions collected over multiple runs for each scenario. The latency distributions are compared using a statistical similarity test, and cliques of passing distributions are visualized to show some of the potential modes of behaviour under each scenario.

4.1 Result dataset

The logs produced by the Tsung testing tool [19] were analyzed to check the validity of the testing procedure and system configuration. In Figures 4.1 - 4.5, the request rate for a single 30 minute run of each scenario is plotted over time.

4.1.1 Baseline Traffic Scenario

Table 4.1: Baseline Arrival Rates

	Mean Arrival Rate (per second)	Std. Arrival Rate
User Sessions	35.24	5.43
Requests	52.41	6.49

In the baseline traffic scenario, the model derived in Chapter 3 was translated into a Tsung load test configuration using the template in Appendix A. The model parameters from Appendix B were used, which were extracted from the production Pretio system

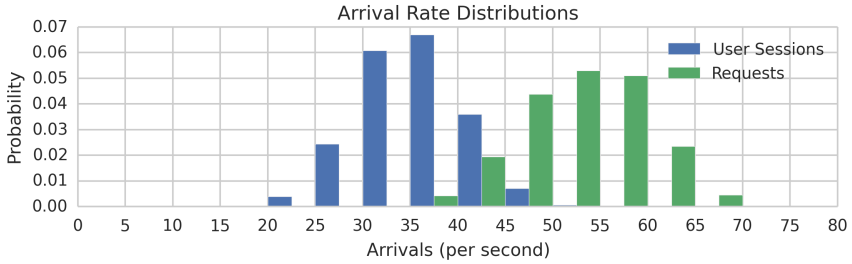


Figure 4.1: Baseline Scenario Arrival Rate

using the logs analyzed in Section 3.1. The resulting arrival rates and their distributions are shown in Table 4.1 and Figure 4.1.

4.1.2 Increased Traffic Source Scenario

Table 4.2: Increased Traffic Source Arrival Rates

	Mean Arrival Rate (per second)	Std. Arrival Rate
User Sessions	47.23	6.75
Requests	69.41	8.49

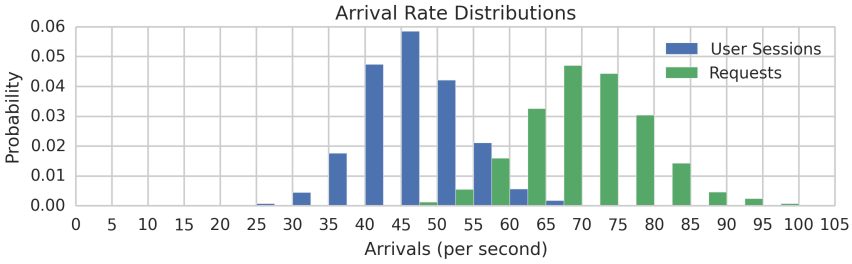


Figure 4.2: Increased Traffic Source Scenario Arrival Rate

In this scenario, the average rate of user session arrivals from the largest traffic source was increased by a factor of 2. The remaining traffic sources remained unmodified from the baseline scenario. This is not the same as increasing the overall traffic level by a factor of 2, as the largest traffic source represents only 34% of the load on the system. The resulting arrival rates and their distributions are shown in Table 4.2 and Figure 4.2.

Table 4.3: Increased Duration Arrival Rates

	Mean Arrival Rate (per second)	Std. Arrival Rate
User Sessions	35.33	6.07
Requests	52.47	7.44

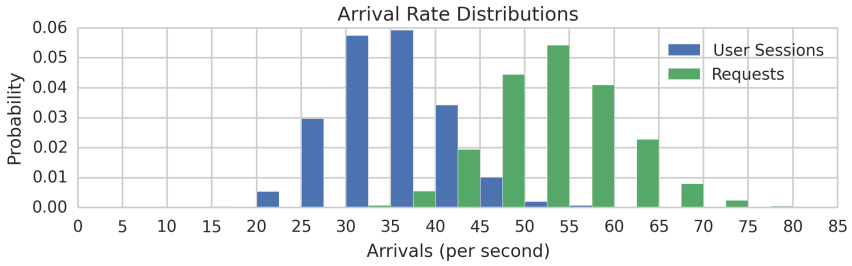


Figure 4.3: Increased Duration Scenario Arrival Rate

4.1.3 Increased Duration Traffic Scenario

In this scenario, a single, much longer load test was performed, lasting 5 hours, and the results were divided into consecutive windows of 30 minutes. Each 30 minute window was treated as an individual run and its latency distribution was compared against the others to assess the stability of the system over time. Other than the longer run time, this scenario was identical to the baseline scenario. The resulting arrival rates and their distributions are shown in Table 4.3 and Figure 4.3.

4.1.4 Scalability Traffic Scenario

Table 4.4: Scalability Arrival Rates

	Mean Arrival Rate (per second)	Std. Arrival Rate
User Sessions	69.71	8.44
Requests	103.47	10.44

In this scenario, the average rate of user session arrivals for all traffic sources was increased by a factor of 2, along with a proportional increase in the number of application servers to handle the load. The resulting arrival rates and their distributions are shown in Table 4.4 and Figure 4.4.

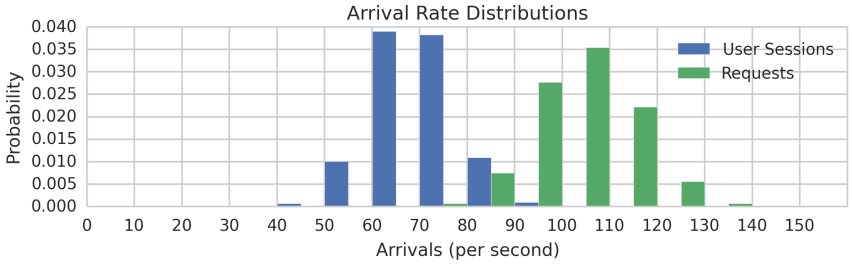


Figure 4.4: Scalability Scenario Arrival Rate

4.1.5 Increased Session Length Traffic Scenario

Table 4.5: Increased Session Length Arrival Rates

	Mean Arrival Rate (per second)	Std. Arrival Rate
User Sessions	35.71	6.21
Requests	52.92	7.47

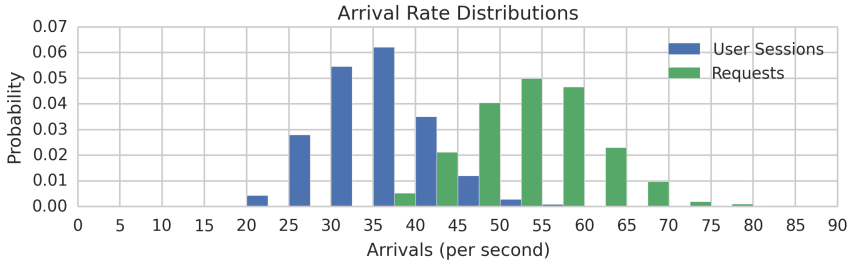


Figure 4.5: Increased Session Length Scenario Arrival Rate

In this scenario, the mean think time between requests in each simulated user session was increased to 60 seconds, drawing from an exponential distribution. The intention of this change was to measure the effect of longer sessions on the system, and determine whether this factor affected performance and stability in any way. The mean user session arrival rate was unchanged, and the steady state request rate was not affected. Due to the longer session durations, the load test took longer to reach a steady state of requests arriving. The resulting arrival rates and their distributions are shown in Table 4.5 and Figure 4.5.

4.2 Distribution of Response Time

The latency distribution of the API requests was calculated using the request duration field from the log files. In Figures 4.6 - 4.10, the distribution across all runs for each scenario is plotted, with a bin size of 2 milliseconds. This bin size was chosen to accurately represent the width of the peaks in the latency histogram while keeping the plot readable.

Across all the scenarios, a few key features were apparent from the latency distributions. The resulting distributions are multimodal, with two obvious peaks in the latency histogram. These likely represent requests that require different processing times based on the internal configuration of the system for each traffic source and are expected to occur in normal operation of the production system. Some traffic sources are configured with larger lists of advertisements to serve and more complex filtering requirements that must be processed on each request. This multimodal behaviour highlights the utility of testing the "Increased Traffic Source" scenario in addition to simply increasing the overall load. An increase in traffic from a source with a longer service time will likely have a larger impact on the system's performance. The distributions also have a long tail of higher latencies, with maximum values ranging as high as 400 - 1200 ms. Although these measurements make up a small fraction of the overall results, they may have an impact on a significant subset of users' perception of the system. If a subset of users see an advertisement loading slowly, this can result in negative feedback to publishers, which can in turn impact the company's reputation.

There are visible differences between the distributions within each scenario. The analysis of their statistical similarity in 4.3 shows that many runs are dissimilar despite the only variable changing across runs being the random seeds for generating the user sessions and user think times. Factors that could cause this include the use of virtualized public cloud servers for running the system under test and sensitivity of the system to minor variations in the workload. Using cloud servers means that the underlying hardware is not guaranteed to be free from contention for resources such as CPU, disk usage, and caches [9]. Although the system is recreated between runs by provisioning a new set of virtual machines in the Amazon Web Services public cloud, hardware with degraded performance or unexpected contention from a 3rd-party application for the same hardware could cause substantial variation in application performance. Although this effect could be avoided to some degree by using an modified deployment configuration, the goal with this analysis was to match the production environment as closely as possible.

These results suggest that the latency distribution for the system's steady-state be-

