Customer-Driven Cost-Performance Comparison of a Real-World Distributed System

by

Nicholas James Nickerson Turner
B.Seng, University of Victoria, 2013

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

© Nicholas Turner, 2019
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Supervisory Committee

Customer-Driven Cost-Performance Comparison of a Real-World Distributed System

by

Nicholas James Nickerson Turner
B.Seng, University of Victoria, 2013

Supervisory Committee

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

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

Abstract

Many modern web applications run on distributed cloud systems, which allows them to scale their resources to match performance requirements. Scaling of resources at industry scales, however, is a financially-expensive operation, and therefore one that should involve a business justification rooted in customer quality-of-service metrics over more commonly-used utilization metrics. Additionally, changing the resources available to such a system is non-instantaneous, and thus a reasonable effort should be made to predict system performance at varying resource allocations and at various expected workloads.

Common performance monitoring solutions look at general metrics such as CPU utilization or available memory. These metrics are at best an indirect means of evaluating customer experience, and at worst may provide no information as to whether users of a commercial application are satisfied with the product they have paid for. Instead, the use of application-specific metrics that accurately reflect the experience of system users, combined with research into how these metrics are affected by various tunable parameters, allows a company to make accurate decisions as to the desired performance perceived by their users versus the costs associated with providing that level of performance.

This thesis uses a real-world software-as-a-service product as a case study in the development of quality-of-service metrics and the use of those metrics to determine business cases and costing packages for customers. The product used for this work is Phoenix, a state-of-the-art social media aggregation and analytics software-as-a-service web platform developed by Echosec Systems, Ltd. The product will be tested under real-world conditions on cloud hardware with a minimal test harness to ensure a realistic depiction of live production conditions.

Table of Contents

Supervisory Committee....ii
Abstract....iii
Table of Contents....iv
List of Tables....vi
List of Figures....vii
Acknowledgments....viii
Dedication....ix
1 Introduction....1
  1.1 Phoenix Overview....1
  1.2 Thesis Objectives....2
2 Literature Review....4
  2.1 Workload Generation....4
  2.2 Software Predictions....4
  2.3 Cloud Predictability....5
  2.4 Website Usability....5
  2.5 Summary....6
3 Methodology....7
  3.1 Phoenix Architecture Details....7
    3.1.1 Input modes....7
    3.1.2 Event queueing....9
    3.1.3 Phoenix processing pipeline....10
    3.1.4 Pipeline timeouts....12
  3.2 Phoenix Testing Modifications....13
    3.2.1 Data API call mocking....13
    3.2.2 Internet access removal....14
    3.2.3 Websocket session activity....15
    3.2.4 Performance test harness....16
  3.3 Key Performance Indicators....17
    3.3.1 Time to first post....18
    3.3.2 Time to last post....18
    3.3.3 Websocket packet count....18
  3.4 Summary....19
4 Results....20
  4.1 Isolated input....20
  4.2 Single burst input, increasing search count....22
  4.3 Single burst input, increasing worker count....31
  4.4 On-off inputs....45
  4.5 System Performance Breakpoints....52
  4.6 Summary....54
5 Conclusions and Future Work....56
  5.1 Objective Evaluation....56
  5.2 Limitations....56
  5.3 Future Work....57

5.4 Recommendations.....57
Bibliography.....59
Appendix A – Single-Burst Workload Generation.....61
Appendix B – On/Off Workload Generation.....64

List of Tables

Table 1: TTFP/TTLP for 16-search trial......28
Table 2: Locations and timings of timeout bursts during 128-search trials......38
Table 3: Time to first post rate of increase for 128-search trials, workers 8-32......39
Table 4: Time to first post rate of increase for 128-search trials, workers 40-64......41

List of Figures

Figure 1: Phoenix Input Processing Paths....2
Figure 2: Quick searches per day (linear scale)....8
Figure 3: Streaming post counts for Superbowl (logarithmic scale)....9
Figure 4: Phoenix processing pipeline event listener sequencing....12
Figure 5: IPTables rules to disable Docker internet access....14
Figure 6: Performance test harness visual overview....17
Figure 7: Isolated single search packet dispatch timings....20
Figure 8: Isolated single search packet dispatch timings - closeup of packets 84-93....21
Figure 9: Single burst timeline, 8 workers, 16 searches....22
Figure 10: Single burst timeline, 8 workers, 24 searches....23
Figure 11: Single burst timeline, 8 workers, 32 searches....23
Figure 12: KPI comparison plot, 8 workers, 16 searches....24
Figure 13: KPI comparison plot, 8 workers, 24 searches....25
Figure 14: KPI comparison plot, 8 workers, 32 searches....25
Figure 15: Single burst timeline, 8 workers, 64 searches....28
Figure 16: Single burst timeline, 8 workers, 128 searches....29
Figure 17: Single burst timeline, 16 workers, 128 searches....31
Figure 18: Single burst timeline, 24 workers, 128 searches....32
Figure 19: Single burst timeline, 32 workers, 128 searches....33
Figure 20: KPI comparison plot, 8 workers, 128 searches....34
Figure 21: KPI comparison plot, 16 workers, 128 searches....35
Figure 22: KPI comparison plot, 24 workers, 128 searches....35
Figure 23: KPI comparison plot, 32 workers, 128 searches....36
Figure 24: KPI comparison plot, 40 workers, 128 searches....39
Figure 25: KPI comparison plot, 48 workers, 128 searches....39
Figure 26: KPI comparison plot, 64 workers, 128 searches....40
Figure 27: Single burst timeline, 32 workers, 256 searches....42
Figure 28: KPI comparison plot, 32 workers, 256 searches....43
Figure 29: On/Off timeline, 24 workers, 20x10 searches, 30 second delay....45
Figure 30: KPI comparison plot, 24 workers, 20x10 On/Off searches, 30 second delay. 46
Figure 31: On/Off timeline, 24 workers, 20x10 searches, 45 second delay....48
Figure 32: On/Off timeline, 24 workers, 20x10 searches, 60 second delay....49
Figure 33: KPI comparison plot, 24 workers, 20x10 On/Off searches, 45 second delay. 50
Figure 34: KPI comparison plot, 24 workers, 20x10 On/Off searches, 60 second delay. 50

Acknowledgments

I would like to thank:

- My family and friends, for supporting and encouraging me to press on with this project,
- Echosec Systems Ltd, for providing the resources, technology, time, and opportunity needed to complete this thesis, and
- Dr. Stephen Neville, for his advice and feedback in the creation of this work.

Dedication

To my parents, Marla and Lance, and my sister Rachel, who have always supported me.

1 Introduction

Modern, real-world distributed systems make extensive use of cloud environments to deliver cost-effective services to clients. By making use of commercial clouds, system developers can deploy applications to customers without needing to invest in and maintain their own physical infrastructure. Moreover, cloud environments provide an abundance of resources to developers, allowing software companies to solve many performance and scaling problems by paying for additional cloud computing capacity.

However, the use of cloud computing resources at industrial scale is not free. For example, Twitter Inc reported that their 'cost of revenue' for the year of 2018 was $964 million, which includes infrastructure costs [1]. A hypothetical data centre using 1000 copies of the default AWS server instance, an m5.large, would cost $840, 000 per year to operate [2] . While a company with a scalable distributed software product can theoretically achieve any desired level of performance through the use of cloud computing resources, doing so may not be financially practical.

To evaluate the business case for cloud resource levels, a means of evaluating performance from the business perspective must be derived. The concern of business is with the experience of their customers, which determines the likelihood for repeat or continued use of the SaaS product. As such, technical metrics such as CPU performance or network bandwidth must be replaced with customer-driven quality-of-service metrics such as response time or service reliability when comparing infrastructure cost with system performance.

Once suitable metrics have been identified, the company can experimentally determine how these metrics change at different resourcing levels. Various tunable parameters, such as input rates and service resourcing levels, can be identified and their impacts relative to their costs measured. The results of this research will allow the business to identify quality-of-service breakpoints that provide the most value to their customers at the lowest cost, providing a tangible competitive advantage to that company.

1.1 Phoenix Overview

Phoenix is the internal codename of a software-as-a-service (SaaS) web application owned and operated by Echosec Systems Ltd. Phoenix is capable of aggregating, analyzing, and visualizing data from 18 different online data sources (as of this writing), including popular social media platforms such as Twitter and Reddit, news aggregations services such as Bing News, and contextual data sources such as Wikipedia. Phoenix receives search requests from Echosec's customers, executes those requests against its plurality of social media providers, runs a variety of analytics on the data received, and returns both the posts and the analytics generated to the customer for review.

The core of Phoenix is built in PHP, making use of the Laravel web framework to receive and process web requests executed by Echosec customers [3] . Queries for data cause Phoenix to queue up data retrieval jobs, which are processed by the Laravel event queue. These jobs query the APIs of the various third-party data providers that Echosec

has partnered with, retrieving a fixed number of recent results for each of those providers. These results are then passed through a pipeline of data analytics jobs that perform various post-processing operations to add additional context and value to the results. Finally, once all these jobs are completed, the final results are streamed back to the customer's browser via websocket technology [4] to minimize the delay before those results are made available to the customer.

There are three paths through which data requests may move through Phoenix. The first is the method described above, where a customer executes a web request querying for a snapshot of data from a location or keyword. These direct web requests are called 'quick searches' by Phoenix. The second and third modes are both driven by 'saved searches', an alternate way for Phoenix customers to query for data. Saved searches allow the system to continuously fetch new data matching a search query as those data become available. This continuous retrieval is executed in two ways. The first is by periodically re-executing the saved search as though it was a new quick search, causing the system to retrieve all data available on the query, and then identifying results not found by previous re-executions. The second is by configuring a stream of data from the data provider, which will receive new data as soon as it becomes available. Streaming data is the preferred way to retrieve new results for a saved search, but is not available for all third-party data providers, necessitating the use of periodic re-execution to provide continuous results. See Figure 1: Phoenix Input Processing Paths for a visual overview of how these three types of input are managed by Phoenix.

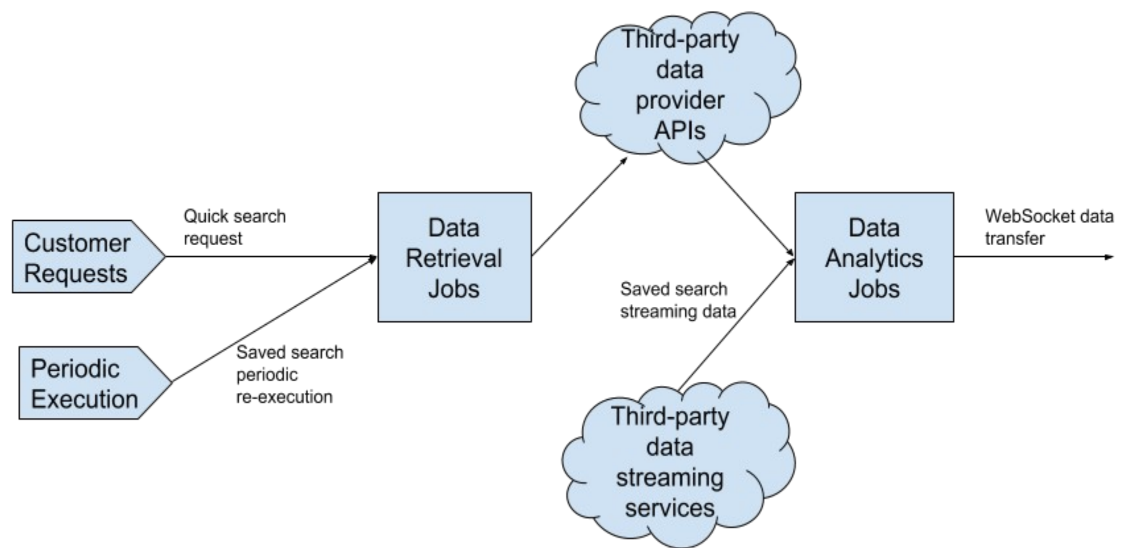

Figure 1: Phoenix Input Processing Paths

Each of the modules shown above are, in practice, groupings of many sub-modules with more narrowly-scoped functions, many of which which will be discussed in more depth throughout the system methodology.

1.2 Thesis Objectives

The goals of this thesis are:

1. To determine suitable quality-of-service metrics for Phoenix, driven by customer experience,
2. To examine the behaviour of these quality-of-service metrics under a variety of operating conditions at varying costs,
3. To demonstrate that these quality-of-service metrics shift between various non-stationary modes of operation under varying conditions, and
4. To develop a means of controlling the mode of operation into which these metrics will shift, based on the costs available to the customer.

These goals will be executed against a functional copy of the Phoenix platform running in a limited test harness, to demonstrate the real-world applicability of these objectives. The results of this thesis will be used by Echosec System Ltd to guide the further development of Phoenix.

2 Literature Review

In this chapter, previous research into workload generation, software predictability, and website user experience will be reviewed and discussed.

2.1 Workload Generation

Workload generation is the process of automating the production of workload inputs to a software application [5]. Performance evaluation is critical for complex software applications, and to extract data on software performance in a consistent and reliable manner, workload generation is essential.

Workload generation can be done in two different ways. Either real inputs from the system can be captured and replayed, or real inputs can be processed into a workload model that can in turn generate synthetic workloads. Workload modelling has the advantage of allowing for the testing of hypothetical workloads that have not yet been observed in the real world, allowing for flexible sensitivity analysis [5]. Additionally, workload modelling allows the testing of data at throughput rates in excess of those seen in the live system. Replaying data repeatedly can function similarly to increase throughput, but if the system under test has special handling for duplicate or repeated requests, such as caching, repeated replay of past workloads may fail to generate an accurate performance report.

Creating accurate synthetic workloads for web servers poses several principle challenges, which centre around the variability of Internet traffic. Web requests arrive in an inconsistent manner, and furthermore the self-similarity of these requests can add additional impact to the performance of the system under test [6]. The SURGE tool [6] was among the first to study these challenges in the context of internet workload generation, and proposed an 'ON/OFF' model to simulate requests to specific endpoints. This model simulates a user's interaction with a web page as a series of URL requests with delays. Each request is an 'ON' period, representing active load on the server, while an 'OFF' period exists after each request and after each series of requests, representing user consumption and review of each response.

2.2 Software Predictions

Much work has been done in the past on the ergodicity of theoretical software queueing networks. In [7], the author discusses stability of queueing networks, both when handling bounded queue lengths and handling long-tailed queues. In [8], an open queueing network is discussed and demonstrated to be ergodic under a set of reasonable preconditions. In [9], the steady-state performance of Petri nets is analyzed as a throughput of transitions. The author notes the difficulty in demonstrating ergodicity of a system model in the general case, and provides a definition for a concept of 'weak ergodicity', which is applicable to finite-energy environments without a known probability density function.

The existing body of research focuses primarily on simulated results from theoretical systems. In many cases, such as [10], these theoretical systems are modelled from

proposed real-world architectures, with the intent of predicting the performance characteristics of a system before that system is built. This predictive analysis allows software architects to alter the design of potential new systems before those systems are implemented, at which point the costs of altering the system are significantly cheaper than post-development. However, these solutions do not leverage additional information available in an already-implemented system. Other solutions reduce and simulate the system [11] [12], which provides an effective mechanism for gathering information on large systems before or after release, but may neglect real-world concerns not present in the simulation. To make use of these techniques to analyze an existing software solution, a suitable queueing model would have to be reverse-engineered from the completed system, which risks the loss of real-world behaviours present in the actual system that are not reflected in the model.

Less research exists for analyzing existing software solutions. In [13], the costs associated with scaling software applications on cloud providers are formally measured. This article focuses on utilization of key cloud resources, such as memory or CPU use, which directly relate to cost of server space. While this directly impacts the cost of the system, the use of general measures such as CPU and memory utilization do not reflect knowledge of internal system components, such as the distribution of specific processes or the impact of various workload patterns, which can be extracted from a more in-depth analysis of a software application. Evaluating a software application using metrics specific to that installation is difficult to apply more broadly, which explains the current lack of these specific metrics in the field.

2.3 Cloud Predictability

While research into system predictability post-implementation is less common, some research has been done into the behaviour of cloud systems at industrial scales. In [14], a dynamic resource management process for distributed cloud services is proposed. This process takes virtual machines, or VMs, and finds a performant way to deploy those VMs onto physical hardware. In [15], a testbed-driven solution is used to experimentally validate possible cloud utilization strategies, providing a middle ground between pure simulation of a real system and experimental testing of that system.

VM-driven cloud systems provide an opportunity to examine software applications in a more real-world environment while still using general metrics for evaluation and comparison, with [14] using CPU, memory, and bandwidth usage and variability with varying weights to evaluate VM performance. The use of general metrics allows these techniques to be applied to a broad range of applications without requiring additional information about the intended functionality of the applications in question.

2.4 Website Usability

Usability concerns have been at the forefront of website design and analysis. In [16], customer satisfaction with a website is driven by that site’s usability, which is measured through five major factors:

1. Ease of understanding the website's features and contents,
2. Simplicity of use for new users,
3. Speed with which users can find the information they want,
4. Ease of navigation to find desired results,
5. Ability to control their actions

While the individual factors change, many other studies echo this multi-dimensional approach to website usability. In [17], a survey of 273 US female online shoppers broke its questions into six factors: web appearance, entertainment, informational fit-to-task, transaction capability, response time, and trust. This study concluded that all six dimensions were independent, confirming the multi-dimensionality of the results, but that not all factors were necessarily of equal importance, with customer satisfaction being driven more by website informativeness than by visual or emotional appeal.

In [18], the importance of System Quality, Trust, Extension Quality, and Propriety of Content to apparent customer satisfaction with a B2C website were analyzed. Trust was found to be the most important when interacting with E-commerce websites, as may be expected for a website handling a user's financial data, as was extension quality, defined as the overall support delivered by the website. System quality, including performance characteristics, were found to be less important, which the authors suspect is due to system quality having an indirect influence, and propriety of content was found to be unimportant in user satisfaction. These findings disagree with their own literature review, suggesting that the weighting of different factors differs with the industry and the purpose of the website in question.

In [19], the authors report that 40% of internet consumers wait no more than 3 seconds for a website to load or respond before abandoning their request. This report suggests a threshold-based approach to performance metrics, concerned with reducing the number of users whose experience falls behind a given threshold. Users whose response times exceed this threshold will likely presume that their request will never complete, and may retry the request, which may add to an existing load problem and cause further delays for other users.

2.5 Summary

In this chapter, past and present research into workload generation, software predictability, and website user experience was reviewed. The On/Off workload pattern was discussed as a realistic solution to generating synthetic workloads. Performance prediction was discussed, with current research focused primarily on simulated or modelled software, often prior to implementation, rather than studying live software applications in real environments. The facets of a usable website were reviewed, and a maximum delay of 3 seconds before data are returned was recommended.

3 Methodology

The goal of this work is to determine suitable quality-of-service metrics for Phoenix, to then extract those metrics from the system under load, and finally to analyze them to find performance breakpoints of business use. In this chapter, the architecture of Phoenix will be discussed at a high level and used to determine suitable metrics, and the changes that must be made to the application to support stress-testing in a production-like environment will be enumerated.

3.1 Phoenix Architecture Details

Before any further discussion can be had on metrics or proposed changes, more information must be disclosed about the structure of Phoenix and its handling of customer input. Much of the architecture of Phoenix is covered by Echosec's patent [20], but this section will cover the components of Phoenix most relevant to this work in greater depth.

3.1.1 Input modes

As summarized in the Phoenix overview and architecture, there are three primary modes of input that drive the Phoenix platform. These inputs are:

1. Quick search requests, caused by customers performing an explicit and immediate request for recent or historical data.
2. Saved search streaming data, caused by customers making a request for future data on a service that supports streaming.
3. Saved search polling requests, caused by customers making a request for future data on a service that does not support streaming.

Each input form generates a different behaviour pattern in Phoenix, as examined below.

Quick search requests are generated explicitly and directly by customer interaction with the system. As a result, quick searches are an erratic and variable process. See Figure 2: Quick searches per day (normalized, linear scale) for a day-by-day summary of quick searches generated on Phoenix by real customers for the 30 days following September 24th.

Figure 2: Quick searches per day (normalized, linear scale)

Figure 2 shows the presence of a 7-day cycle, with weekends being significantly lower-traffic than other days, suggesting that many customers utilize Echosec primarily during work days. However, certain days (such as Sept 27th and Oct 17th) demonstrate extreme spikes of 3-5x the usual traffic, showing how heavily the ingestion of quick searches can vary. The variability of this work load suggests that deviations from standard performance in the system may be caused by these abnormally-high spikes.

From conversation with Echosec developers, there is a possible explanation for some of the spikes seen in Figure 2. Echosec partners with training organizations around the world to train active and potential customers on Phoenix. This training consists of training sessions, during which many users will perform similar searches over a brief window in response to the trainer's instructions. This behaviour suggests that the ON/OFF model [6] is a reasonable choice for simulating quick search requests.

Saved search streaming data is provided from third-party services via streams. This data source generally streams in at a consistent rate. However, real-world events may cause certain streaming sources to significantly increase their throughput without any change from Phoenix itself. Figure 3: Streaming post counts for Superbowl (logarithmic scale) shows the number of streaming posts generated per day by a search for the keyword 'superbowl'. The date of the Superbowl itself is clearly visible from the graph, with a peak of over 800 000 posts received in a single day, a significant shift when the Phoenix platform consumes 10-15 million posts each month.

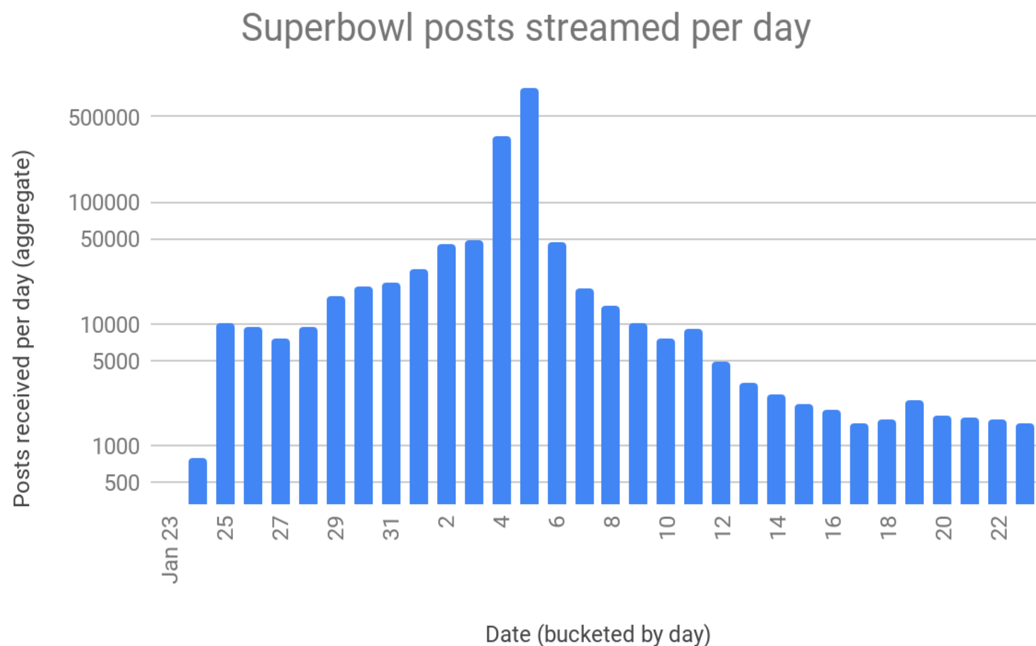

Figure 3: Streaming post counts for Superbowl (logarithmic scale)

Polling requests are the most consistent in terms of data throughput. Phoenix tracks the number of polling requests that are actively being performed and limits the number of simultaneous polling operations to a configurable value. Once the maximum number of concurrent polling requests are executing, no further polling requests will be run until at least one polling request fully executes and leaves the work queue. Each individual request will retrieve a capped number of results for each third-party service, limited by that service's API limitations. Between the fixed number of concurrent requests and the fixed number of results per request, polling results are the least-variable source of work for Phoenix.

Of the three request types discussed in this section, the most variable one is quick search requests. Additionally, quick search requests are directly visible to the customer, being performed in response to their direct input and leaving them waiting until a response is received, and thus the most sensitive to quality-of-service concerns. As a result, this work will focus primarily on simulating and analysing the results of quick search requests on the Phoenix architecture.

3.1.2 Event queueing

The search processing elements of Phoenix are driven by an asynchronous queueing architecture, following a standard pattern for the handling of queued actions:

1. The system generates an event.
2. One or more event listeners registered to that event will be triggered.
3. The triggered event listeners are placed on a queue as a job.
4. When a job executor is available, the queued event listener will be taken off the queue and processed by calling its '`handle()`' function.

5. Depending on the listener in question and its results, zero or more additional events will be triggered, potentially starting the process over again.

Each job executor is a PHP process running the Laravel framework, polling for jobs on a 1-second interval. All job executors run the same code, and thus all job executors are capable of executing any event listener queued by the system. The number of job executors per server node is static in the current deployment, with the number of server nodes scaling in response to high-level system monitoring of statistics such as CPU or memory utilization.

Laravel supports multiple queues, but does not require them. Because each executor is capable of running all possible Laravel event listeners, Laravel does not require a separate queue for each type of event listener, and each job executor is capable of listening to multiple queues at once, pulling a job from the highest-priority queue that has work available. At present, Phoenix defines 6 queues: `quick`, `default`, `poll`, `notification`, `detection`, and `metrics`, which act as broad categories that event listeners are assigned to. All worker nodes run by Phoenix execute all 6 of these queues, in the priority order listed above. At present, the Phoenix processing pipeline leverages two of these queues, `quick` and `default`. More on these will be discussed after reviewing the event listeners involved.

3.1.3 Phoenix processing pipeline

Summarizing the entire processing capabilities of Phoenix is out of scope for this document. What follows is a high-level summary of the core events involved in executing a quick search in Phoenix, beginning with a customer's search request being received by the application and ending with the successful return of results data to that customer.

1. A web request is received at URL `/api/v1/search`.
2. A `Search` object is created in the database, and a `SearchCreated` event is triggered.
3. This `SearchCreated` event causes a single `ExecuteQuickSearch` event listener to fire, immediately causing a more general `ExecuteSearchQuery` event to trigger.
4. This `ExecuteSearchQuery` event again causes a single `QueryProviders` event listener to fire, which performs some initial sanitization on the search and then triggers a `ProviderSearch` event.
    i. While not relevant to this discussion, polling search requests would also trigger the `ExecuteSearchQuery` event. The apparent redundancy in this chain of events is largely to support polling searches in addition to quick searches.
5. The `ProviderSearch` event causes many event listeners to fire, each one an implementation of the `ExecuteSocialQueryAbstract` event listener. One event listener is registered to this event for each API data source available to Phoenix, including some data sources with two or more APIs to retrieve different types of data. At present, there are 25 different data sources available to the application.

   i. All 25 data sources follow similar processing pipelines. To avoid discussing all 25 separately, 3 exemplar API sources and their corresponding event listeners will be discussed: `ExecuteRedditQuery`, `ExecuteFoursquareQuery`, and `ExecuteFlickrQuery`.
6. Each data source event listener decides whether or not to execute on the search, based on the search's parameters and filters. If the search is suitable for execution, the data source event listener will execute synchronous HTTPS calls to the API as needed to retrieve relevant data, and will then trigger the `ProcessResults` event.
   i. For instance, `ExecuteFoursquareQuery` will immediately terminate on a keyword search, as Phoenix is not capable of performing keyword searches on Foursquare.
   ii. Likewise, `ExecuteFlickrQuery` will immediately terminate if the search does not have 'Flickr' in the user's desired list of providers.
   iii. If no termination conditions are found for the `ExecuteRedditQuery` listener, then this event listener will execute a synchronous call to the Reddit API to retrieve search data from Reddit, and will then attach that search data to the subsequent `ProcessResults` event call.
   iv. Each successful data source listener will generate a separate `ProcessResults` call. No attempt is made to merge these calls together, and so each service is functionally independent for the rest of the search processing.
7. The `ProcessResults` event causes the `ExecuteSearchPostProcessingPipeline` event listener to fire. This event listener initializes a series of additional processing steps on the data, including filtering out duplicate data, identifying content type, or applying object detection to images. Once complete, the pipeline triggers a `SearchPostProcessingCompleted` event.
8. Finally, the `SearchPostProcessingCompleted` event fires a `BroadcastSearchResults` event listener. This event listener causes the data received from the data APIs and processed by the search processing pipeline to be dispatched via websocket technology, which will forward the results data to the customer's web browser, allowing their search results to seamlessly arrive as they are made available.

This process is summarized in Figure 4: Phoenix processing pipeline event listener sequencing.