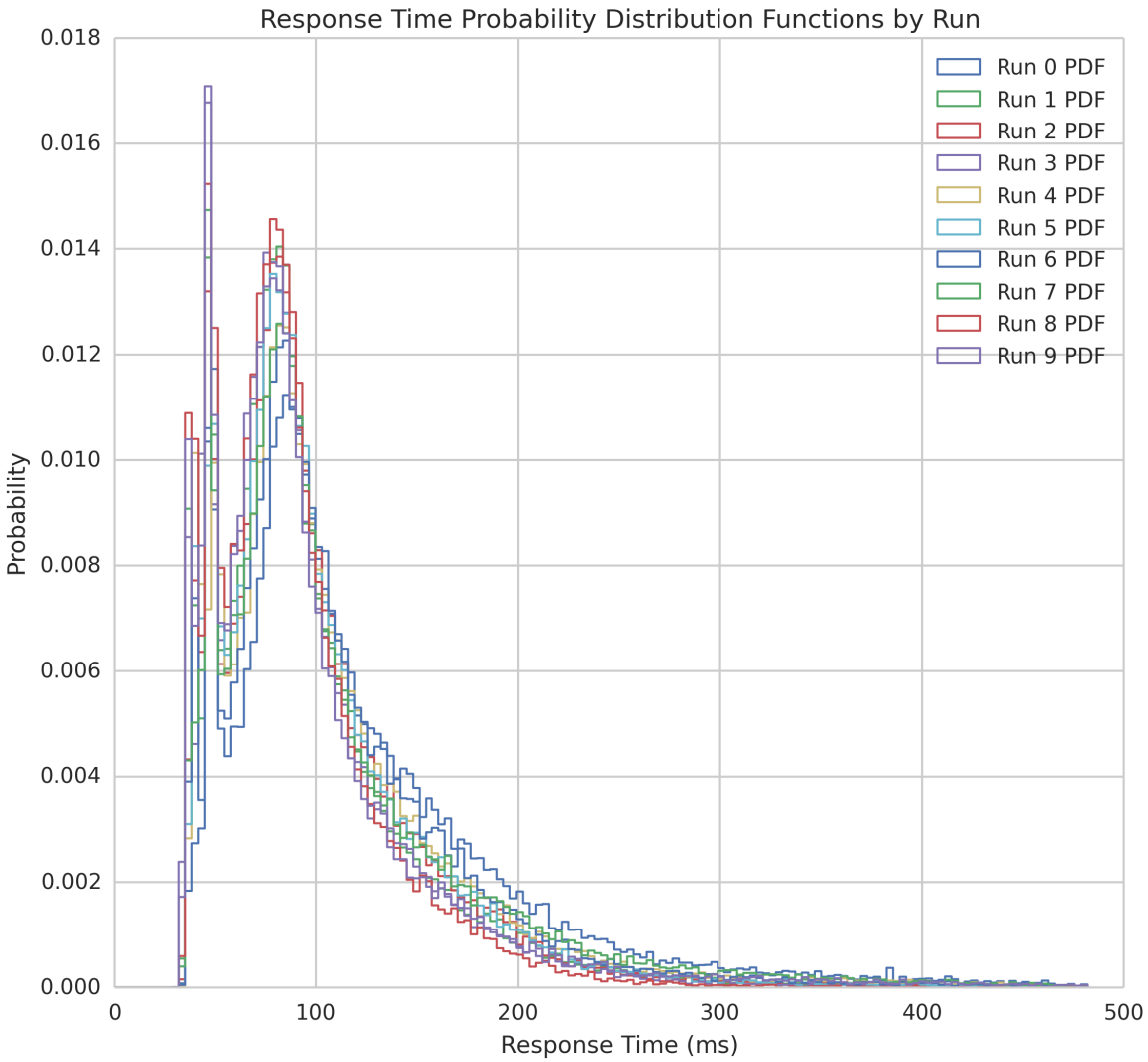


Figure 4.6: Baseline Traffic Scenario Response Time Distribution

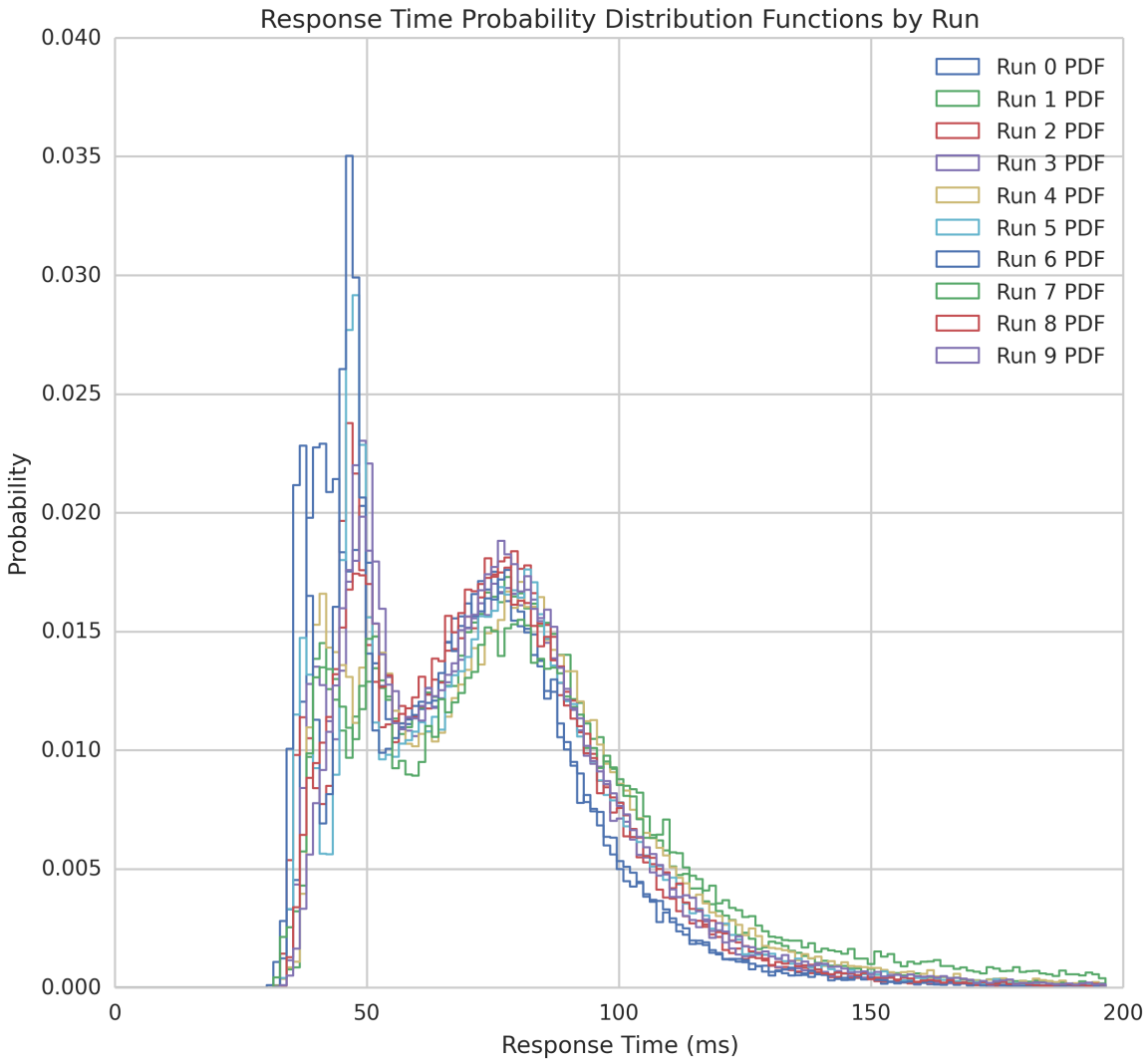


Figure 4.7: Increased Traffic Source Scenario Response Time Distribution

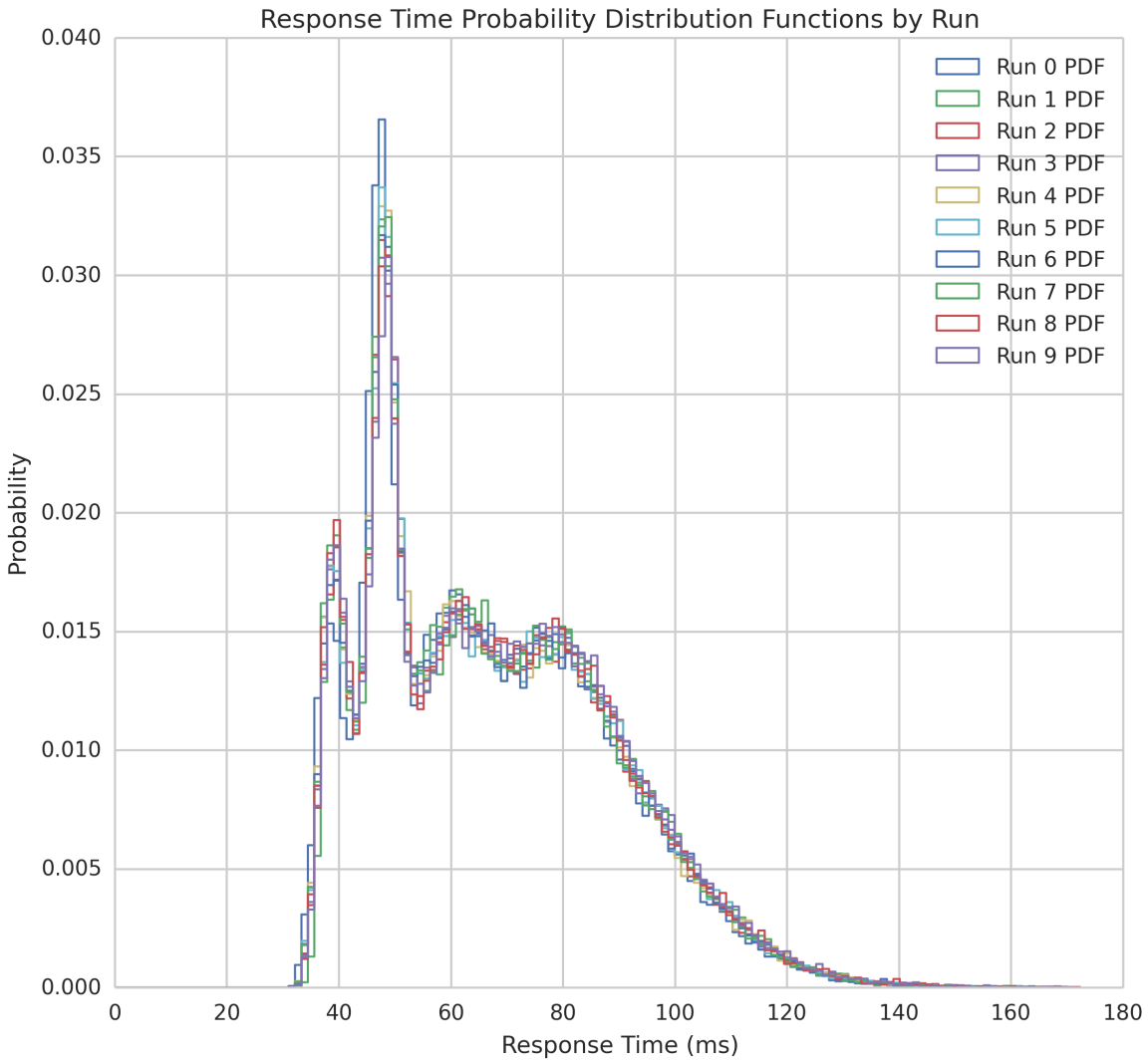


Figure 4.8: Increased Duration Scenario Response Time Distribution

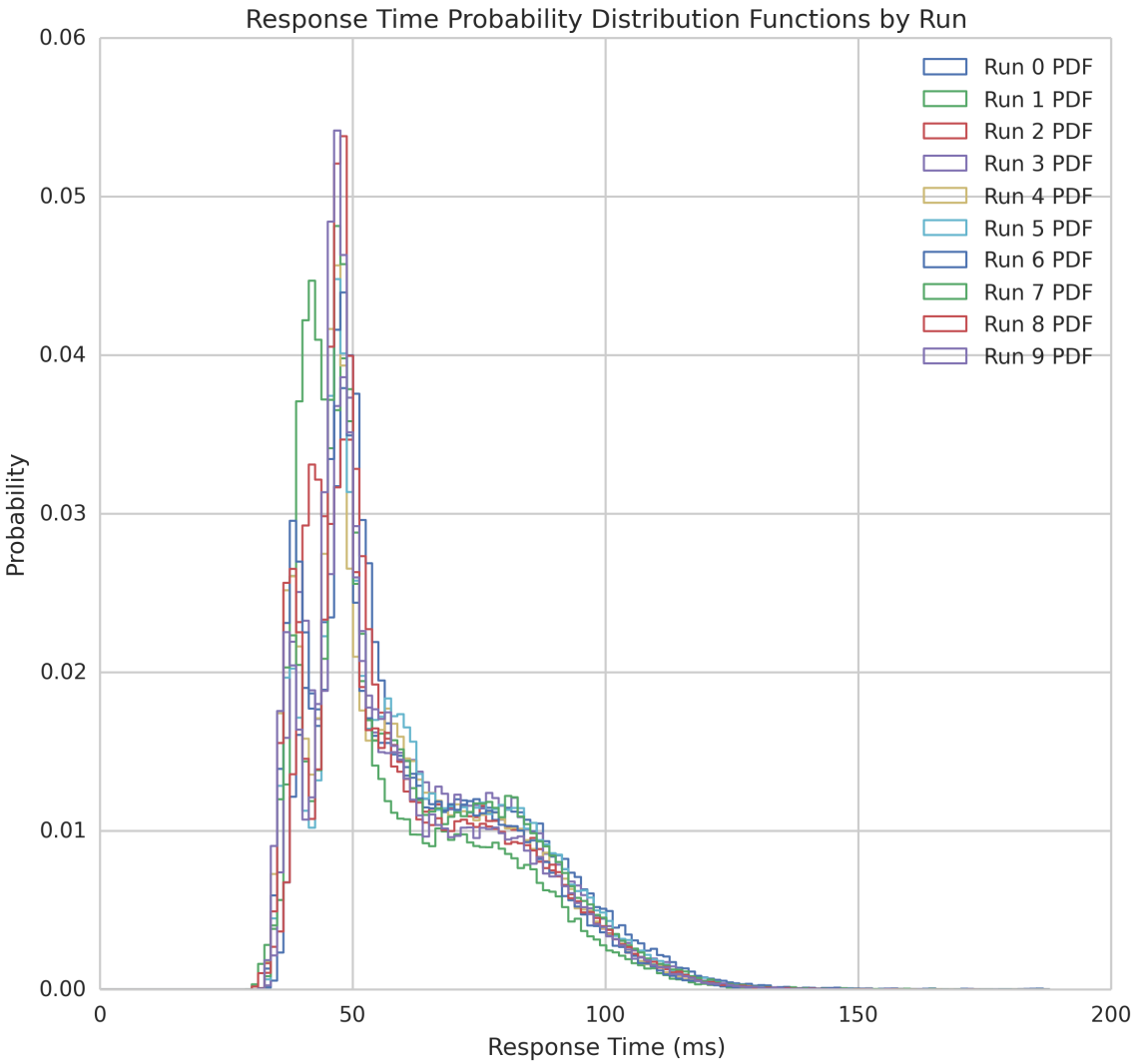


Figure 4.9: Scalability Scenario Response Time Distribution

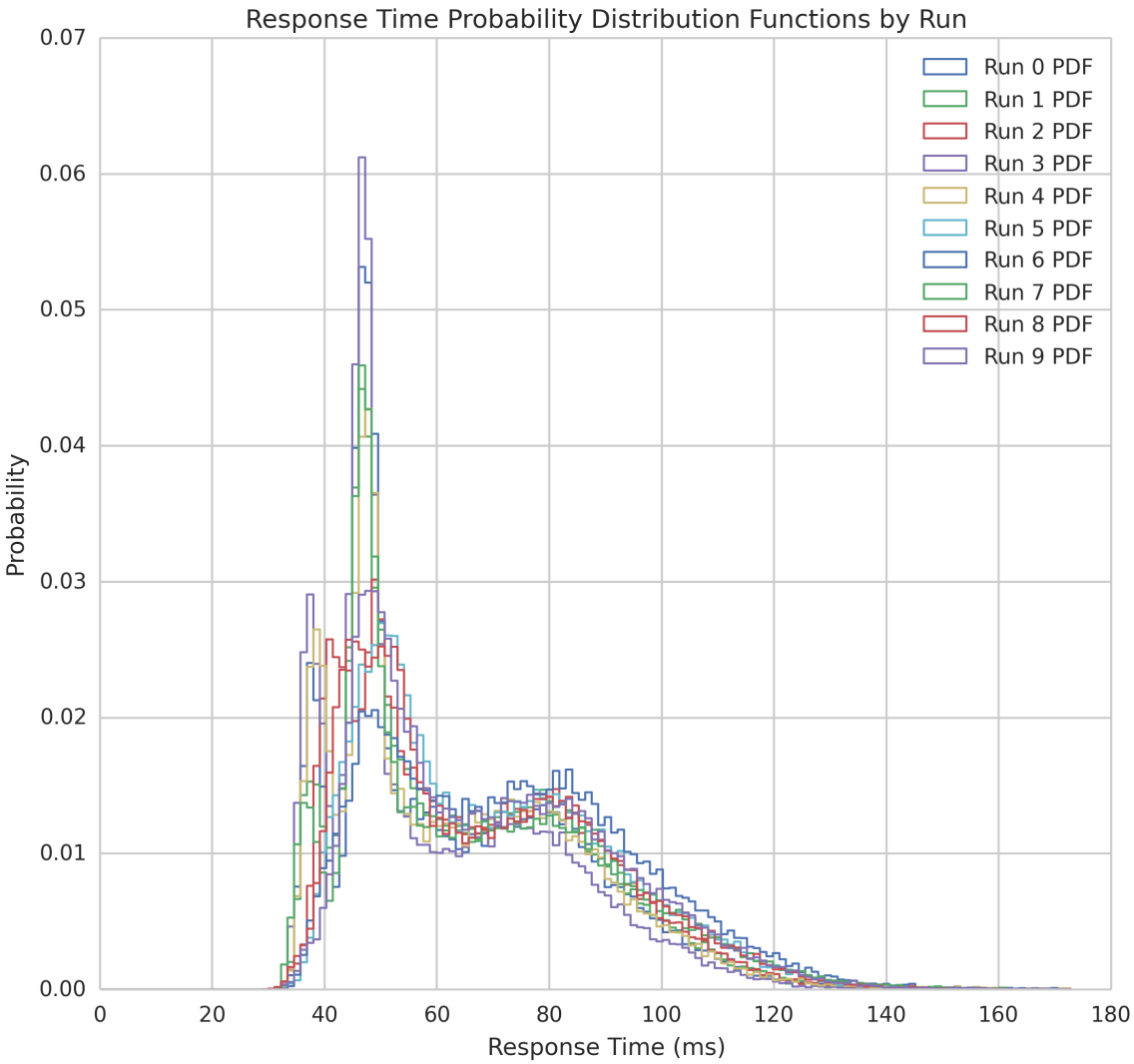


Figure 4.10: Increased Session Length Scenario Response Time Distribution

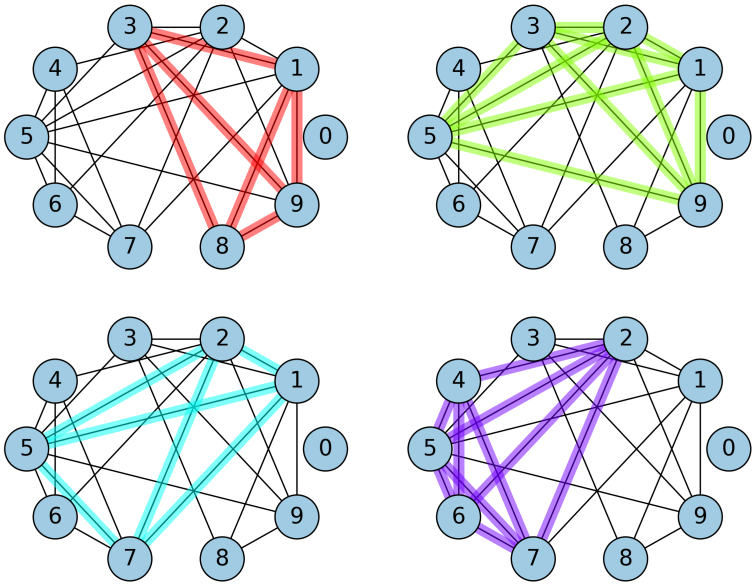
haviour for each run is dependent on the initial conditions. One possible conclusion from this result is that a series of independent runs is required to adequately assess the performance of the system in a given scenario. A single run may not reflect the real behaviour of the system in production. Moreover, average performance statistics are unlikely to be representative of the system's initial behaviour. However, the latency distributions are still quite similar in shape and location, and many of them could be treated as identical, depending on the goals of the testing. The results of the statistical testing process described in Section 4.3 could be adjusted by using a different test or confidence level to achieve this.