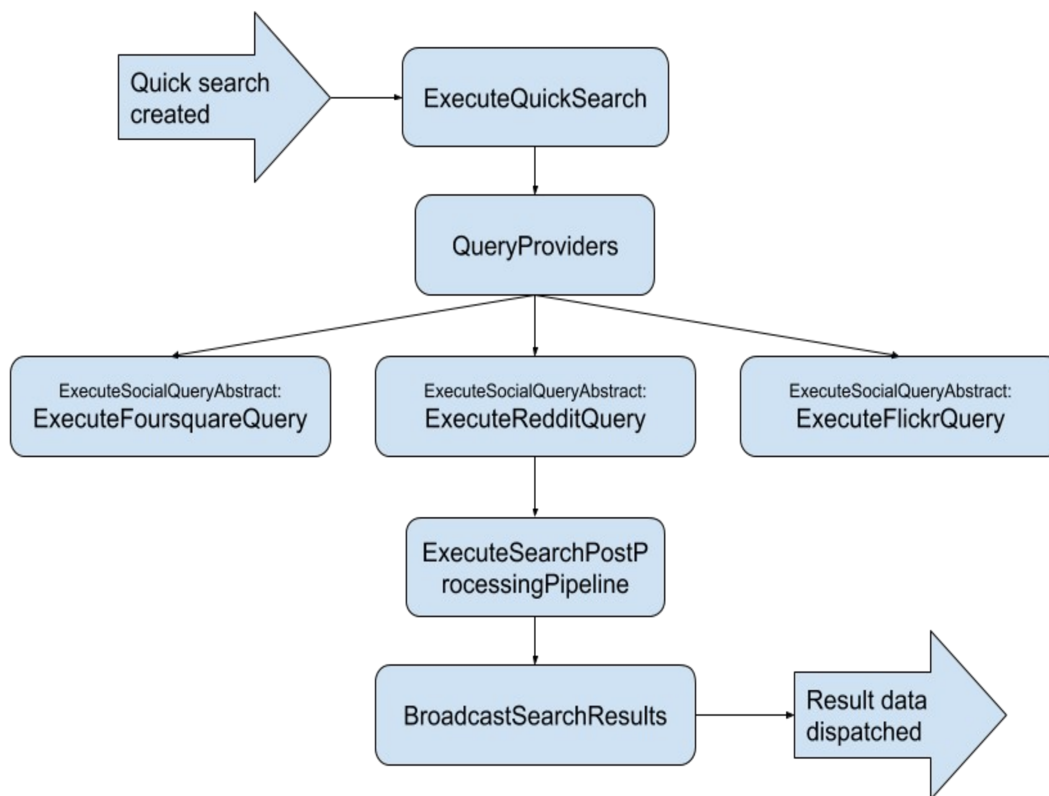

Figure 4: Phoenix processing pipeline event listener sequencing

Each event listener in Figure 4 is associated with one of the six queues discussed in Event queueing. All instances of a listener will be queued on exactly one of these six queues, determined by a constant embedded in the listener. At present:

- `ExecuteQuickSearch` and `ExecuteSearchPostProcessingPipeline` run on the `quick` queue.
- `QueryProviders`, all `ExecuteSocialQueryAbstract` implementations, and `BroadcastSearchResults` run on the default queue.

3.1.4 Pipeline timeouts

One final note about the Phoenix architecture is how queue length and timeouts are handled. Laravel queues are implemented in Redis via the List construct, which is very nearly unbounded, with limits from both theoretical queue length and memory space far exceeding any reasonable threshold that Phoenix can achieve.

A more relevant limitation is the timeout on each data source event listener. The parent class of all these event listeners, `ExecuteSocialQueryAbstract`, defines a 60-second timeout period through the use of the Laravel `retryUntil()` method. If the event listener is not processed until 60 or more seconds after it has been created, it will be dropped instead of executed. Notably, this is checked when the listener is dequeued, and thus the length of the queue in Redis does not reflect the number of listener jobs that may have timed out.

3.2 Phoenix Testing Modifications

The goal of this work is to assess quality-of-service metrics in a real-world environment. Making significant changes to the application would reduce the applicability of these results to the system they were generated from. Nevertheless, certain changes to support the testability of Phoenix in a high-load environment are necessary for both the testing itself and as a result of practical concerns surrounding that testing.

3.2.1 Data API call mocking

Under normal operation, the Phoenix retriever logic performs REST-based API calls to a wide variety of external services, including contacting data providers such as Twitter or Webhose, but also including other external tools to control feature flags, manage social authentication, process billing updates, and many others. These services are essential to the operation of the Phoenix platform, and removing these calls entirely would be an extremely expensive operation and would render any tests non-representative of the system's real-world performance.

However, many of these services also have costs associated with calling them. For instance, Twitter's public API allows only a certain number of requests per 15 minutes before it terminates access to the calling account, while Webhose deducts a charge for each search performed against a credit, which must be paid off before it is entirely depleted. During standard business operation, these costs are managed by customer subscription fees, which grant access to a finite number of search requests, effectively throttling the cost of these queries. During the stress testing required to execute this analysis, however, those costs would be significantly higher than during normal system operation and would have no corresponding repayment to offset them. As a result, running a full-system query that contacts these third-party APIs would generate unacceptable charges to Echosec and potentially interrupt the availability of core services, and must therefore be avoided.

The solution to this costing problem is to mock the API calls. Phoenix executes the majority of its web requests through the popular Guzzle [21] library. By taking use of Laravel's IoC container, the Guzzle client can be replaced with a mock client that executes requests by retrieving and returning the contents of files on the local file system instead of contacting remote third-party services, avoiding the costs of contacting remote services entirely. This mock solution is capable of simulating variable delay on requests, and can provide error pages as appropriate. The delay on requests and probability of error pages can be determined in advance by running a small, representative sample of requests on each service.

Note that the mocked API calls will return data, but the data returned will not necessarily match the parameters of the search that made the request. For example, if the only file available for Twitter contains data from Ottawa, that file will be returned even if the search request was for Toronto. At present, Phoenix attempts to filter out posts with precise geographical information that do not match the requested location, which means that requests for specific locations may fail to yield data when used in conjunction with

this mock client. Search requests that contain only keyword or author filters do not attempt any sort of filtering in this manner. As a result, the workload generator should focus primarily on keyword searches to avoid purging data from the system.

Further note that the mocked API calls will return a very limited set of data. Most services have only a single static file available to return, which means that all search requests will receive the same set of data. This repeated processing of data does not affect the overall performance of quick searches, however, because Phoenix does not attempt to cache or otherwise retain quick search results between executions for privacy and data security reasons.

3.2.2 Internet access removal

With all functional social media API calls mocked to return static data, the testing environment can safely disable all further calls from the worker to the internet. Phoenix leverages third-party tools in a variety of environments, including authentication via single sign-on, web socket dispatch of posts to active browsers, and feature verification. None of these third parties are necessary to execute the core quick search functionality under test, and any one of these could be adversely affected by the volume of testing involved. To avoid unnecessary internet traffic, the worker's Docker image was blocked from accessing the internet at large, its outgoing traffic restricted to only other Docker images through the use of IPTables as shown Figure 5: IPTables rules to disable Docker internet access.

```
Chain DOCKER-USER (1 references)
target     prot opt source               destination
ACCEPT     all  --  172.18.0.9           172.18.0.15
ACCEPT     all  --  172.18.0.9           172.18.0.14
ACCEPT     all  --  172.18.0.9           172.18.0.13
ACCEPT     all  --  172.18.0.9           172.18.0.12
ACCEPT     all  --  172.18.0.9           172.18.0.11
ACCEPT     all  --  172.18.0.9           172.18.0.10
ACCEPT     all  --  172.18.0.9           172.18.0.9
ACCEPT     all  --  172.18.0.9           172.18.0.8
ACCEPT     all  --  172.18.0.9           172.18.0.7
ACCEPT     all  --  172.18.0.9           172.18.0.6
ACCEPT     all  --  172.18.0.9           172.18.0.5
ACCEPT     all  --  172.18.0.9           172.18.0.4
ACCEPT     all  --  172.18.0.9           172.18.0.3
ACCEPT     all  --  172.18.0.9           172.18.0.2
DROP       all  --  172.18.0.9           anywhere
RETURN     all  --  anywhere             anywhere
```

Figure 5: IPTables rules to disable Docker internet access

With these changes in Figure 5: IPTables rules to disable Docker internet access made, one further change must be made to the application due to how a lack of internet access interacts with Phoenix's websocket broadcasts when sending result data to the customer.

Normally, dispatching results to the client via websockets requires one or more network calls, as the maximum size of a single payload on Echosec's websocket provider is smaller than the usual size of Phoenix's data results for each service. As a result, Phoenix will normally send a single websocket payload with as many results as it can, then chain into another recursive call to dispatch a websocket packet with the remaining data, a process that is repeated until the remaining data can fit within a single packet. Disabling internet access significantly affects this process by causing the first websocket dispatch to time out, which means that the dispatch call takes considerably longer than usual. Additionally, once the dispatch eventually times out, the resulting exception causes the call to not recurse, resulting in no further attempts to send data over the web socket. The combination of these two factors significantly affects system performance, resulting in all available worker nodes blocking themselves on failed websocket dispatches and greatly reducing parallel execution in Phoenix's job executors. To resolve this concern, the `PushSearchResults` event was modified to no longer broadcast itself over the web, but to instead mock out the dispatch with a brief delay. This change allows all websocket dispatches to resolve as though the internet was still available, allowing for an accurate count of packet dispatches and avoiding any additional slowdown due to essential network calls timing out.

3.2.3 Websocket session activity

One further change to Phoenix's websocket broadcasts is required, however, in addition to the mocking described above. Before attempting to dispatch packets via websocket, Phoenix checks whether the user who executed the original search request is is still 'active'. This activity check is performed with the aim of reducing the number of websocket broadcasts made, lowering total network overhead and cost, by only sending when a client is believed to be actively listening for websocket data. Phoenix performs this activity check by querying the user's last active session from the database, which is refreshed via client interaction with the browser. By default, no results will be sent over the web socket if the user has been inactive for 6 minutes.

The tests undertaken for this work will frequently require more than 6 minutes to fully resolve. As a result, to ensure that results continue to be sent, the workload generator would either need to simulate periodic activity to keep the last active session current, or the timeout value would have to be increased. Notably, generating search requests does not refresh the active session, and so the workload generator would have to identify and execute a separate periodic call to keep the active session refreshed. The call to this endpoint is performed every 4 minutes and does not involve the work queue, and thus has a minimal effect on the performance of the job executors that are the primary focus of this work. To avoid adding additional complications to the workload generator, the broadcast session timeout has been increased from 6 minutes to 120 minutes, effectively preventing broadcast sessions from timing out and ensuring that Phoenix will always attempt to send result packets.

Note that the session activity timeout is not the same as the timeouts applied to individual jobs in the system. The session activity timeout applies only after all results have been retrieved and processed by the system, and is checked immediately before the

recursive call to dispatch results via the websocket. Additionally, during normal system execution with a valid client waiting for results, this timeout will never be hit, as the separate workflow mentioned above will keep the client's session active and prevent a session activity timeout from occurring. As a result, this timeout is not believed to significantly contribute to the KPIs of Phoenix, and so removing it from consideration will not overtly alter the behaviour seen from the trials in this work.

3.2.4 Performance test harness

Many performance monitoring solutions analyze generic summary statistics collected from the servers or nodes executing the application. These statistics include CPU utilization, amount of available RAM, hard disk IOPS, and so forth. Summary statistics have the advantage of being available for all software applications. However, summary statistics provide only limited information about individual system components, and are of minimal use in identifying bottlenecks in software applications when multiple parts of the application are running on the same node and are thus both captured in the same set of system metrics.

For higher-resolution statistics on system components, a custom metric tool can be employed to report on the performance of each component of the system. Thanks to the efforts of Michael Anderson, such a tool already exists for Phoenix. This tool reports on the processing time of each nested operation in the system, allowing for a more fine-grained investigation. The tool is developed using the popular open-source Jaeger tracing library [22] backed by a Cassandra database [23] to provide efficient storage for the vast number of actions associated with a sizeable workload generation. The tool is injected into the Phoenix application through a Laravel service provider, the OpenCensus provider, using Laravel's eventing system to initialize and resolve tracing spans around each event in the application. Results are received by using the Jaeger web interface on port 16686 to query for specific events and manually inspect low-throughput events, or accessing the Cassandra database directly to retrieve bulk data for large-scale results processing. The graphs and discussions throughout this work are primarily generated through bulk export of Cassandra data.

See Figure 6: Performance test harness visual overview for a visual representation of the relationship between the workload generator, performance test harness, and core architecture of Phoenix as tested in this work.

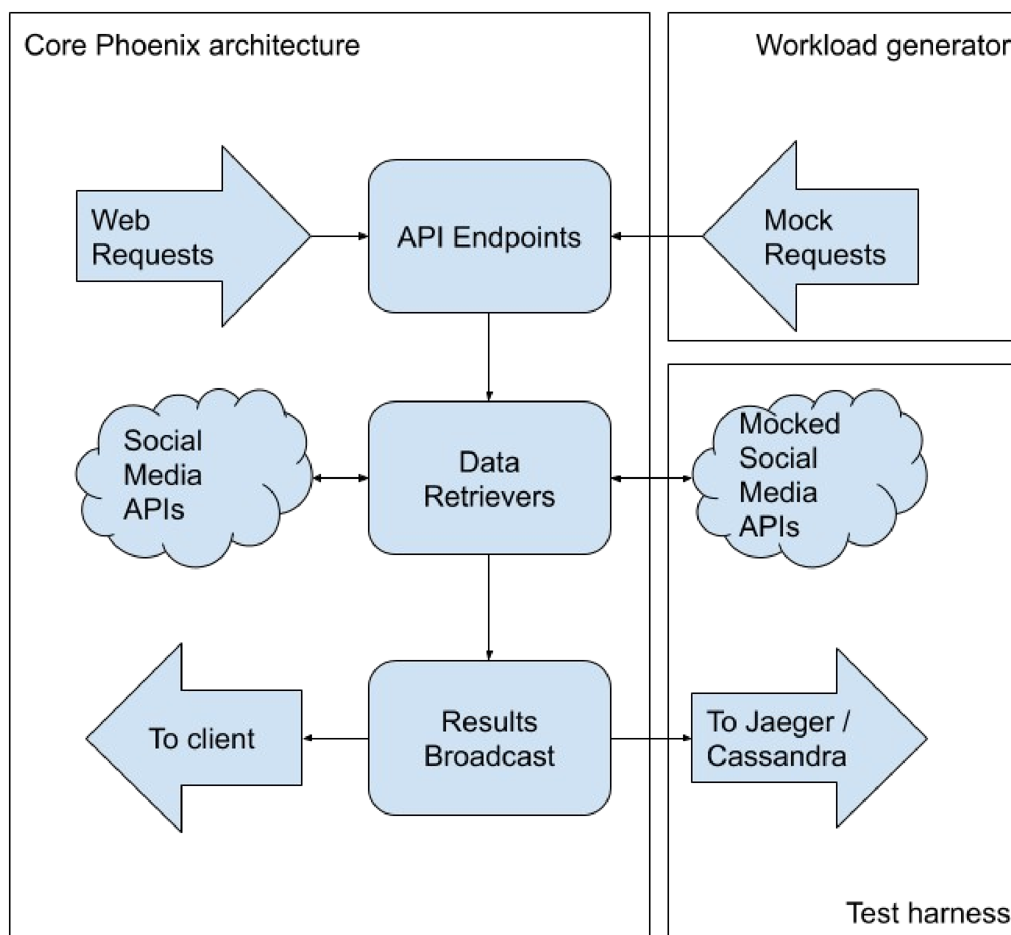

Figure 6: Performance test harness visual overview

3.3 Key Performance Indicators

To discuss quality-of-service metrics from Phoenix, the key performance indicators (KPIs) that must be established. These values are primarily driven by user experience, reflecting Echosec's goal for Phoenix to present a responsive and enjoyable experience to its customers, and influenced by market research that indicates a strong competitive advantage in having an expedient and reliable interface.