4.3 Testing Distributions for Similarity

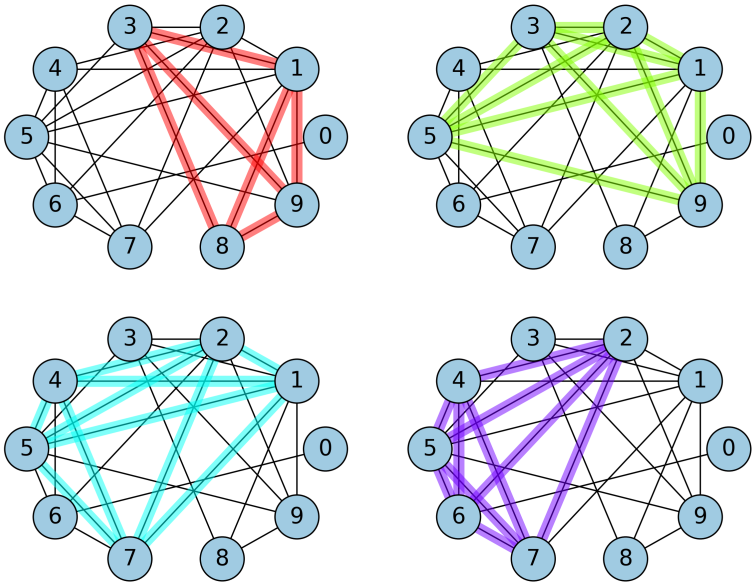
In this section, the cliques constructed using the process described in Section 3.4.3 are discussed. Sets of cliques identified by the KS-test and AD-test are shown together to compare the results of using each statistical test. The response time distributions across each clique are also shown to highlight the differences between cliques.

4.3.1 Baseline Scenario

In the baseline traffic scenario (Figure 4.11), not all of the runs were found to be statistically similar by the test, and several distinct cliques were identified. As there appears to be no obvious relationship between the order of the runs and their similarity, these distinct mode of operation were attributed to differences in the initial conditions for each run, such as the random seed used to generate the traffic, the underlying hardware the system was running on, and changes in the network conditions. The results of the KS-test and AD-test were also quite similar, identifying roughly the same cliques with only a few variations in members.



(a) KS-test



(b) AD-test

Figure 4.11: Baseline Scenario Cliques

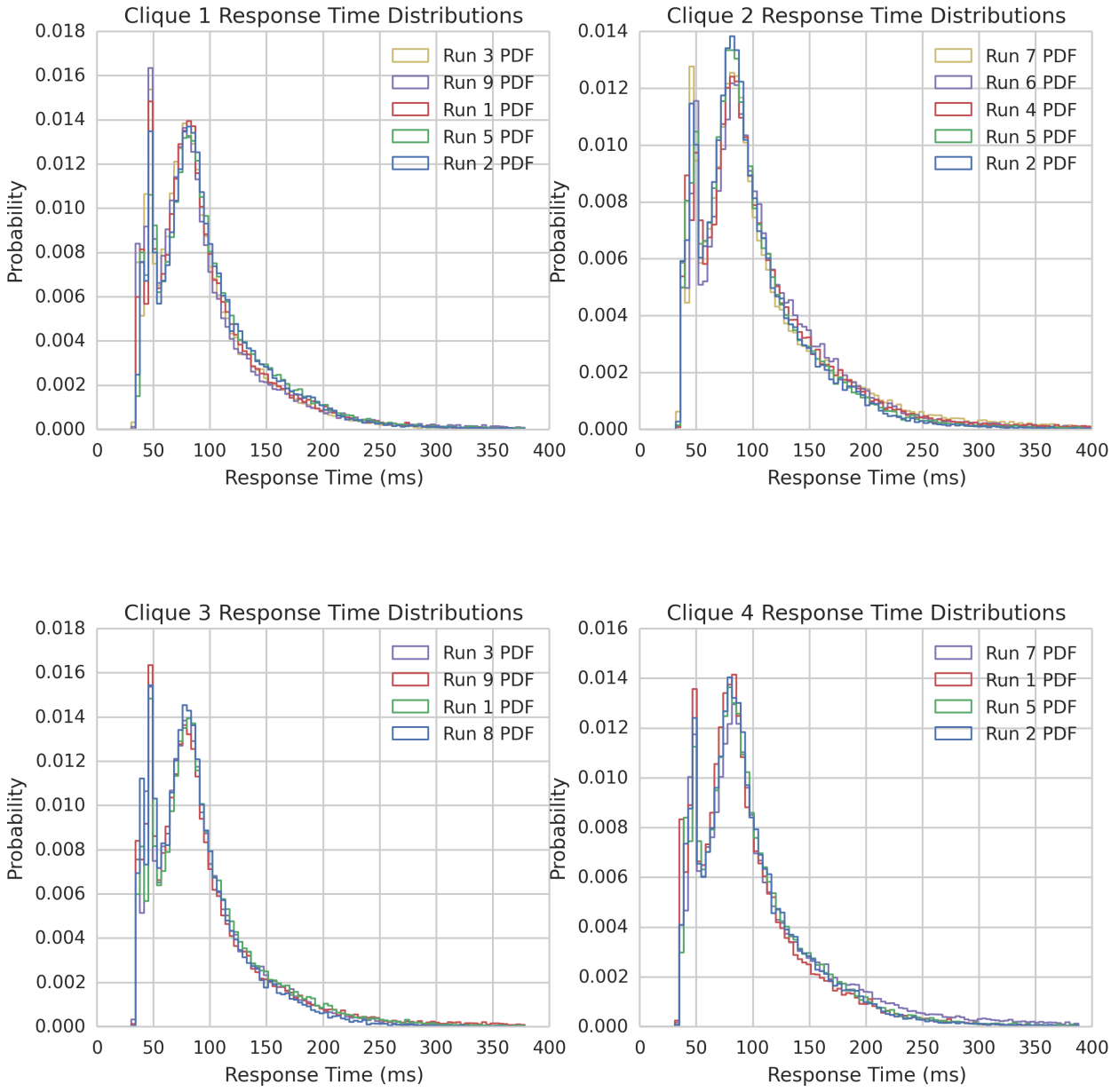


Figure 4.12: Baseline Scenario Clique Distributions (KS-test)

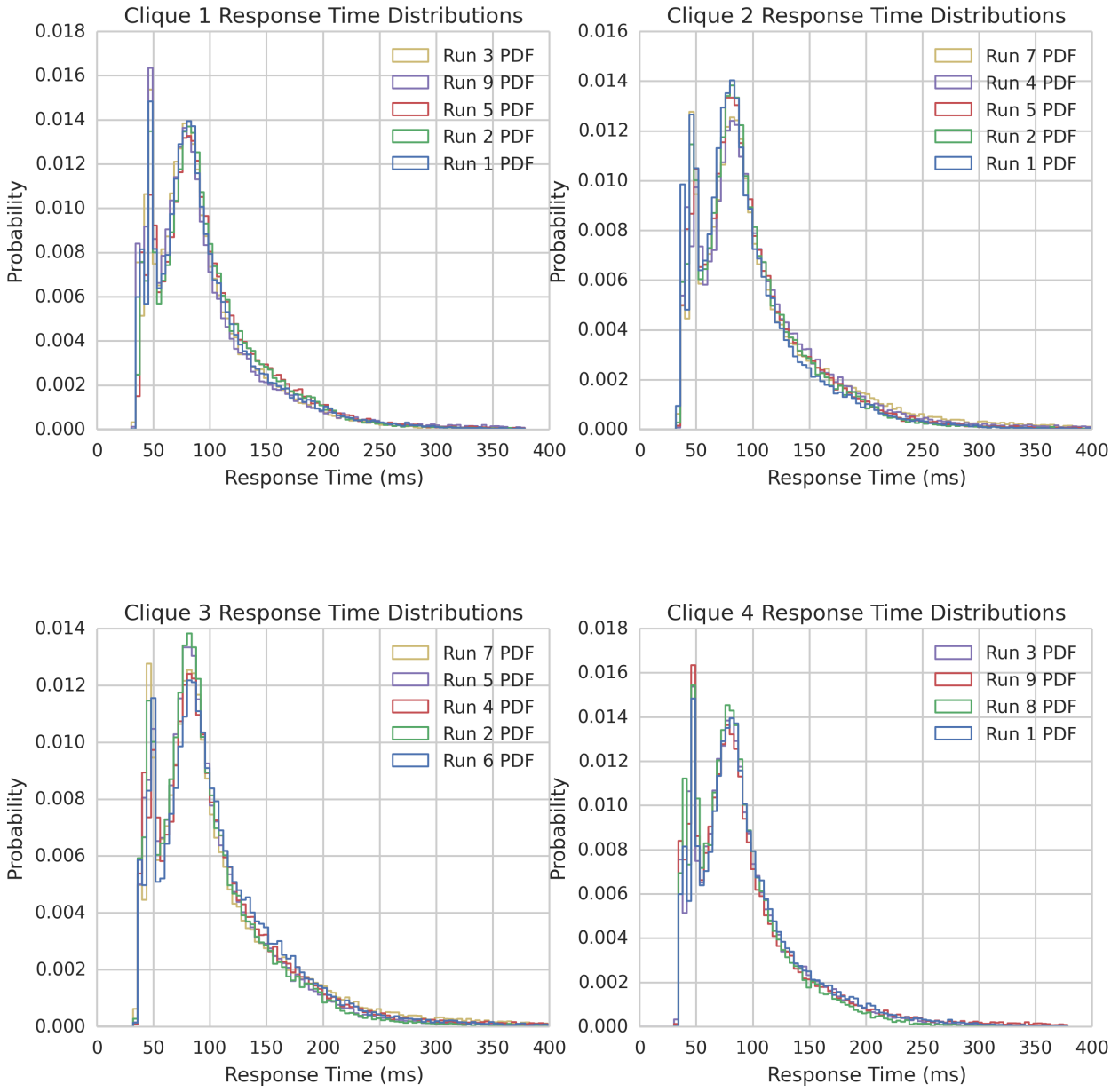
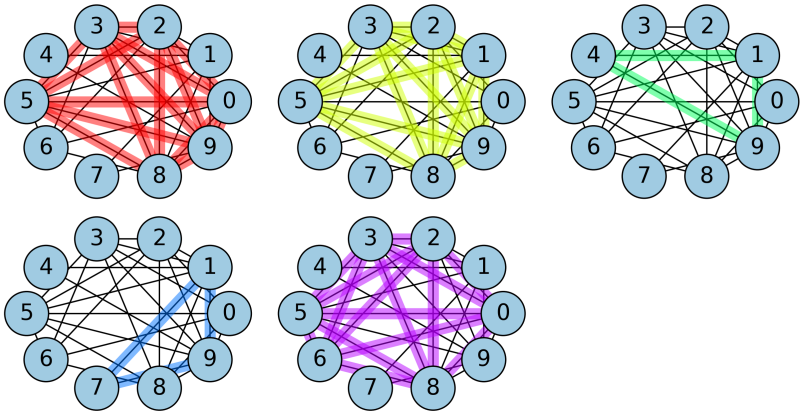


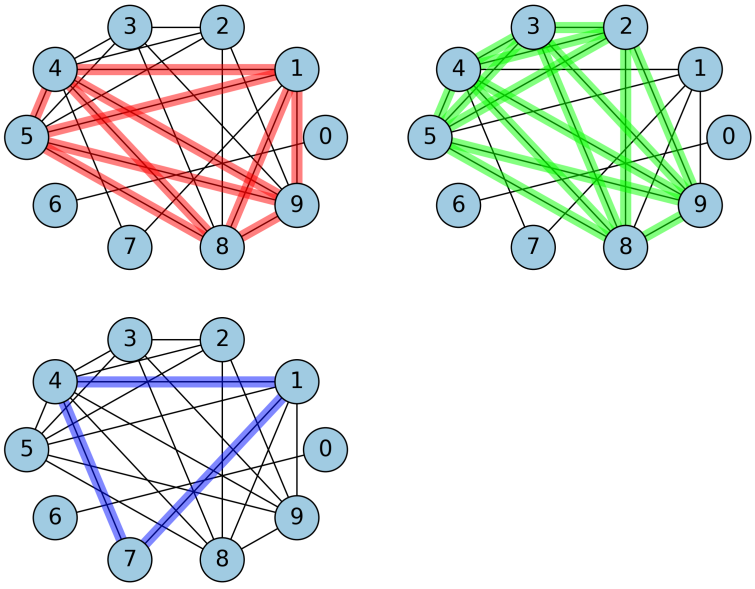
Figure 4.13: Baseline Scenario Clique Distributions (AD-test)

4.3.2 Increased Traffic Source Scenario

In the increased traffic source scenario (Figure 4.14), the number of distinct cliques increased in the KS-test, and fewer pairs of runs passed the AD-test. This suggests that the system's response becomes more sensitive to initial conditions and randomness as the load increases. The AD-test had fewer passing tests, but a smaller number of cliques. This suggests more of the distributions were statistically dissimilar, but more distinct modes of behaviour were identified only by the KS-test.



(a) KS-test



(b) AD-test

Figure 4.14: Increased Traffic Source Scenario Cliques

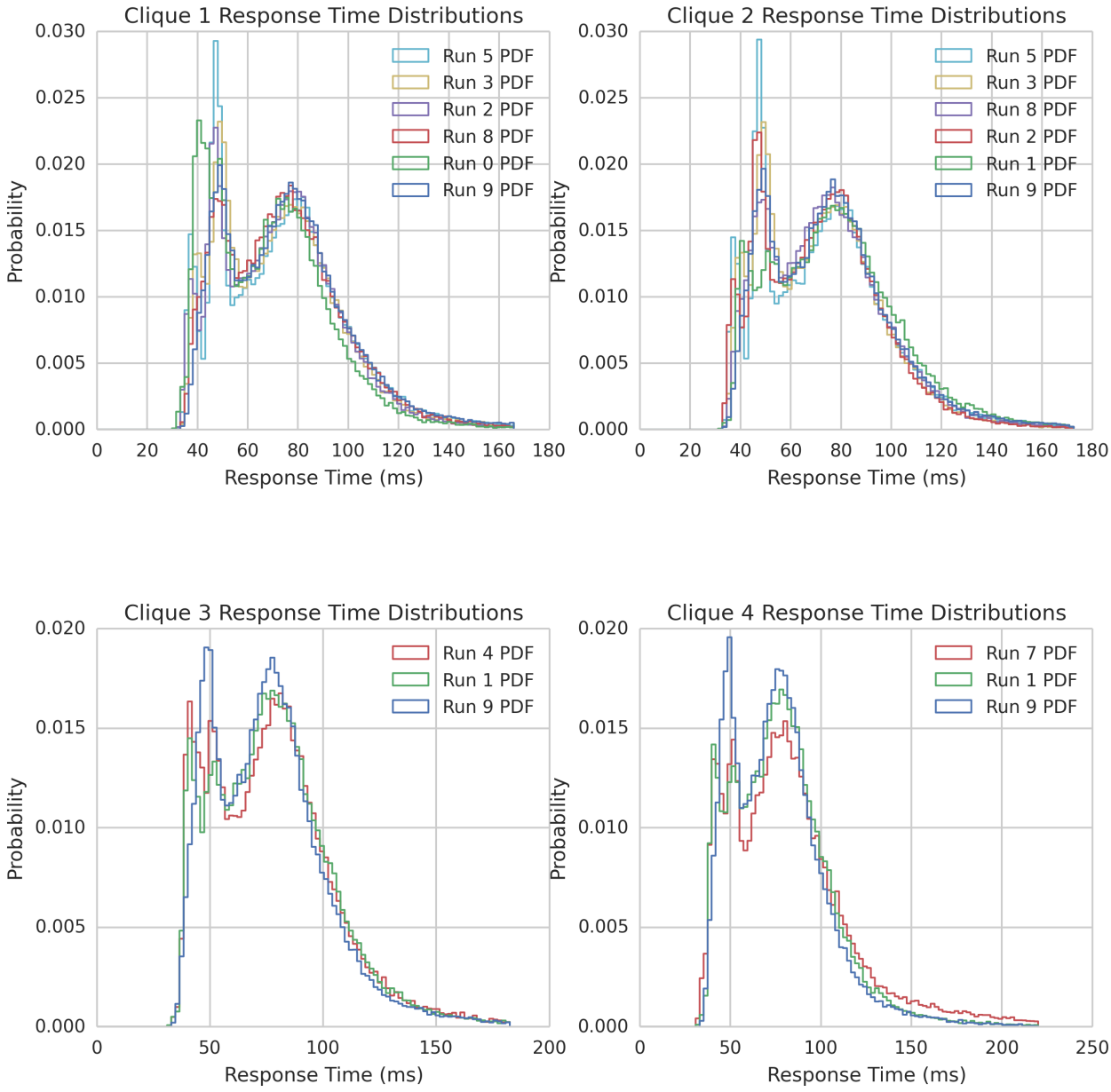


Figure 4.15: Increased Traffic Source Scenario Clique Distributions (KS-test)