As discussed during literature review, many different factors determine apparent quality of service. Apparent performance to the customer in the form of response times to queries has been identified as a relevant quality by both previous studies and by the Echosec customer support team, who have flagged the system's quick responses to requests as a key differentiator with competing products. Additionally, response times that are slower than average are seen by both developers and customers of Echosec as a primary indicator of possible system problems or outages. This concern surrounding response time leads to the two primary KPIs of relevance to this work: 'time to first post' and 'time to last post'. Additionally, concern over data reliability and arrival leads to the third KPI discussed below: 'websocket packet count'.

3.3.1 Time to first post

This is the time interval between when a quick search request is received by the system, and when the first post is transmitted back to the client. Customers cannot perform any meaningful action or analysis of a search before at least one result has been received, making this the shortest period of time between a customer performing a request and the customer being able to interact with it. Additionally, the appearance of the first post is a clear visual indicator that the search has been formatted correctly and that no technical errors have occurred, providing the customer with the confidence needed to continue waiting for further results to arrive.

This measure will begin at the time the search is dispatched to the application. It will end when the first response packet containing the search's ID is identified. As a result, this measure will depend on the response time of the fastest third-party data service as well as the time needed for a post to traverse the processing pipeline.

3.3.2 Time to last post

This is the time between when a quick search request is received by the system, and when the last post is transmitted back to the client. While customers may begin interacting with a search as soon as some data has arrived, any summary or full-fledged analysis cannot begin until all posts have been received, and the user experience will be disrupted if the customer is operating on the search when new data flows in.

This measure will begin at the time the search is dispatched to the application. It will end when the final response packet containing the search's ID is identified. This time is, by definition, longer than the time to first post, with the further delay being incurred as the system must wait for the slowest service to respond and the time needed for all posts to move through the processing pipeline.

Note that this measure cannot easily be computed in real time, as identifying the 'last packet' in real time is not possible without additional metadata tracking that is not present in the Phoenix architecture. This limitation does not affect the results of this work as data analysis is being done after the fact, but will be relevant for any functional changes proposed that would affect real-time performance adjustments. This limitation provides further impetus to the importance of this measure; as the customer cannot clearly identify when the last post has arrived, the time to last post is a period of uncertainty during which a customer may incorrectly believe they are operating on a complete set of data, and being aware of this period is vital for the Phoenix support team.

3.3.3 Websocket packet count

This is the number of websocket packets dispatched by Phoenix to the customer as a result of a single quick search request. Each websocket packet sent in this manner contains one or more posts relating to a single data provider, along with any additional analytics or context generated by the Phoenix processing pipeline. Phoenix enforces a cap on the size of packets that may be dispatched in this manner due to third-party constraints, and so a single data provider may generate multiple websocket packets depending on how large the individual result posts are.

The number of websocket packets successfully dispatched by Phoenix acts as a measure for data completeness. When Phoenix is not able to keep up with its event queues, results may be dropped in the system, which will be seen by the customer as a reduced quantity of data. Loss of data is of high concern to Echosec, as missing results greatly reduce customer confidence in the functionality of the system.

Because the static mocked data provider API calls discussed above push the same data through the system on each search query, Phoenix will generate the same number of websocket packets for all search queries. From experimentation, a successful execution will generate exactly 93 packets in this testing environment. A packet count below 93 indicates the presence of lost data, and furthermore a packet count of 0 indicates that no data whatsoever was received by the customer.

In practice, this work will consider the packet count to be in one of three states: complete, partial, and absent. A complete packet count indicates that all 93 packets expected by the customer were dispatched, and is the normal and expected operating state for Phoenix. A partial packet count indicates that only a subset of the 93 packets were dispatched. Partial packet counts may still have useful TTFP / TTLP metrics available, but these values must be considered carefully, especially the TTLP, because they are both derived from the timings of packet dispatches, and some packets did not dispatch successfully. The extreme case would be only a single packet being dispatched, which would lead to TTFP and TTLP both being exactly equal, as the first and last packet are one and the same, which will generally result in the TTLP being deceptively low relative to completed searches. Finally, an absent packet count means that neither TTFP nor TTLP may be computed for the request, as no packets were dispatched at all and thus there is no secondary timing to end the window. Searches with no result packets dispatched represent a complete loss of data and a frustrated end user.

3.4 Summary

In this chapter, the methodology by which the rest of this work will operate was discussed. Details on the functionality of Phoenix as it relates to this work were established, including the available types of input and the mechanics of Phoenix’s job queueing structure. The modifications made to Phoenix to support stress testing and metrics extraction were discussed. Finally, the customer-facing metrics that will be used throughout this work were listed and explained.

4 Results

This chapter will discuss the results of many of the trials executed on the Phoenix platform, where discussion is relevant. Four groups of trials will be executed in this chapter. The first will test the behaviour of Phoenix when subjected to isolated, single search requests, which will be used to set a level of baseline performance when no other requests are being executed. The second group will test behaviour of the system when subjected to an increasingly-long stream of search requests with a constant number of workers, while the third group will use a constant number of search requests with a variable number of workers. Finally, the workload generator will change to an On/Off strategy to determine how Phoenix's behaviour changes when subjected to bursts of inputs as opposed to a constant stream.

4.1 Isolated input

This trial consisted of the Phoenix application being provided with a single search 10 times, with its queues fully flushed between each search. The objective of this trial is to generate a baseline predictor for the desired KPIs for a search being run in isolation. Under these conditions, each search generated exactly 93 websocket packets with predictable timings, as shown in Figure 7: Isolated single search packet dispatch timings.

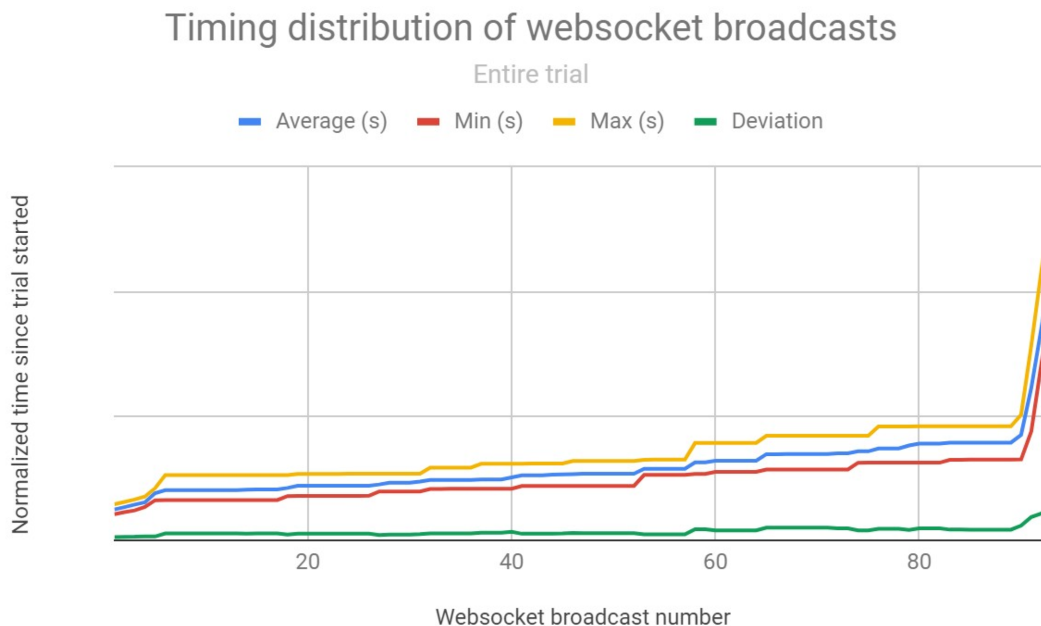

Figure 7: Isolated single search packet dispatch timings

On average, the time to first post for a single search during this trial was 2.47 seconds with a standard deviation of 0.26. The average time to last post was 21.89 seconds with a standard deviation of 2.63. However, as shown by the above graph, the tail behaviours are not necessarily indicative of the general behaviour of the graph. Taking samples at 10% and 90% gives us values of 4.03 seconds and 7.84 seconds, respectively.

Note that this graph does not invalidate the original, customer experience-driven KPIs. A customer with 90% of the data received is still in possession of an incomplete search, regardless of how much more quickly those results become available.

Additionally, while the graph shows a very dramatic increase in delay for the last few posts, this increase is highly consistent across all ten runs. Figure 8: Isolated single search packet dispatch timings - closeup of packets 84-93 shows a close-up view of the final 10 packets of Figure 7 above.

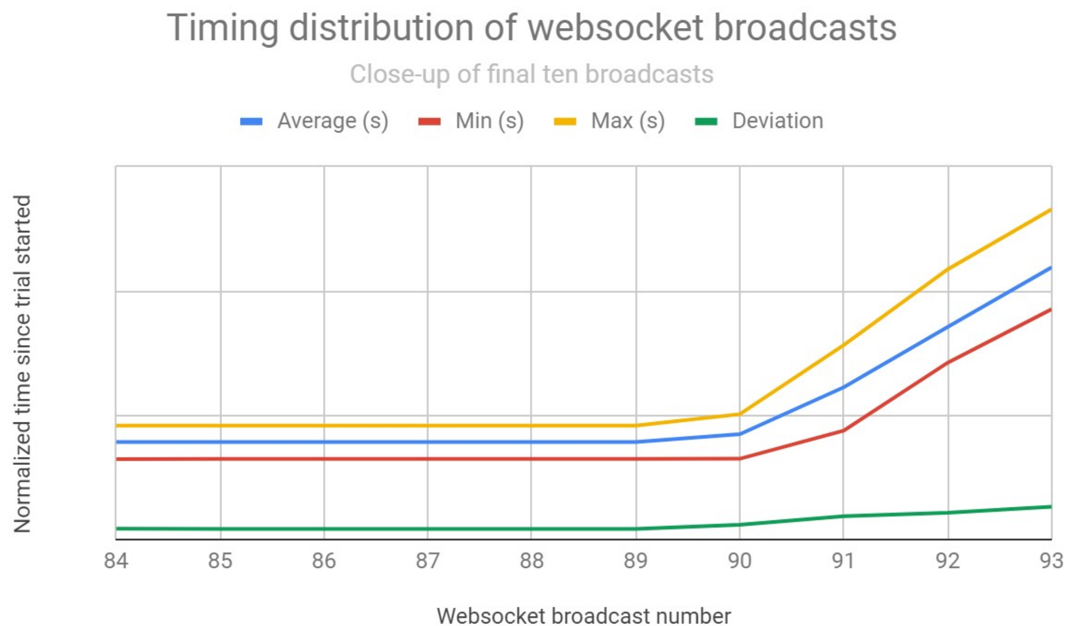

Figure 8: Isolated single search packet dispatch timings - closeup of packets 84-93

As shown in Figure 8, packets 84 through 89 have an extremely flat distribution on minimum, average, and maximum alike. This is consistent with a single social media service resolving its query and sending many packets near-simultaneously in all ten runs. Packets 90 through 93 show a consistent and substantial increase, where at no point does the *maximum* value of packet k exceed the *minimum* value of packet k+1. Additionally, the standard deviation of the test data increases much more slowly than the data themselves, indicating that this increase is predictable. This pattern supports that time to last post will remain a usable KPI for the purposes of this work, as opposed to shifting to a 90% threshold value that would not accurately reflect customer experience, even though the final few packets show significantly different behaviour from the rest of the samples. Further, as these final packets are likely caused by a single social media provider generating consistently-slow responses, omitting these final packets would be clearly noticeable to a customer.

4.2 Single burst input, increasing search count

These next trials show how Phoenix reacts to single bursts of consecutive searches, executed against a fixed number of queue worker processes. Searches are sent with a one-second delay between each completed network call, meaning the actual delay may be slightly longer than one second after factoring in web response times.

The first set of graphs shown below, Figures 9, 10, and 11, are timeline graphs. These show the start and end times of each search in the trial, measured as the time the search request was received by Phoenix and the time to first post for that search request, plotted against the absolute time that has elapsed during the trial. This provides a chronological summary of the trial, indicating how the user experience varied for each hypothetical user responsible for providing these search requests relative to when that user's search request was received.