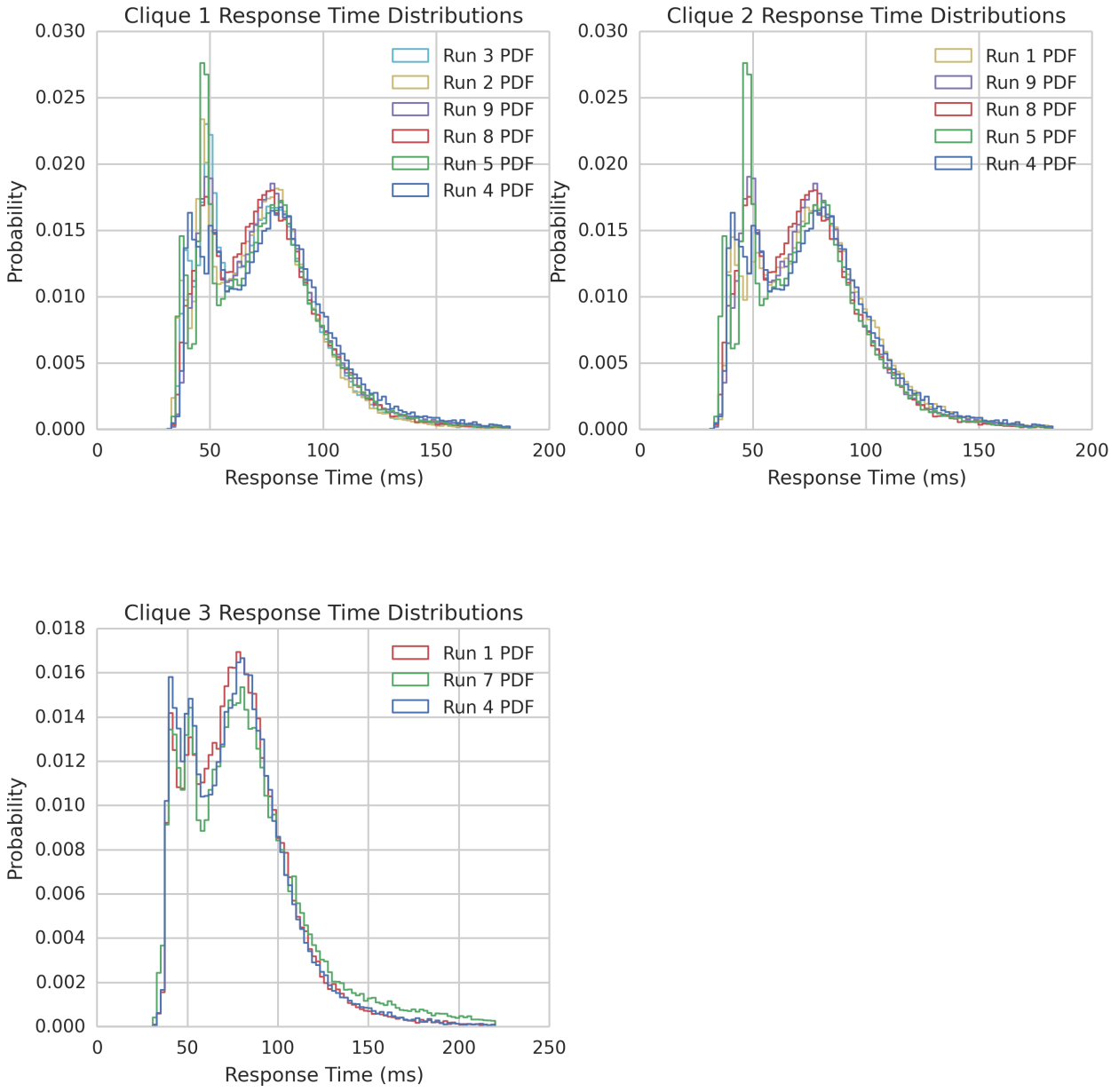
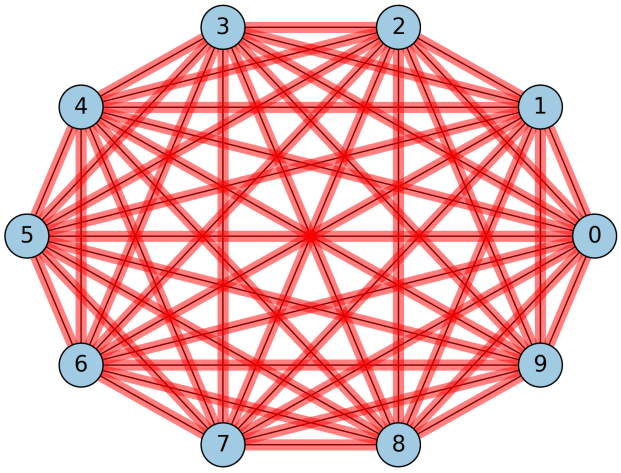


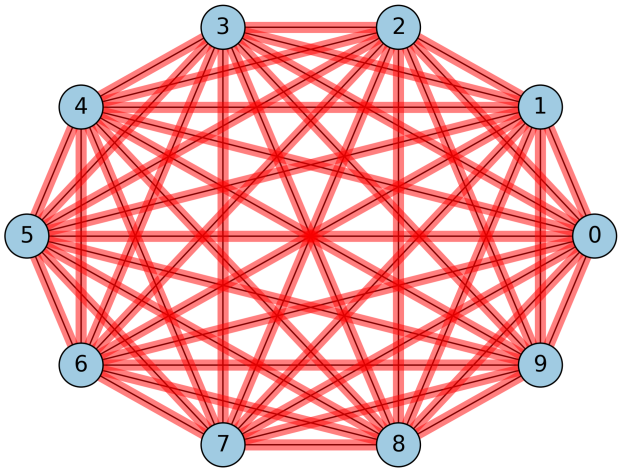
Figure 4.16: Increased Traffic Source Scenario Clique Distributions (AD-test)

4.3.3 Increased Duration Scenario

In the increased duration scenario (Figure 4.17), the latency distributions were visually quite similar across the consecutive time windows, and all of the statistical similarity tests passed. This supports the explanation that the changing behaviour across runs in the other scenarios is due to variations in the initial conditions, such as underlying hardware, network conditions, and random seeds. It also suggests that the system is relatively stable over the course of longer periods, since no statistically significant change was detected between the windows in time.



(a) KS-test

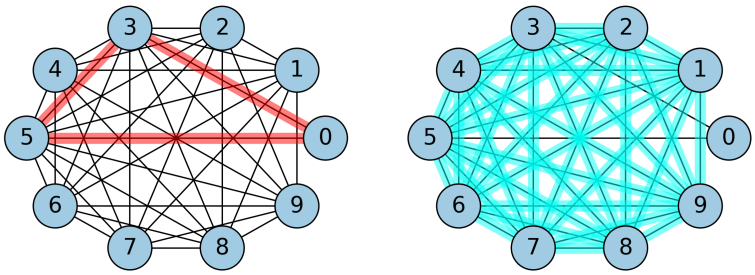


(b) AD-test

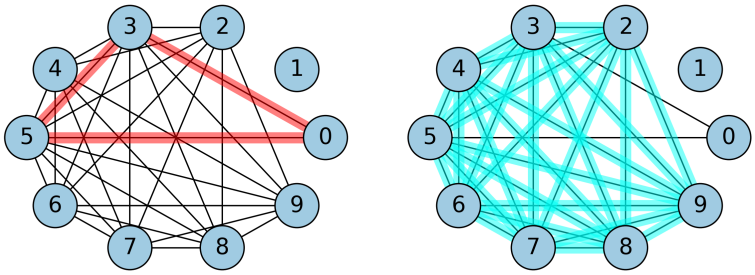
Figure 4.17: Increased Duration Scenario Cliques

4.3.4 Scalability Scenario

The scalability scenario (Figure 4.18) resulted in a lower median latency, indicating that the system was capable of scaling linearly up to this level of traffic. The increase in servers likely improved performance despite the proportionally increased traffic, because requests would have been spread evenly among servers by the load balancer, resulting in less requests being queued up for service by a back-end server. Most of the statistical similarity tests passed for both the KS-test and AD-test, resulting in two cliques, one much smaller than the other. This suggests that the smaller clique represents a different mode of behaviour, one with a wider distribution of response time. This result is particularly interesting because of the much smaller size of the second clique - if a single test had been run instead of an ensemble, it might not have been discovered at all.



(a) KS-test



(b) AD-test

Figure 4.18: Scalability Scenario Cliques

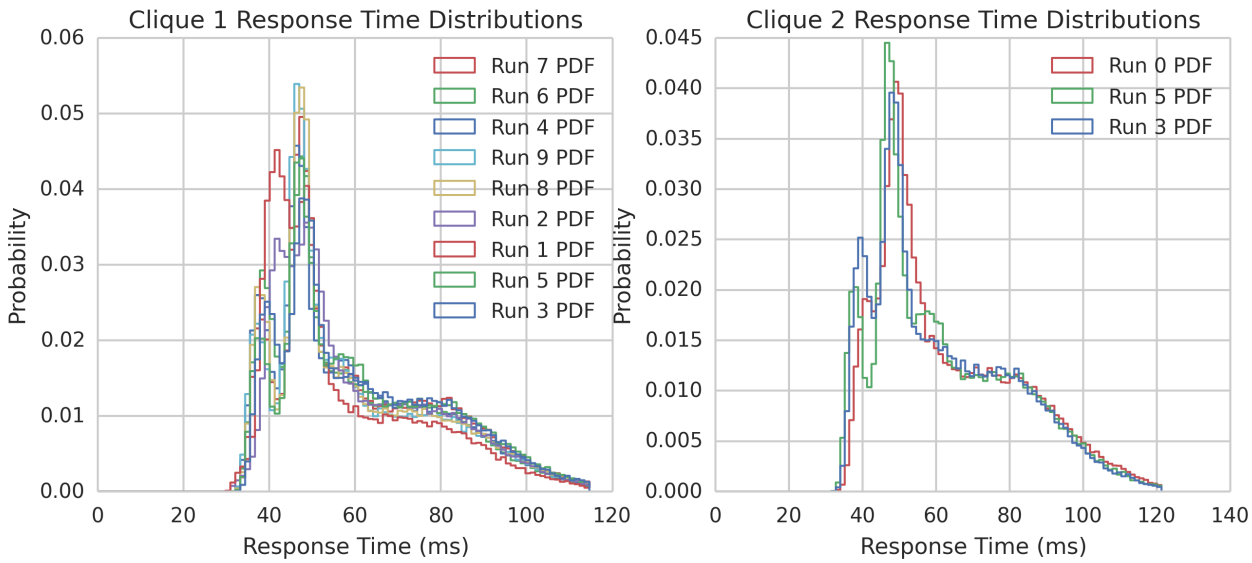


Figure 4.19: Scalability Scenario Clique Distributions (KS-test)

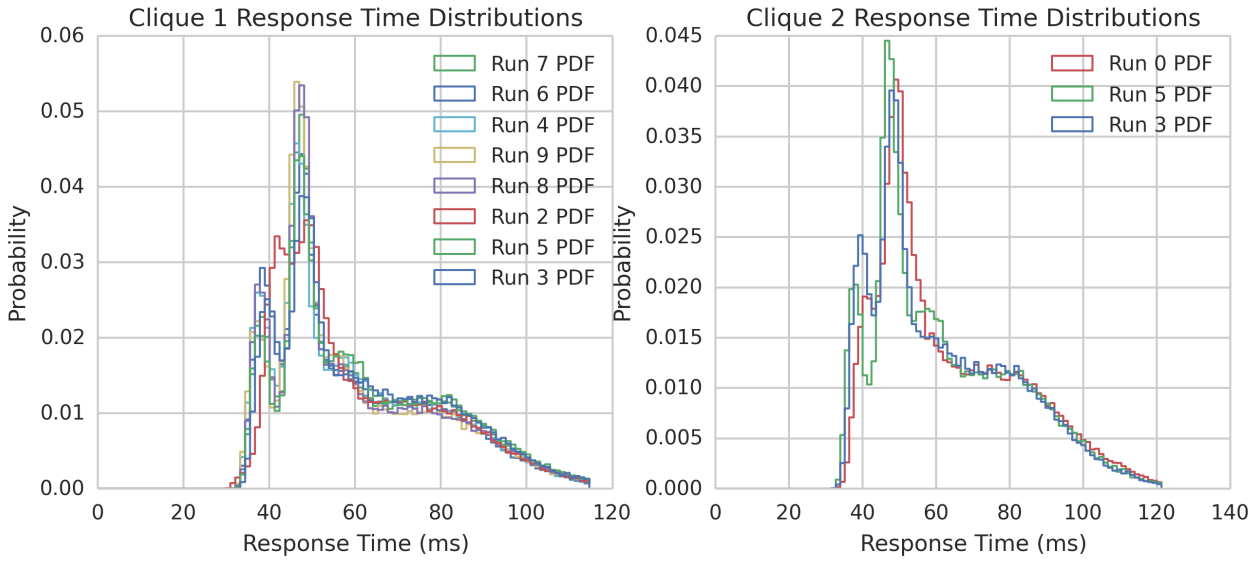
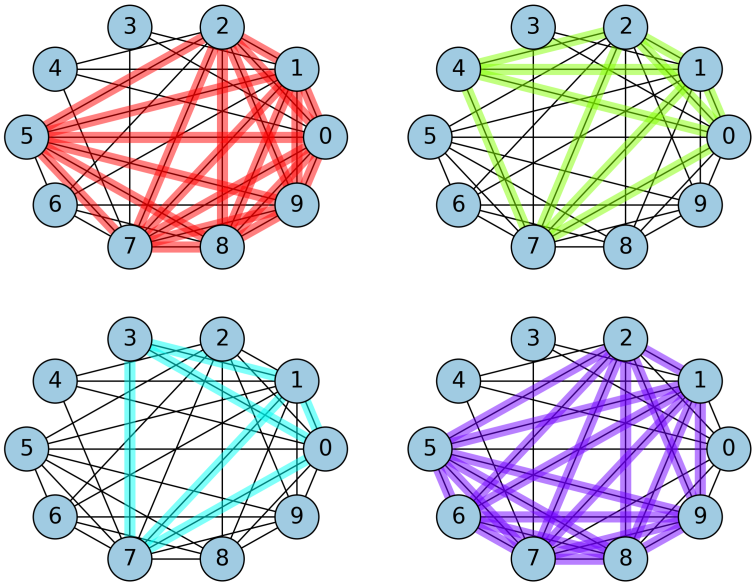


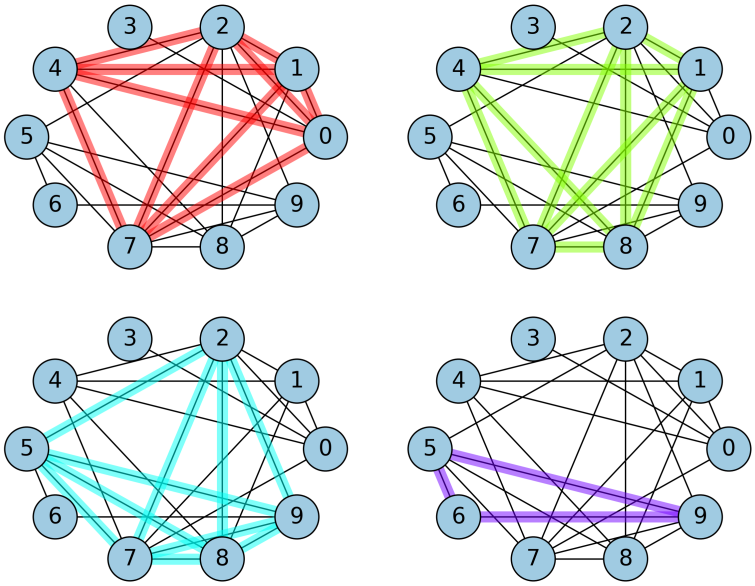
Figure 4.20: Scalability Scenario Clique Distributions (AD-test)

4.3.5 Increased Session Length Scenario

In the increased session length test (Figure 4.21), many of the tests passed but very different cliques were identified by each statistical similarity test. It appears from the results that introducing a longer delay between requests in the user sessions makes the resulting response time distributions more variable. Depending on the initial conditions of the test and random variation in the session arrivals, this change could cause requests to arrive in bursts of traffic that would alter the performance in unpredictable ways. The AD-test, which passed on fewer pairs of runs overall, is likely to pick up on different variations in the distributions than the KS-test and resulted in a different set of cliques.



(a) KS-test



(b) AD-test

Figure 4.21: Increased Session Length Scenario Cliques

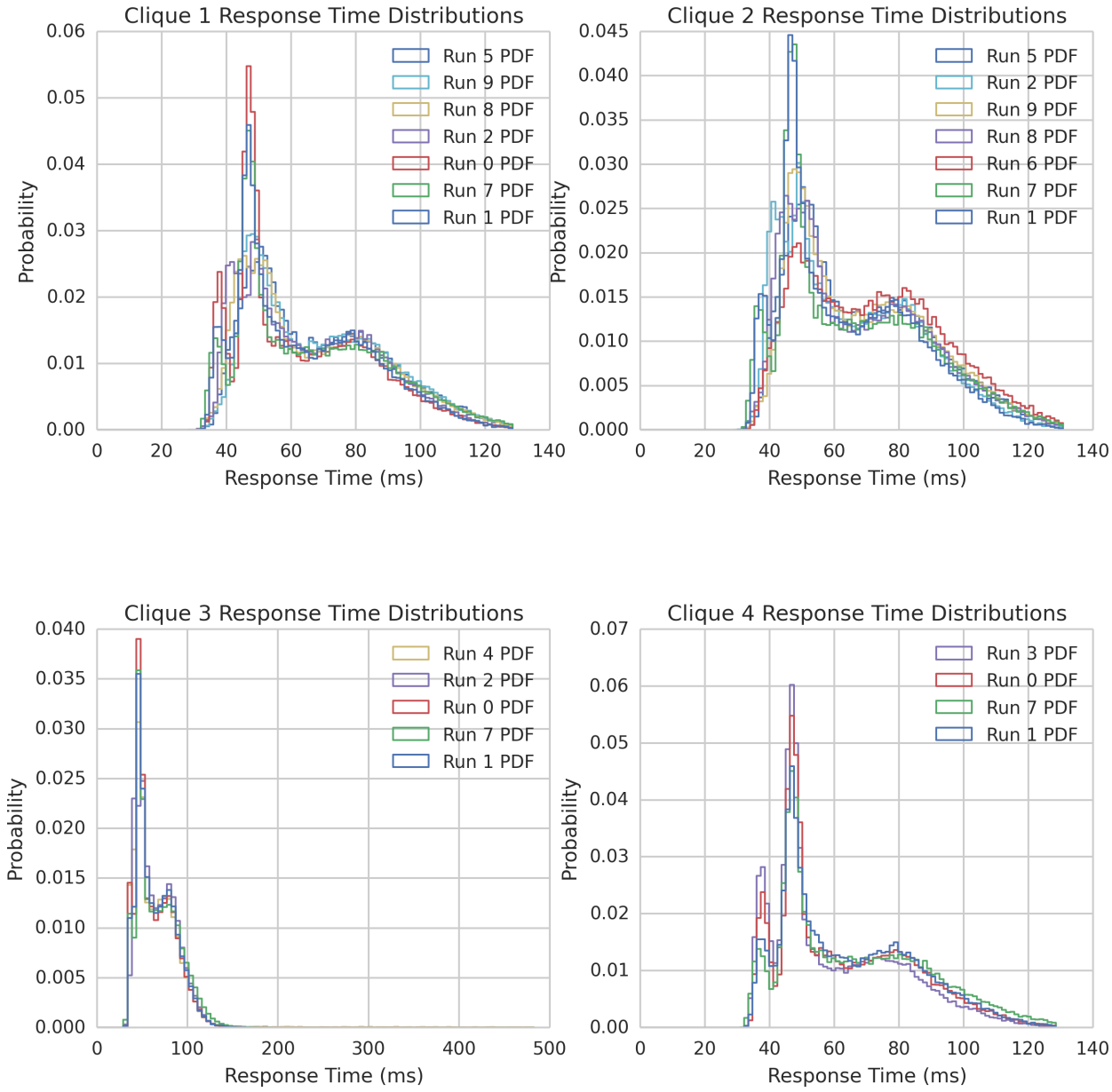


Figure 4.22: Increased Session Length Scenario Clique Distributions (KS-test)