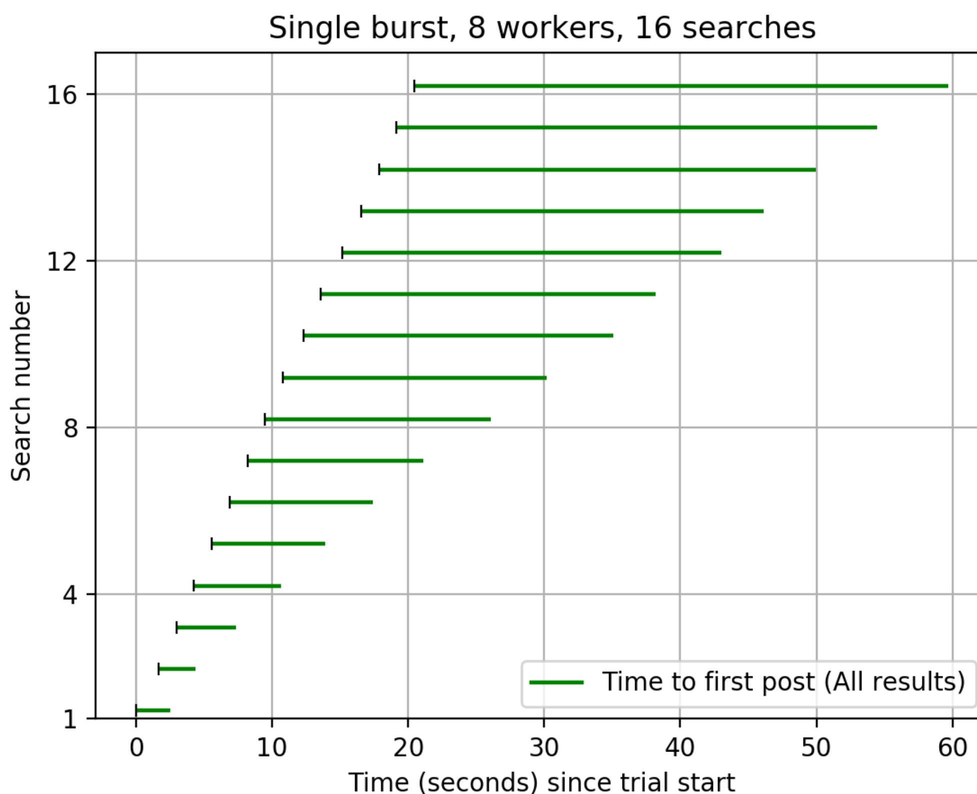

Figure 9: Single burst timeline, 8 workers, 16 searches

Figure 9 shows a simple series of successful search requests. At the bottom of the graph, the short line at (0, 1) shows the first search request starting at t=0, and ending at t=2.52, for an elapsed TTFP of 2.52s. Directly above it, the second, slightly longer line begins at t=1.64s and ends at t=4.37s, for an elapsed TTFP of 2.73s. Each subsequent search request has a slightly longer line, indicating a slow, gradual increase of TTFP as the trial progresses. The final search request begins at t=20.4s and ends at t=59.7s for an elapsed TTFP of 39.3s.

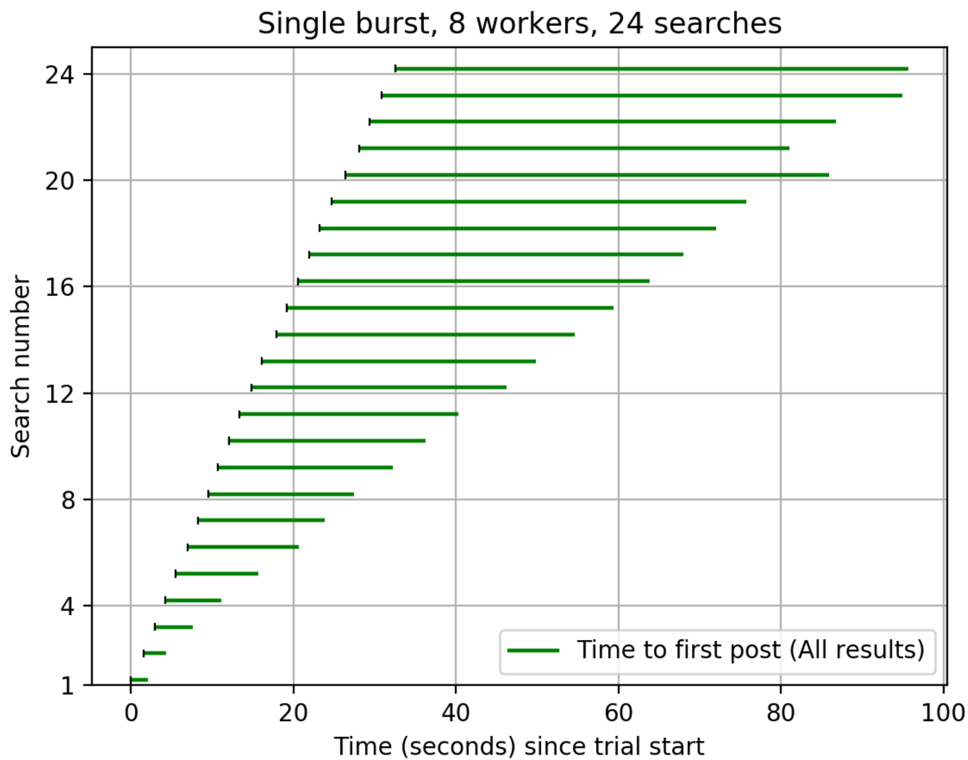

Figure 10: Single burst timeline, 8 workers, 24 searches

Figure 11: Single burst timeline, 8 workers, 32 searches

Figure 10: Single burst timeline, 8 workers, 24 searches shows similar results to Figure 9. Figure 11: Single burst timeline, 8 workers, 32 searches on the other hand, shows the first failed searches of these results. Searches 23 and 24 returned only partial results, each returning only 33 result packets instead of the expected 93. Searches 25 through 30 all failed to return any data at all, indicated by a small 'x' on the chart at the moment the search request was made.

Interestingly, after 6 consecutive searches dispatched no results, searches 31 and 32 from Figure 11 not only returned with data, but returned with a complete 93 result packets. This suggests that, while Phoenix entered a failure state, it was able to recover from this failure state and resume returning data. More information about this recovery will be discussed later.

Even without failed searches, however, the 16- and 24-search trials were not without useful insights. The following graphs, beginning with Figure 12: KPI comparison plot, 8 workers, 16 searches, show the time to first and last post for each search in each of these trials. Unlike the timeline plots shown previously, these comparison plots show each search *relative* to the time it was executed, instead of considering the *absolute* time to the start of the trial. Using relative time does not show the chronology of the request series, but does allow for an easier comparison of the time to first and last post as the trial progresses.

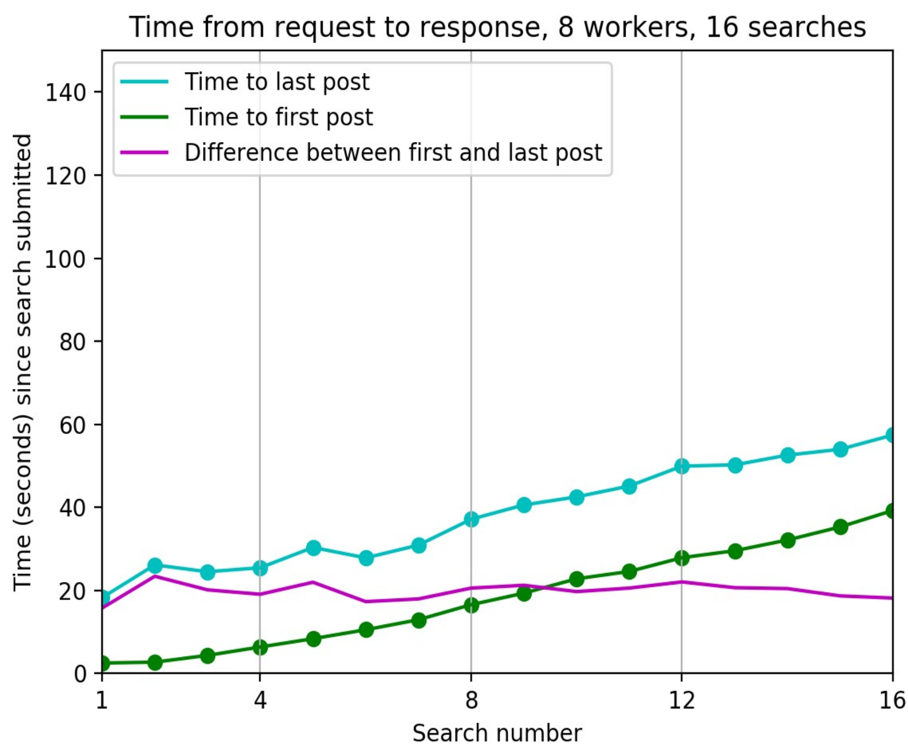

Figure 12: KPI comparison plot, 8 workers, 16 searches

As with the timeline graph, Figure 12 shows the first search with a TTFP of 2.52s. Additionally, the TTLP for this search request is now provided, at 18.3s. However, the start time of each search request is no longer considered on this relative graph, so the final search is shown with a TTFP of 39.3s and TTLP of 57.4s, ignoring the 20.4s offset present in absolute time. Additionally, the third line without markers indicates the *difference* between the TTFP and TTLP for each search request.

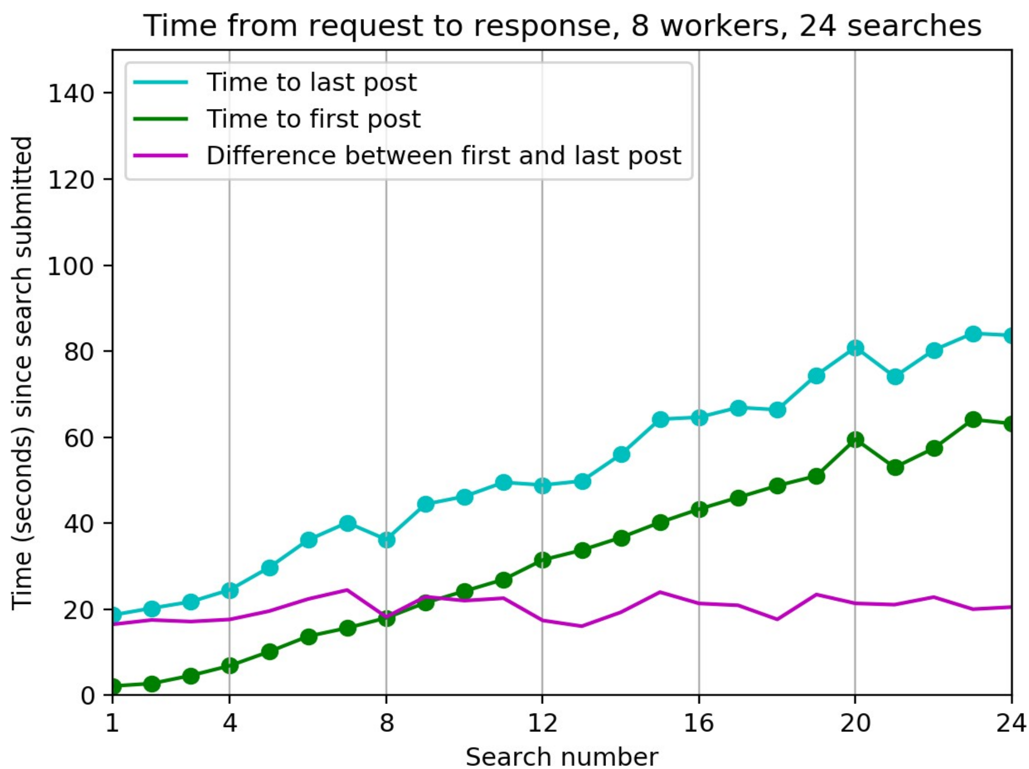

Figure 13: KPI comparison plot, 8 workers, 24 searches

Figure 14: KPI comparison plot, 8 workers, 32 searches

The comparison graphs for the 16-, 24-, and 32-worker plots provide a different set of insights into the behaviour of Phoenix. As shown previously, the TTFP and TTLP of the system scales up at a steady rate, with each subsequent search taking longer to return results. However, the timeline graphs make it difficult to observe that the TTFP is increasing even in relative time. In other words, subsequent search requests don't just start later in the trial, they also take longer to return results relative to when they're started. This was indicated on the timeline graphs by the length of each search's line, but is made far clearer in these comparison graphs.

By contrast to the magnitudes of the TTFP/TTLP, the difference between these two values remains largely consistent between runs. Table 1: TTFP/TTLP for 16-search trial shows the complete TTFP, TTLP, and the difference between them for the 16-search trial.

Table 1: TTFP/TTLP for 16-search trial

Search number	**Time to first post (seconds)**	**Time to last post (seconds)**	**Difference (seconds)**
1	2.51	18.3	15.8
2	2.74	26.2	23.4
3	4.37	24.5	20.1
4	6.39	25.5	19.1
5	8.38	30.4	22.0
6	10.5	27.8	17.3
7	13.0	30.9	18.0
8	16.6	37.1	20.6
9	19.3	40.6	21.2
10	22.8	42.5	19.7
11	24.6	45.1	20.6
12	27.9	50.0	22.1
13	29.6	50.2	20.7
14	32.1	52.6	20.4
15	35.3	54.0	18.7
16	39.3	57.4	18.2

Despite the TTFP increasing from 2.51s up to 39.3s and the TTLP increasing from 18.3s up to 57.4s, the difference between these two values remains between 22.1s to 15.8s with no discernable upwards trend.

The difference between the TTFP and TTLP for the 24- and 32-search trials can also be checked, from the third line on Figures 13 and 14. Outside of a brief deviation on Figure 14, both figures show a consistent difference to the 16-search trial. The deviation near the end of the 32-search trial is caused by the two searches that returned only partial results, which resulted in a smaller difference between the TTFP and TTLP because many of the 'last posts' were not dispatched successfully, and so the TTLP is being computed from a packet earlier in the dispatch pattern, which as shown in Figure 8 can result in a 10 second or longer reduction in the TTLP with the absence of only the final 3-4 packets.

Figure 14 returns to an approximate 20s difference for the final searches after the timeouts, showing that the deviation is caused by the partial and failed searches and returns to normal once searches with complete responses resume. As the difference between the TTFP and TTLP is largely constant, it is not shown on timeline graphs during this work.

As the 32-search trial failed only on the final few searches, two more larger-scale tests are shown below to further demonstrate system behaviour at high loads.