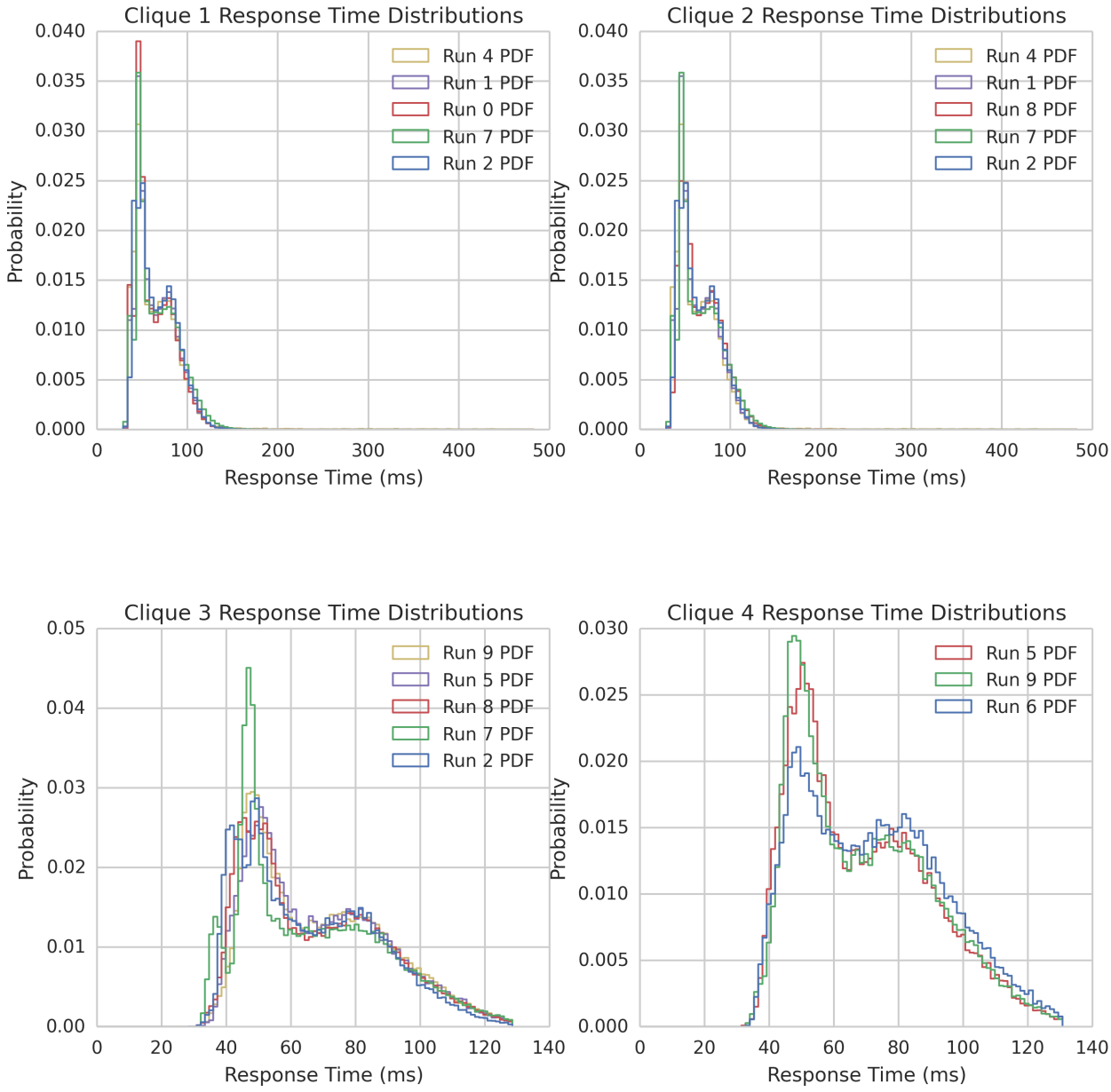


Figure 4.23: Increased Session Length Scenario Clique Distributions (AD-test)

Chapter 5

Conclusions

This chapter summarizes the objectives and results of the thesis. The limitations of the methodology and current level of study are discussed. It concludes by discussing several areas of future work and applications.

5.1 Evaluation of the Results

The objective of this thesis was to develop a model for realistic load testing of a real-world distributed system and analyze the results of several different workload scenarios. This thesis focused on characterizing the workload of the real-world Pretio distributed system using logs captured from the production deployment, modelling a workload from those logs, and analyzing the impact on a test deployment of the system of a series of scenarios providing different parameters to the model.

The data from the production logs indicated that using a Poisson model for the arrivals of individual user sessions, and generating the subsequent requests with a Markov model would be sufficient to reproduce the peak steady-state workload accurately. Further analysis on the properties of self-similarity and long-range dependence in the session arrivals determined that there was no evidence of these effects.

The data from analyzing the impact of the generated workload against the test deployment suggested that the results were highly variable and dependent on initial conditions such as the underlying cloud servers, the network conditions, and random variations in the arrivals of user sessions and timing of requests. The results indicate that a series of repeated tests is necessary to obtain a clear picture of the results for a given test sce-

nario, and that the results of a single test may not be indicative of the real response time distribution that arises in a real-world scenario.

The individual scenarios yielded results that were consistent with this overall trend, but also confirmed several hypotheses about the system under test. The result of increasing the traffic from a single publisher (Section 3.3.4) resulted in more response time distributions that appeared to be statistically dissimilar, suggesting that this change to the workload makes the system's response more sensitive to initial conditions. In contrast, the results from the increased duration scenario (Section 3.3.5) suggested that the system's response time distribution was stable over the longer period, and did not exhibit the same variation in behaviour between time-based windows of the system's response time. The results from the scalability scenario (Section 3.3.6) indicated that the system was capable of horizontally scaling up to the increased workload, and that the resulting response time distributions were actually less sensitive to initial conditions. In the increased session length scenario (Section 3.3.7), the results of the two statistical similarity tests were more inconsistent. This suggests that the change in simulated user behaviour did have an impact on the response time which only one of the two statistical similarity tests (KS-test and AD-test) is effective at highlighting. However, the results of this scenario did not appear to otherwise suggest a noticeable difference in the number of statistically similar response time distributions or how many distinct cliques were formed.

5.2 Limitations

The limitations of this work fit into two broad categories: limitations in the number, duration, and scale of load tests, and simplifications in the workload model that make it less representative of the real-world workload.

One of the primary conclusions of this thesis is that multiple test runs are necessary to adequately assess the performance of the system for a given workload. It is likely that many more tests are also needed here to understand the system's performance fully, and that their duration should be increased beyond 30 minutes. This would provide more data and a more complete understanding of the differences in response time distributions measured across multiple runs of the same test. Given the results of the scalability scenario (Section 4.1.4), it appears that the response time distributions may be less statistically different when a larger workload is used. The limits of this effect are outside the scope of this work, and the upper limit to the system's ability to scale horizontally by adding more web servers was not determined. Although the long running test suggested that the

system was stable across consecutive time periods and across the entire 5 hour duration of the test, it is unknown whether this holds true for longer time periods or at higher traffic volumes.

Many aspects of the workload model were simplified in order to keep the complexity of the load testing process low, and to avoid errors in the model. The parameters of each request are simulated based on real example requests and some simple assumptions around the number of distinct request parameter values and their distributions. However, these parameters could have an impact on the time taken to service a request and the long-running operation of the system, and should be modeled in a more realistic way. Although the real-world workload is known to exhibit a daily cycle, only the peak steady-state load behaviour was modelled. Modelling this daily cycle would allow for a more realistic long-term test with a duration in the order of hours or days. The increased session length scenario (Section 3.3.7) was designed to simulate the effects of user behaviours that could cause self-similarity in the request arrival rates. However, this behaviour was not modelled based on the logs - instead, an exponential think time distribution was assumed for each scenario based on average think times of users in the production system. This simplified model was used because of the complexity involved in fitting a distribution to the think times within individual sessions, as opposed to between session arrivals.

5.3 Future Work

Larger scale versions of these load tests should be repeated with longer time frames and more test runs. Based on the results, it seems likely that tens or even hundreds of tests would be required to get a more accurate picture of the different modes of behaviour the system displays within each scenario. A much longer version of the long running scenario (in the order of days) would provide a more conclusive measurement of the system's stability over time.

To address one of the limitations in the workload, a more accurate method of simulating the request content would need to be devised. The method described in Krishnamurthy et al. [14] may be a viable starting point. This technique involves sampling from a set of sequences of inter-related requests that are derived from the production system's log files. If these sets of request sequences were large enough, sampling from them should result in a representative distribution of request parameters. To address the issue of the daily cycle in the workload, the workload model could use a Nonhomogeneous Poisson Process [10]

for request arrivals, where the arrival rate parameter λ changes according to a function of the time of day.

Lastly, the implementation of the system under test was frozen at a particular point in time to allow for a consistent testing process and to avoid incompatibility errors between the workload and the features supported by the system. A possible next step for making this research useful to the company would be to update the system to the latest version, and set up periodic automated load tests with reporting of the results to the development team. This would aid in exposing performance regressions and allow testing of hypothetical scenarios instead of relying solely on benchmarks and testing in production.

5.4 Implications

The cliques discovered in the response time distributions suggest that there are multiple modes of behaviour for the underlying system. If this is the case, 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. It also indicates that for distributed systems, particularly those deployed in cloud environments, many iterations of a load test are required in order to characterize the system's response to a given workload. Properly configuring, deploying, and analysing the results of these realistic workload tests remains a complex task for developers of distributed systems. Software tools that automate these tasks and aid in the analysis of the results would help to make realistic distributed system testing more accessible to software developers.

Appendix A

Tsung Configuration Template

This is the template for the configuration file that is used to control the Tsung load test client. It is encoded in XML, but contains template placeholders delimited with {{ }} and {% %}.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE tsung SYSTEM "/usr/local/share/tsung/tsung-1.0.dtd">
3  <!-- dumptraffic="protocol" forces tsung to log every request as CSV -->
4  <tsung loglevel="warning" dumptraffic="protocol" version="1.0">
5
6    <!-- Client side setup -->
7    <clients>
8      <client host="localhost" use_controller_vm="true" maxusers="10000" />
9    </clients>
10
11   <!-- Server side setup -->
12   <servers>
13     <server host="<LOAD BALANCER HOSTNAME>" port="80" type="tcp"></server>
14   </servers>
15
16   <!-- Maximum duration of entire session -->
17   <load duration="{{ runtime_minutes + 10 }}" unit="minute">
18
19     <!-- Warm up the web servers and caches first in a single-user phase -->
20     <arrivalphase phase="1" duration="1" unit="minute">
21       <users maxnumber="1" interarrival="1" unit="second"></users>
22
```

```
23      <session_setup name="warmup" probability="100"/>
24
25      <!-- Turn off real sessions during warmup -->
26      {%- for api_key in session_probabilities %}
27      <session_setup name="{{ api_key }}" probability="0"/>
28      {%- endfor %}
29
30    </arrivalphase>
31
32    <!-- initial ramp up -->
33    <arrivalphase phase="2" duration="1" unit="minute">
34        <users arrivalrate="{{ global_arrival_rate / 2.0 }}" unit="second" />
35    </arrivalphase>
36
37    <!-- A group of users arriving over a fixed duration -->
38    <arrivalphase phase="3" duration="{{ runtime_minutes }}" unit="minute">
39
40        <!-- Define global arrival rate of user sessions -->
41        <users arrivalrate="{{ global_arrival_rate }}" unit="second" />
42
43    </arrivalphase>
44
45  </load>
46
47  <options>
48
49    <!-- Set a random seed, so that individual tests are repeatable -->
50    <option name="seed" value="{% raw %}{{ seed }}{% endraw %}"/>
51
52    <!-- Disable retries -->
53    <option name="max_retries" value="0" />
54
55    <!-- Timeout for establishing a TCP connection -->
56    <option name="connect_timeout" value="1000" />
57
58  </options>
59
```

```
<sessions>

  <!-- Send a single request for each publisher -->
  <session name="warmup" probability="0" type="ts_http">

    {%- for api_key in session_probabilities -%}
    <request>
      <http url="{{ '/publishers/' + api_key + '/api/?user_id=WARMUP&
          country_code=US' }}" method="GET" version="1.1"></http>
    </request>
    {% endfor %}

  </session>

  {% for api_key, probability in session_probabilities.iteritems() -%}
  <session name="{{ api_key }}" probability="{{ probability }}" type="
      ts_http">

    <!-- Assume each publisher has 100 unique values for sub_id -->
    <setdynvars sourcetype="random_number" start="1" end="100">
      <var name="sub_id" />
    </setdynvars>

    <!-- Generate unique user_id for every session -->
    <setdynvars sourcetype="random_string" length="32">
      <var name="user_id" />
    </setdynvars>

    <!-- Generate random numbers to match against each event probability
        -->
    <setdynvars sourcetype="random_number" start="0" end="99">
      <var name="imp_probability" />
    </setdynvars>
    <setdynvars sourcetype="random_number" start="0" end="99">
      <var name="red_probability" />
    </setdynvars>
    <setdynvars sourcetype="random_number" start="0" end="99">
```

```
94          <var name="cli_probability" />
95        </setdynvars>
96        <setdynvars sourcetype="random_number" start="0" end="99">
97          <var name="con_probability" />
98        </setdynvars>
99
100       <!-- TYPE 1 REQUEST -->
101       <request subst="true">
102
103         <!-- Match 200 response from server or abort user session -->
104         <match do="abort" when="nomatch">HTTP/1.1 200 OK</match>
105
106         <!-- Treat HTML responses as errors -->
107         <match do="abort" when="match">"html":</match>
108
109         <!-- Extract variables from JSON response -->
110         <dyn_variable name="funnel_id" jsonpath="pretio_offer_id"/>
111         <dyn_variable name="creative_url" jsonpath="url"/>
112
113         <!-- Make HTTP request -->
114         <http url="{{ '/publishers/' + api_key + '/api/?user_id=%%_user_id%%&
                sub_id=%%_sub_id%%&country_code=US&sub_id2=US' }}" method="GET"
                version="1.1"></http>
115
116       </request>
117
118       {% if events %}
119       <if var="imp_probability" lt="{{ event_probabilities[api_key].
            imp_percent }}">
120
121         {% if thinktime_override %}
122         <thinktime value="{{ thinktime_override }}" random="true"></thinktime
                >
123         {% else %}
124         <thinktime value="1" random="true"></thinktime>
125         {% endif %}
126
```

```
127         <!-- TYPE 2 REQUEST: IMPRESSION -->
128         <request subst="true">
129
130           <!-- Match 200 response from server or abort user session -->
131           <match do="abort" when="nomatch">HTTP/1.1 200 OK</match>
132
133           <!-- Make HTTP Request -->
134           <http url="/offers/%%_funnel_id%%/impression.gif" method="GET"
                  version="1.1"></http>
135
136         </request>
137
138         <if var="red_probability" lt="{{ event_probabilities[api_key].
                red_percent }}">
139
140           {% if thinktime_override %}
141           <thinktime value="{{ thinktime_override }}" random="true">></
                thinktime>
142           {% else %}
143           <thinktime value="10" random="true"></thinktime>
144           {% endif %}
145
146           <!-- TYPE 2 REQUEST: REDEEM -->
147           <request subst="true">
148
149             <!-- Match 302 response from server or abort user session -->
150             <match do="abort" when="nomatch">HTTP/1.1 302 FOUND</match>
151
152             <!-- Make HTTP Request -->
153             <http url="%%_creative_url%%" method="POST" contents="%%
                    _form_contents%%" version="1.1"></http>
154
155           </request>
156
157           <if var="cli_probability" lt="{{ event_probabilities[api_key].
                  cli_percent }}">
158
```

```
159             <!-- TYPE 2 REQUEST: CLICK -->
160             <request subst="true">
161
162               <!-- Match 302 response from server or abort user session -->
163               <match do="abort" when="nomatch">HTTP/1.1 302 FOUND</match>
164
165               <!-- Make HTTP request -->
166               <http url="/offers/%%_funnel_id%%/claim/" method="GET" version
                      ="1.1"></http>
167
168             </request>
169
170             <if var="con_probability" lt="{{ event_probabilities[api_key].
                    con_percent }}">
171
172               {% if thinktime_override %}
173               <thinktime value="{{ thinktime_override }}" random="true"></
                      thinktime>
174               {% else %}
175               <thinktime value="30" random="true"></thinktime>
176               {% endif %}
177
178               <!-- TYPE 2 REQUEST: CONVERT -->
179               <request subst="true">
180
181                 <!-- Match 200 response from server or abort user session -->
182                 <match do="abort" when="nomatch">HTTP/1.1 200 OK</match>
183
184                 <!-- Make HTTP request -->
185                 <http url="/advertisers/testing/convert.gif?pretio_offer_id=%%
                        _funnel_id%%" method="GET" version="1.1"></http>
186
187               </request>
188
189             </if>
190           </if>
191         </if>
```

```
192       </if>
193       {% endif %}
194
195     </session>
196
197     {% endfor %}
198   </sessions>
199 </tsung>
```

Appendix B

Load Test Parameters

Table B.1: Scenario Parameters

Scenario	Arrival Rate	Runtime (s)	Mean Think Time (s)
Baseline	35.771894	1800	10
Increased Traffic Source	48.114188	1800	10
Increased Session Length	35.771894	1800	60
Increased Duration	35.771894	18000	10
Scalability	71.543788	1800	10

Table B.2: Session and Event Probabilities

The following table contains the probabilities of each type of request occurring in a session. For each user session that is generated, the value in the Session column is the probability that it will be assigned that Publisher as its source. The values in the subsequent columns (Type2A - Type2D) for that row are then the transition probabilities used in Figure 3.5 for that simulated user's session.

Publisher	Session	Type2A	Type2B	Type2C	Type2D
03b3f3547b574aa181de8bb962040538	0.000938	0.122449	0.000000	0.000000	0.000000
f4ca6412daa9406eb86ab9b0a947e259	0.000187	0.000000	0.000000	0.000000	0.000000
1dd8f0dfcdd347aba432fbbdadbd99b1	0.345028	0.247673	0.004014	0.305873	0.452469
f73eae01618642e08c34edeba5847eee	0.000061	0.293478	0.037037	0.893204	1.000000
df64f7b6d391432fb4629b12640595ca	0.002658	0.054054	0.057592	0.925454	0.936275
4b3c1b71058d4445b73b39e012adf7d5	0.003973	0.032252	0.024845	0.879493	1.000000
c0c0de02709a4c06b5fb4ef6ffbf23eb	0.000940	0.123116	0.200000	0.074672	0.816327
64e92412c4c14046847b4077d6ac0981	0.003203	0.159073	0.020305	0.330020	0.956311
c00d0814f0b548c68817473b1605a375	0.000006	0.000000	0.000000	0.000000	0.000000
8b2ce24269c4409a9df4f608a0cf8778	0.005031	0.000000	0.000000	0.000000	0.000000
e4df9da84c404fb8adec83e0a21a63a8	0.003185	0.238629	0.126323	0.890278	0.992992
0840c9cb8bd34e60a65873c919a5ea8d	0.022146	0.012685	0.601770	0.280664	0.982609
680c2039565e4b0b8583a0001739efcf	0.000183	0.000000	0.000000	0.000000	0.000000
a3b90514e21c4ce8a96760d0880264bf	0.017360	0.176569	0.032895	0.063619	0.955975
fa1ad522cc66419b9a68848ab4977349	0.028125	0.011443	0.016000	0.260517	1.000000
e6cff602f6554ddb91f10e3e6729eaed	0.000019	0.000000	0.000000	0.000000	0.000000
587350b06a34433bb658a06c57792020	0.000593	0.004695	0.000000	0.000000	0.000000
c61dd764ec8e43e18e8d27cb728169bd	0.269686	0.027936	0.007030	0.601808	0.953372
382aae29a2d044089970437166cacbd7	0.001738	0.112388	0.147727	0.287129	1.000000
b1c659b54a794edf931edb77a5a703a7	0.000850	0.010909	0.250000	0.880705	1.000000
c155c69aff824afe9305061c24a57d2a	0.000243	0.218143	0.240000	0.913215	0.990099
7ec629dc942c41a0af159b034d4b2e7a	0.002365	0.244573	0.156805	0.861596	1.000000
f06902822ac04273a7f16d7e05edf40e	0.000011	3.375000	0.208333	0.235294	0.888889
4a2dfe5c0a62d6653bc4b66ce1847e8e	0.150752	0.091745	0.003992	0.200637	0.937679
f5b5eb0ecde24cd5a54da62f847ff17a	0.002105	0.063433	0.000000	0.000000	0.000000
d7e7a52de0ab4a7cbf3a1eac7cc51955	0.000308	0.020833	0.500000	0.252632	1.000000
4854330fd23c44bfa0f06b97b4d9fb34	0.007661	0.008354	0.010989	0.952104	0.978495
d3dba5c6bc0a419bb366b603b0ae4902	0.073504	0.010113	0.101813	0.651061	1.000000
f05e3773d9a44527affcb30bcdd2fe9f	0.000175	0.666667	0.000000	0.000000	0.000000
4330a8fec69f47c7b3220a8fd3559fdf	0.000453	0.003831	0.000000	0.000000	0.000000
ceaa8134f220452682e0acdc713ff535	0.052150	0.274397	0.213607	0.399765	0.556986
9c877ccc7a12479e839791e8fa2a6716	0.004360	0.006077	0.055556	0.948751	1.000000

Appendix C

Software Used

- Tsung 1.6.0
- Python 2.7.6
- MySQL 5.6.26
- Redis 2.8.6
- NGINX 1.9.5
- Ubuntu 14.04
- SciPy 0.17.0
- Pretio 1486cae0ce2d191a85c00e2edc3b257136266be7

Bibliography

[1] Theodore W Anderson and Donald A Darling. A test of goodness of fit. *Journal of the American statistical association*, 49(268):765–769, 1954.

[2] Apache Software Foundation. JMeter - graphical server performance testing tool. `http://jmeter.apache.org/usermanual/index.html`, 1998. Accessed on 2016-08-19.

[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Perform. Eval. Rev.*, 26(1):151–160, June 1998.

[4] BlazeMeter. Load Testing and APM - Accelerating Performance Tuning and Troubleshooting. `http://cdn2.hubspot.net/hubfs/208250/Whitepapers/LT-APM_FinalVersion.pdf`, 2016. Accessed on 2016-08-19.

[5] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[6] Frédéric Cazals and Chinmay Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1):564–568, 2008.

[7] Wesley Copeland. Pokemon Go launch suffered because it had 50 times more traffic than expected. `http://ca.ign.com/articles/2016/09/30/pokemon-go-launch-suffered-because-it-had-50-times-more-traffic-than-expected`, 2016. Accessed on 2016-09-30.

[8] Mark E Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions on networking*, 5(6):835–846, 1997.

[9] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

[10] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[11] HP. LoadRunner - User Guide. `http://lrhelp.saas.hpe.com/en/latest/help/PDFs/HP%20LoadRunner%20User%20Guide.pdf`, 2016. Accessed on 2016-10-01.

[12] Krishna Kant, Vijay Tewari, and Ravishankar K. Iyer. Geist: A web traffic generation tool. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, TOOLS '02, pages 227–232, London, UK, UK, 2002. Springer-Verlag.

[13] Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.

[14] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Softw. Eng.*, 32(11):868–882, November 2006.

[15] Stéphane Landelle. Gatling - open-source load testing framework. `http://gatling.io/docs/2.2.2/`, 2012. Accessed on 2016-08-19.

[16] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[17] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[18] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[19] Nicolas Niclausse. Tsung - multi-protocol load testing tool. `http://tsung.erlang-projects.org/user_manual.pdf`, 2001. Accessed on 2016-08-05.

[20] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *SIGCOMM Comput. Commun. Rev.*, 24(4):257–268, October 1994.

[21] Parag Pruthi and Ashok Erramilli. Heavy-tailed on/off source behavior and self-similar traffic. In *Communications, 1995. ICC'95 Seattle,'Gateway to Globalization', 1995 IEEE International Conference on*, volume 1, pages 445–450. IEEE, 1995.

[22] Gil Tene. Understanding latency & application responsiveness. Strange Loop 2015, 2015.

[23] Christian Vögele, Andre van Hoorn, and Helmut Krcmar. Automatic extraction of session-based workload specifications for architecture-level performance models. In *Proceedings of the 4th International Workshop on Large-Scale Testing*, LT '15, pages 5–8, New York, NY, USA, 2015. ACM.