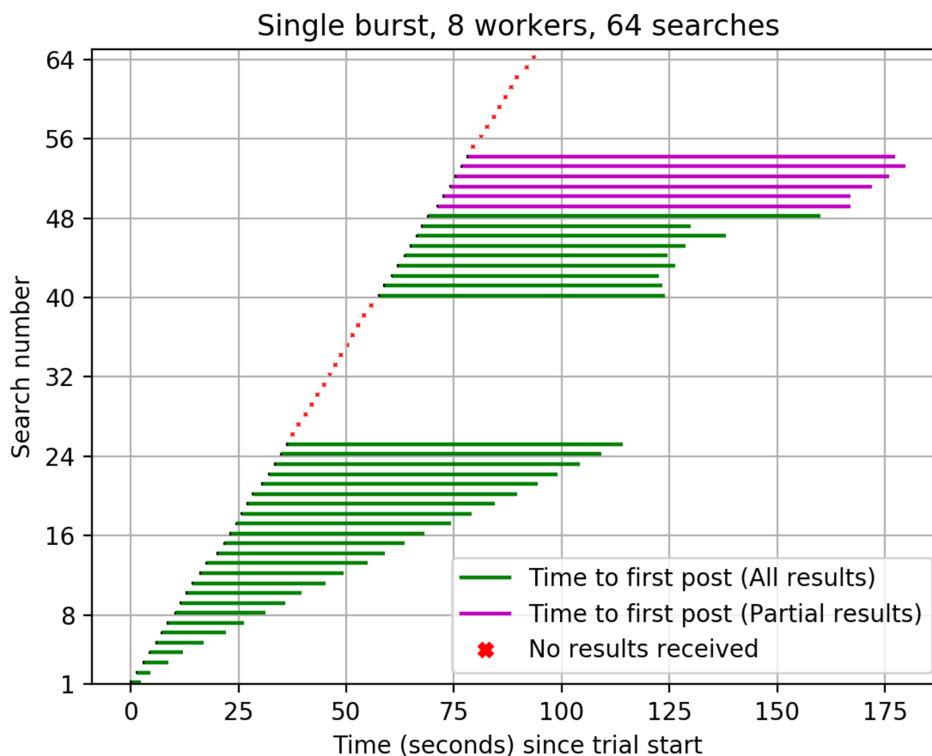

Figure 15: Single burst timeline, 8 workers, 64 searches

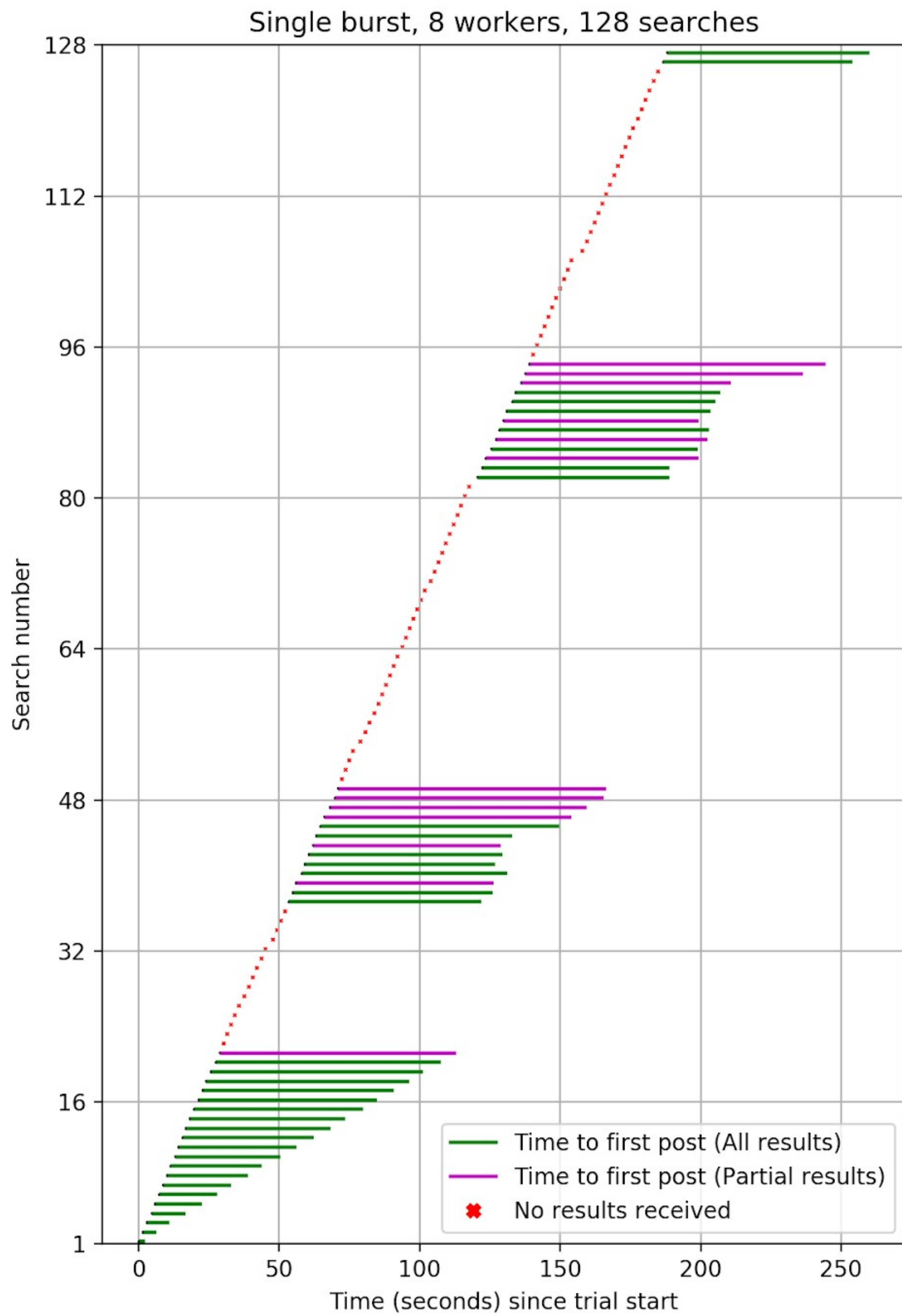

Figure 16: Single burst timeline, 8 workers, 128 searches

The clear trend in Figure 15: Single burst timeline, 8 workers, 64 searches and Figure 16: Single burst timeline, 8 workers, 128 searches is that, once the Phoenix workers reach capacity, multiple consecutive searches will time out entirely until the system recovers, then a burst of searches will execute again, with the cycle repeating potentially multiple times. Concerningly, from the websocket dispatch counts shown in Figure 16, the system appears to never fully recover. After the first set of timeouts, the first few searches to run will successfully resolve the expected 93 websocket packets, but most searches after those first few will return only partial result sets.

Notably, the longest KPIs shown in these charts have last posts, and even first posts, far in excess of the 60-second timeout imposed in the `ExecuteSocialQueryAbstract` implementations. On the 32-search trial, the longest time to first post was 70.27 seconds, and the longest time to last post was 87.29 seconds, while the 64-search trial had results with both KPIs in excess of 100 seconds. The apparent explanation for this stems from the decision to place both `ExecuteSocialQueryAbstract` and its predecessor, `QueryProviders`, on the same default queue. The 60-second timeout does not begin counting down until the `ExecuteSocialQueryAbstract` listener is first created, which occurs after the `QueryProviders` instance is executed. As both of these listeners run on the same queue in FIFO order, a `QueryProviders` instance may sit on the queue for an unbounded length of time, and only once it is executed successfully does it generate `ExecuteSocialQueryAbstract` instances that begin tracking their timeouts. This queueing limitation reduces the ability of the Phoenix team to control their quality-of-service metrics by preventing their assigned timeout values from actually timing out jobs when expected.

4.3 Single burst input, increasing worker count

The previous trials have used an artificially-low worker count to induce performance issues more easily. The operations of Phoenix under tests are not particularly resource-intensive; the limiting factor in social media queries is the time spent waiting for the third-party API to respond to the request, which leaves the Phoenix worker running the query effectively idle for the duration. As a result, increasing the worker count available to Phoenix should provide a considerable improvement to the throughput of the system and, it would be hoped, the corresponding customer-facing KPIs.

The following trials demonstrate the impact of increasing worker counts on a static throughput. All trials use the same workload of 128 searches sent at a consistent rate, with all other values held constant save for the worker count.

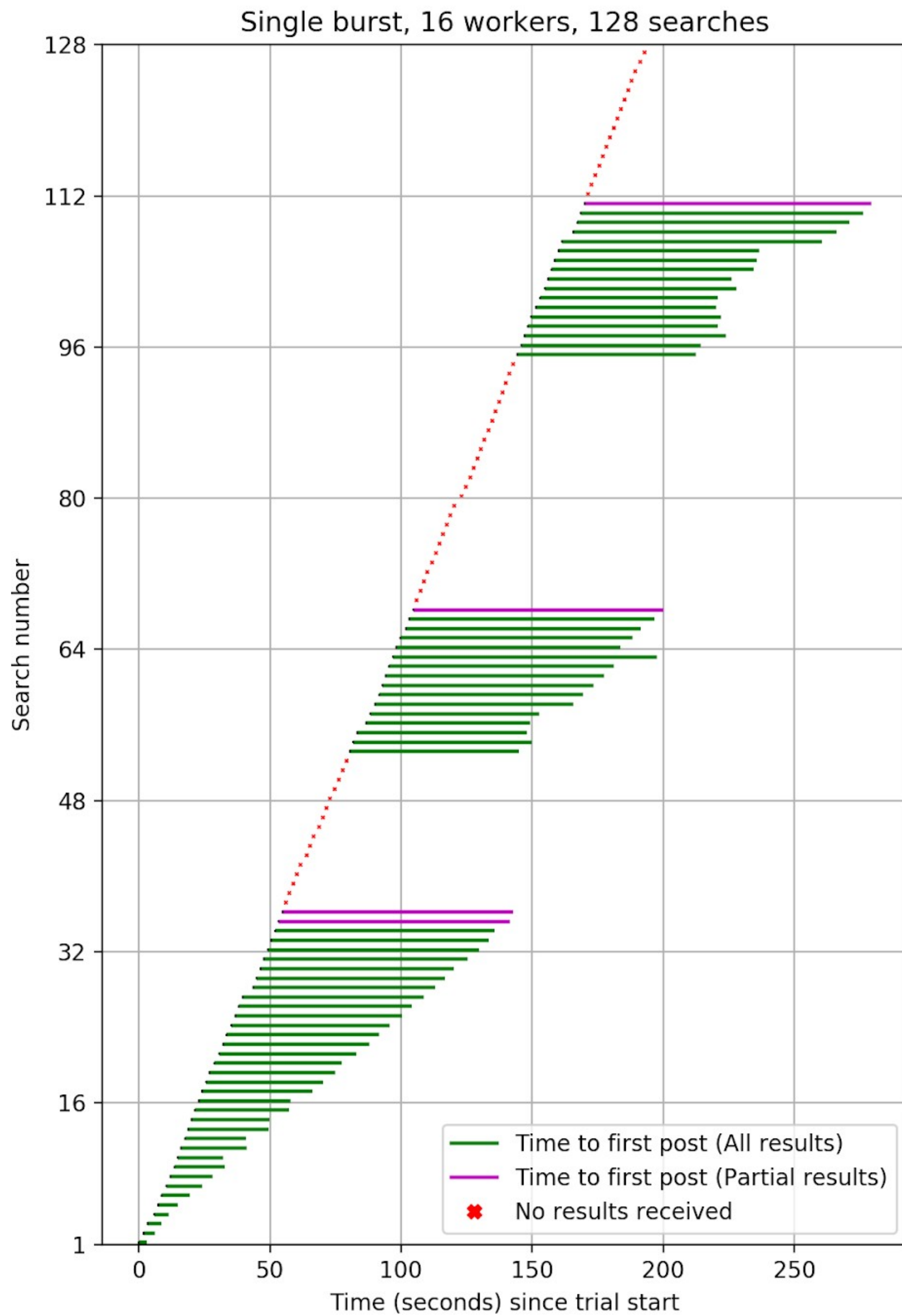

Figure 17: Single burst timeline, 16 workers, 128 searches

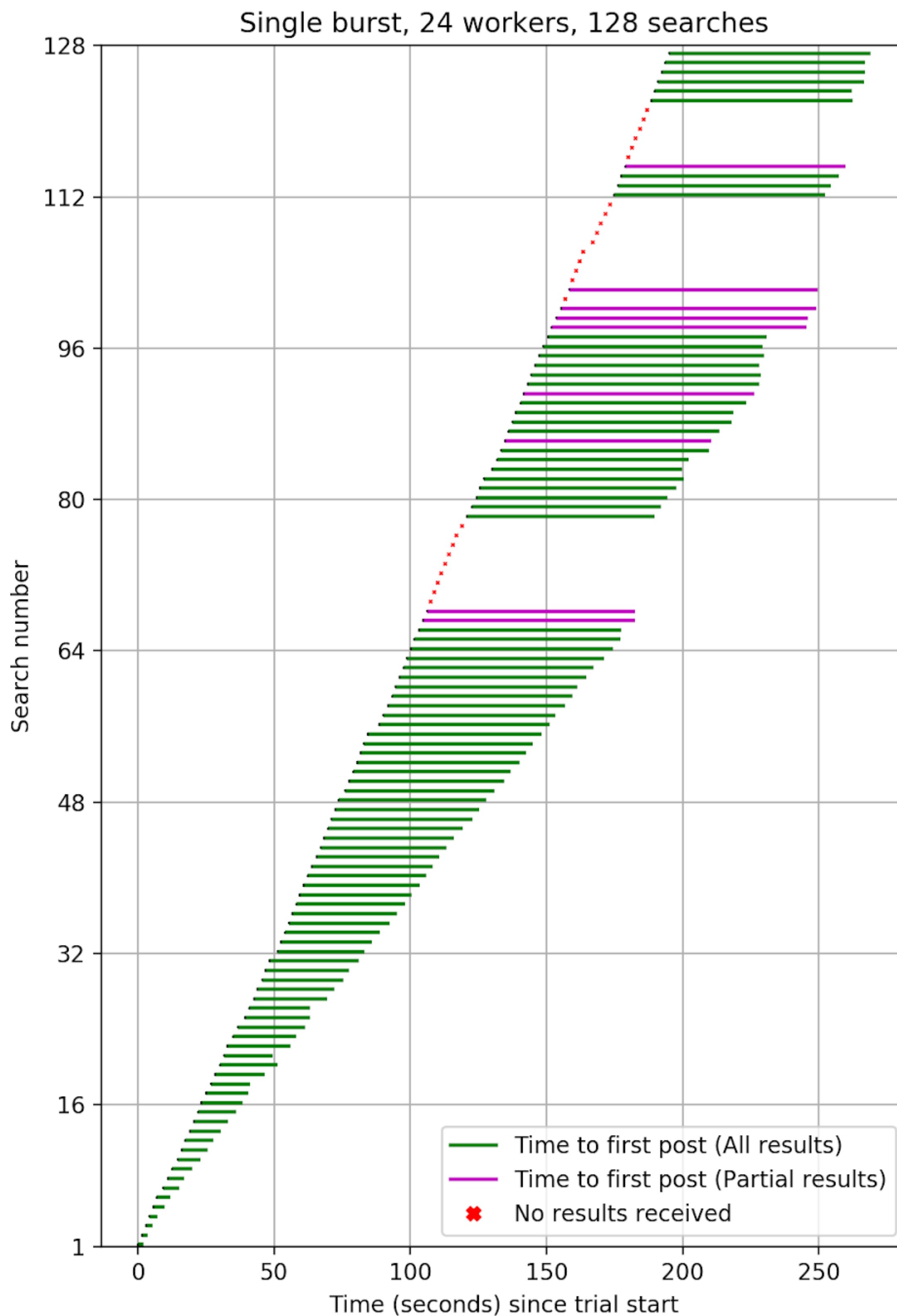

Figure 18: Single burst timeline, 24 workers, 128 searches

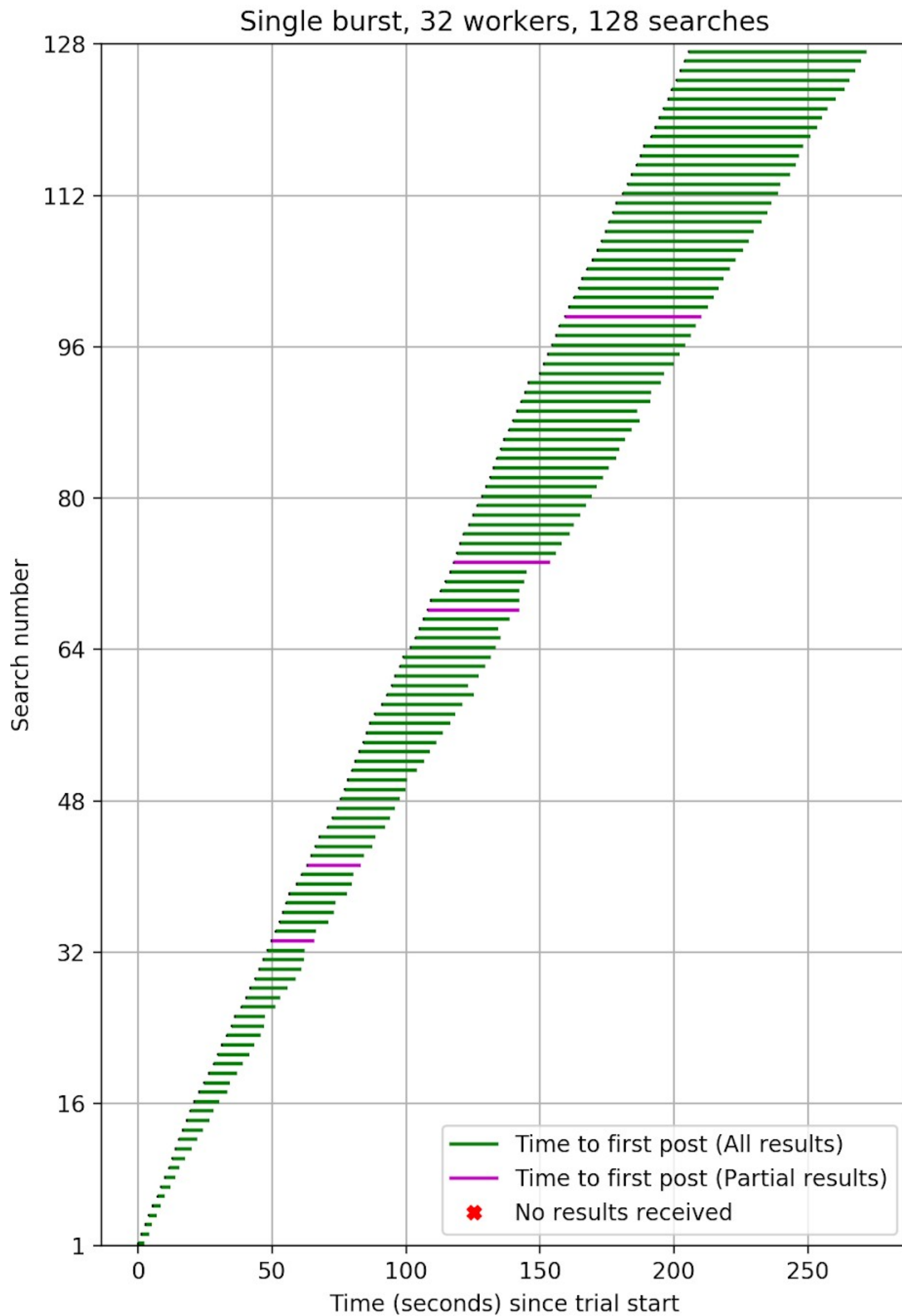

Figure 19: Single burst timeline, 32 workers, 128 searches

Increasing the worker count to 16 in Figure 17: Single burst timeline, 16 workers, 128 searches did not noticeably change the number of searches that received no data. However, it did reduce the number of searches that received only partial data, with only 4 searches receiving a websocket packet count not equal to 0 or 93. Increasing the worker count to 24 in Figure 18: Single burst timeline, 24 workers, 128 searches significantly reduced the number of failed searches, and increasing the worker count to 32 inFigure 19: Single burst timeline, 32 workers, 128 searches completely alleviated timeouts for the full 128-search process.

While increasing the worker count up to 32 almost completely eliminated failed and incomplete searches from the trial, as shown in Figure 19, the effect of increasing the worker count was less dramatic on the TTFP / TTLP. Figures 20 through 23 are comparison graphs showing the elapsed time to first post and time to last post for the 8-, 16-, 24-, and 32-worker tests against the 128-search workload.

Figure 20: KPI comparison plot, 8 workers, 128 searches

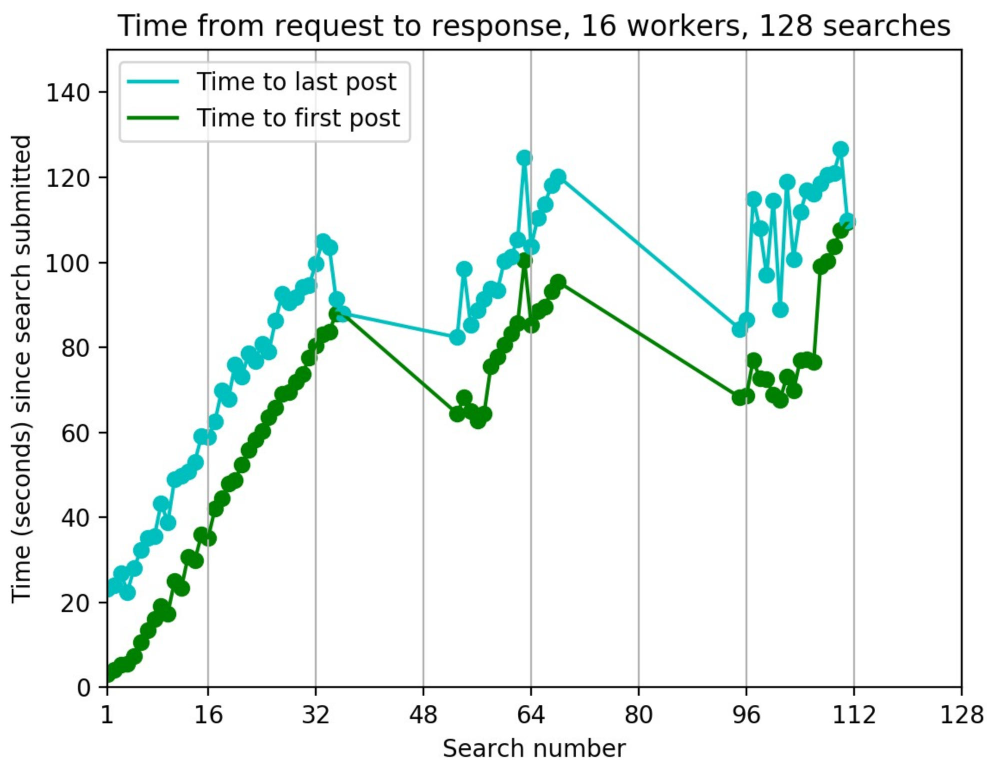

Figure 21: KPI comparison plot, 16 workers, 128 searches

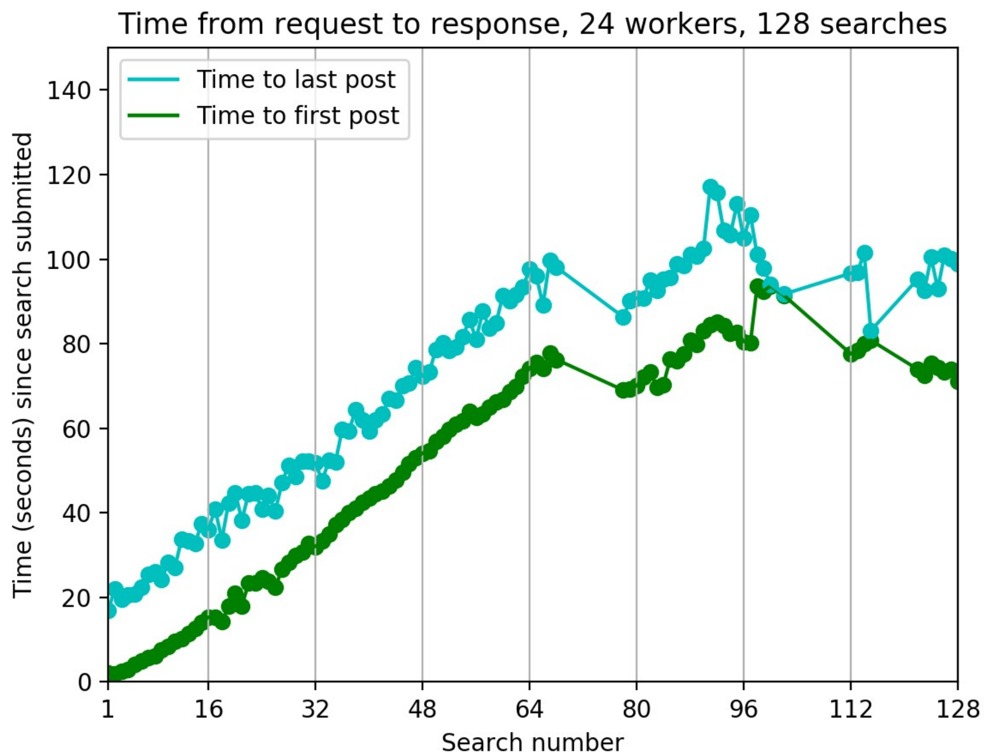

Figure 22: KPI comparision plot, 24 workers, 128 searches

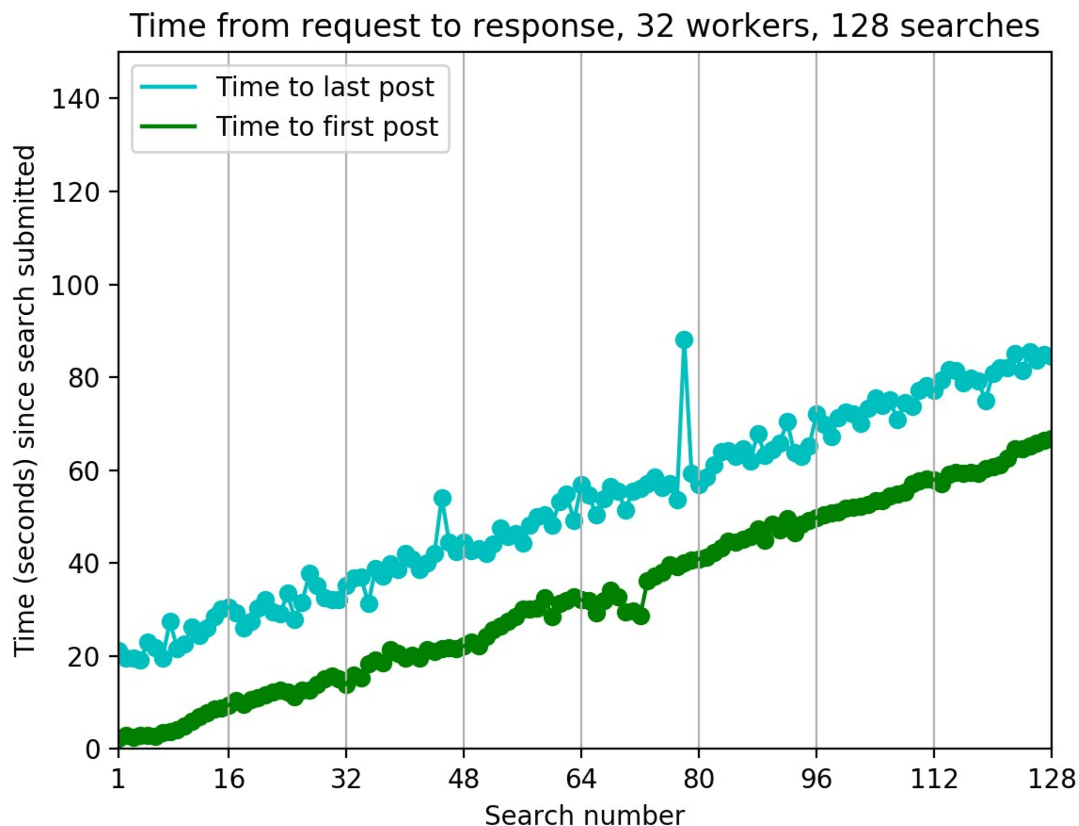

Figure 23: KPI comparison plot, 32 workers, 128 searches

The 8-, 16-, and 24-worker trials all experienced searches timing out after a number of search requests were run. Notably, the timeouts occurred in ‘bursts’, where a number of searches would time out in succession, followed by a number of searches successfully returning results. These periods where multiple consecutive searches time out in a row will be referred to as “timeout bursts” in the tables and discussion below.

Table 2: Locations and timings of timeout bursts during 128-search trials lists identifying information for each timeout burst in the 8, 16, 24, and 32-worker trials. Multiple entries are present for searches that experienced multiple sets of timeouts, with one row for each of these timeout bursts. The 32-worker trial did not demonstrate a timeout burst, and instead lists data from its first search through its last search.

Table 2: Locations and timings of timeout bursts during 128-search trials

Worker count	Timeout batch	Last search to return results before first time out	TTFP of last search to return results before time out (seconds)	Count of searches that timed out during this batch	TTFP of first search after batch ends (seconds)
8	1	21	84.1	15	68.5
8	2	49	95.2	21	68.3
8	3	91	73.1	30	67.5
16	1	34	83.7	15	64.4
16	2	67	93.3	25	68.3
16	3	110	107.5	N/A, trial ended	N/A, trial ended
24	1	68	76.3	9	69.0
24	2	100	93.7	10	77.5
24	3	115	80.8	6	73.9
32	1	128 (all searches returned results)	66.8	N/A, no time outs	N/A, no time outs

The TTFP of the last search before each burst and of the first search after each burst are both interesting, in that they appear to have no correlation to the number of workers in each trial, nor to the number of previous timeout bursts during that trial. This lack of correlation suggests that the TTFP before and after each burst is a constant dependent on some other variable in Phoenix that was not changed between each trial.

Notably, up until the first timeout burst of each trial, the TTFP appears to be growing at a linear rate. Table 3: Time to first post rate of increase for 128-search trials, workers 8-32 shows the rate of increase from the first search of the trial until the final search to return complete results, which is either the beginning of the first timeout burst or the end of the trial.

Table 3: Time to first post rate of increase for 128-search trials, workers 8-32

Worker count	**TTFP rate of increase (seconds per search)**
8	3.88
16	2.37
24	1.09
32	0.51

From these tables, part of the behaviour of Phoenix under load begins to emerge. When subjected to a higher volume of input searches than Phoenix can process at a consistent rate, the TTFP begins to increase at a linear rate dependent on the number of workers available and the rate at which new input is provided. When the TTFP reaches a threshold, at approximately 80 seconds, this rate of growth breaks down and multiple consecutive searches will time out, with searches occurring just before this threshold returning but with only partial results. After a number of searches time out, Phoenix resumes processing results, starting at a lower TTFP of approximately 68 seconds but with a similar rate of increase as before the timeout burst, and soon after the TTFP threshold is reached again and times out another group of searches. This cycle of growing, timing out, and resuming repeats cyclically until the trial ends.

To gather more information on the rate of growth of the TTFP, three additional trials were run with higher worker counts. The 40-worker and 48-worker trials follow the pattern of the previous trials, increasing the worker count by 8 each time, while the 64-worker trial doubles the workers of the previous highest-throughput trial to show the effect of massive increases to resource counts.

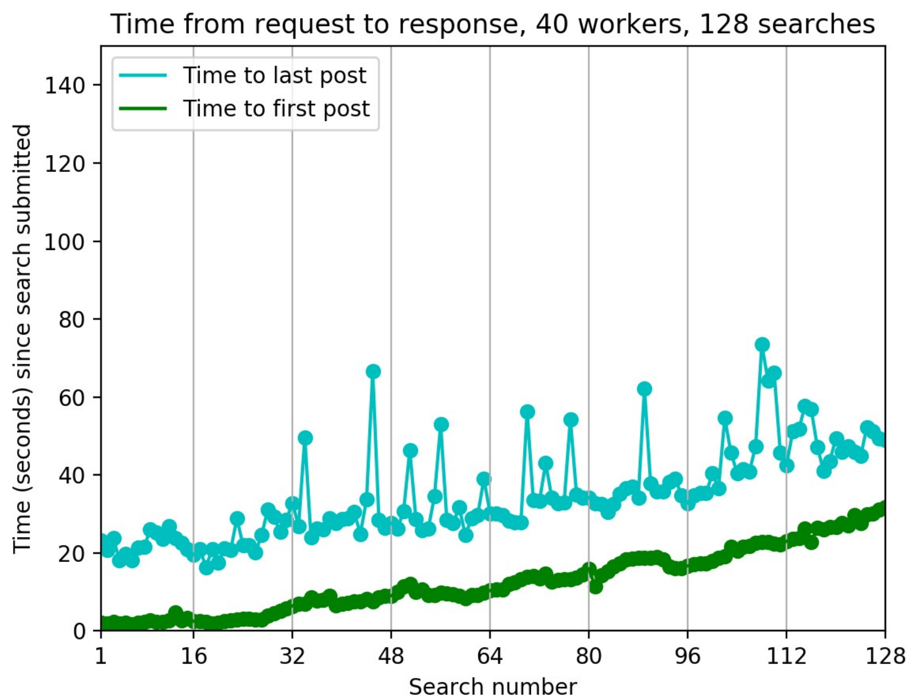

Figure 24: KPI comparison plot, 40 workers, 128 searches

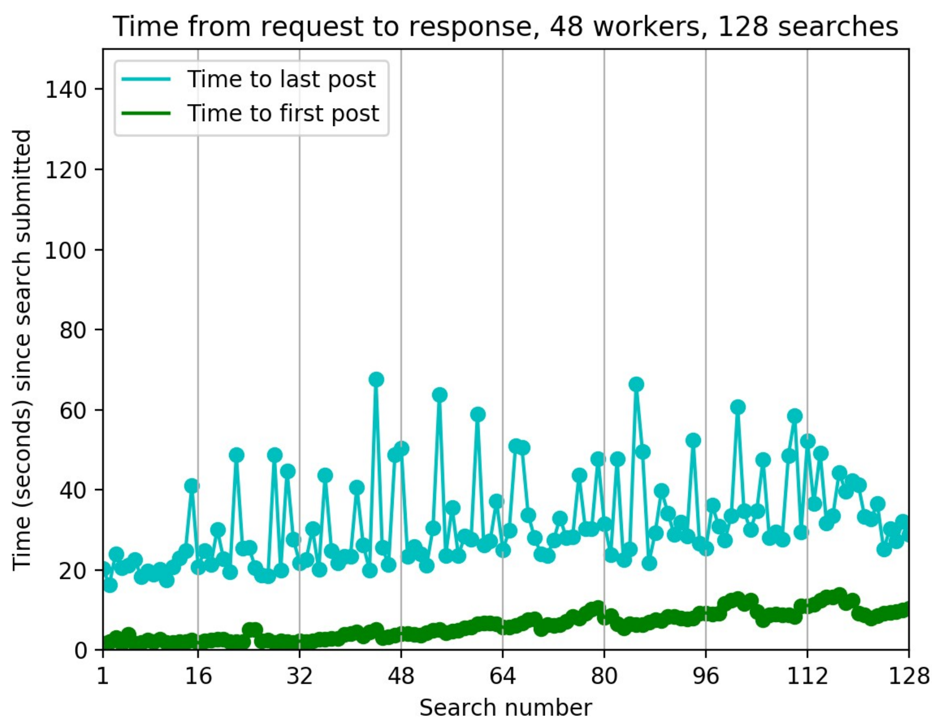

Figure 25: KPI comparison plot, 48 workers, 128 searches

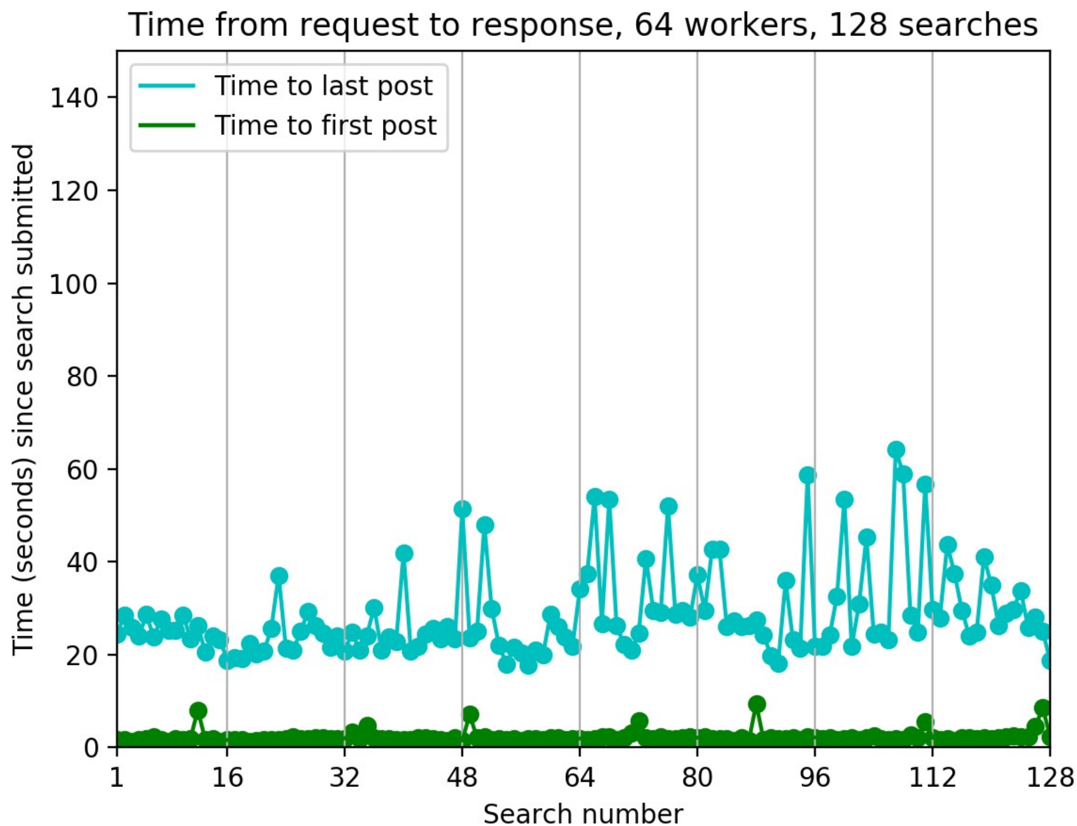

Figure 26: KPI comparison plot, 64 workers, 128 searches

Figure 24: KPI comparison plot, 40 workers, 128 searches and Figure 25: KPI comparison plot, 48 workers, 128 searches continue to show an upward slope in the TTFP of consecutive searches, and as expected that slope lessens as the worker count rises. Figure 26: KPI comparison plot, 64 workers, 128 searches shows a nearly flat slope, suggesting a largely consistent behaviour and a lack of KPI growth, though the time to last post becomes increasingly noisy over the course of the trial and suggests that even 64 workers are insufficient to completely eliminate the effects of scale. These results are summarized in Table 4: Time to first post rate of increase for 128-search trials, workers 40-64.

Table 4: Time to first post rate of increase for 128-search trials, workers 40-64

Worker count	**TTFP rate of increase (seconds per search)**
40	0.23
48	0.069
64	0.005

The final trial performed in this section repeats the 32-worker trial in Figure 19, but with the total number of searches doubled to 256. This trial aims to demonstrate that, while Figure 19: Single burst timeline, 32 workers, 128 searches did not demonstrate any search timeouts, this trial run was on a course to time out search requests in the near future. This prediction is based on the TTFP increase rate of 0.51, the final TTFP of 66.8s, and the knowledge that search requests first begin to time out when the application's TTFP surpasses approximately 75-90s, suggesting that the first search timeout will occur approximately between search 150 to 180.

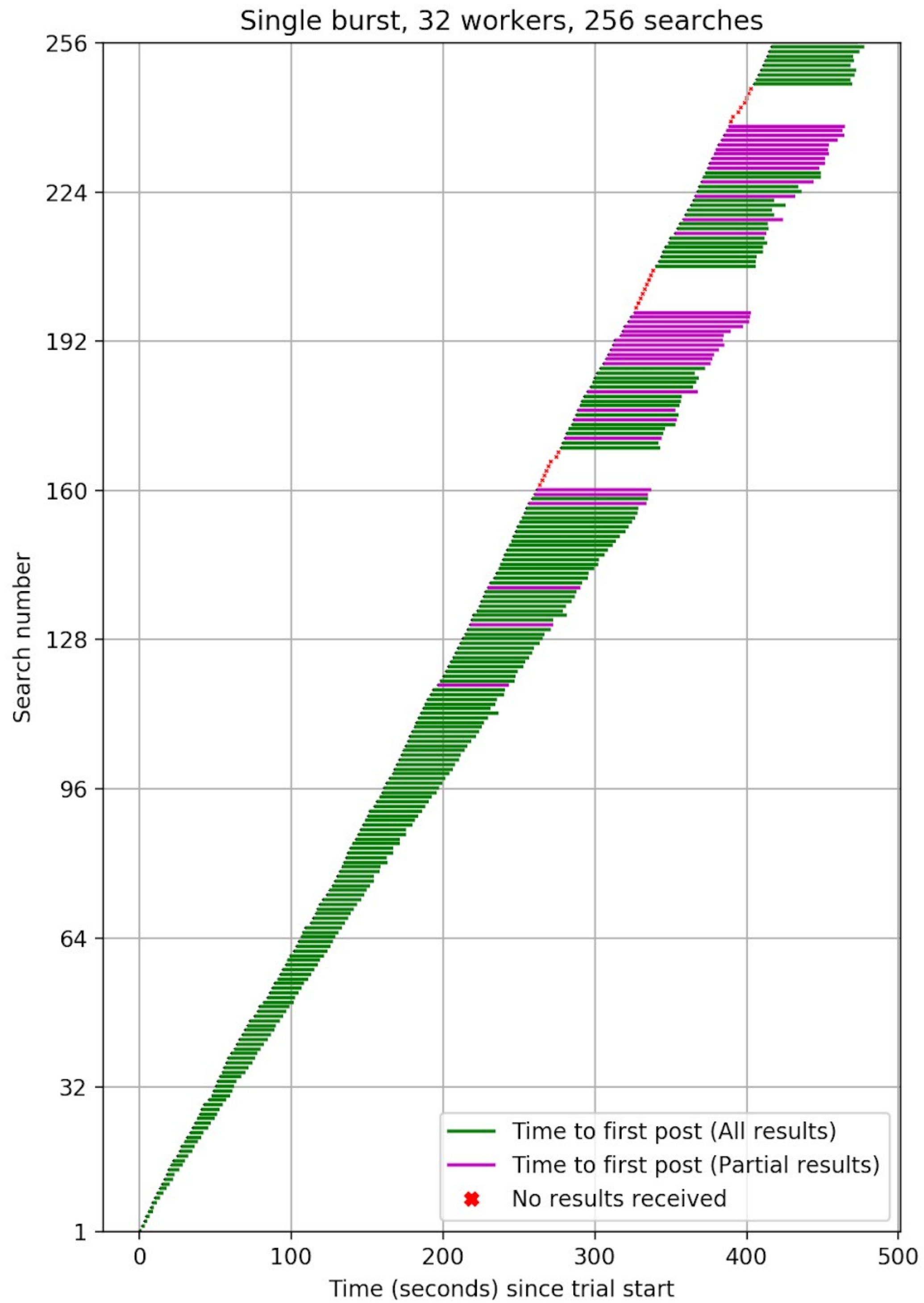

Figure 27: Single burst timeline, 32 workers, 256 searches

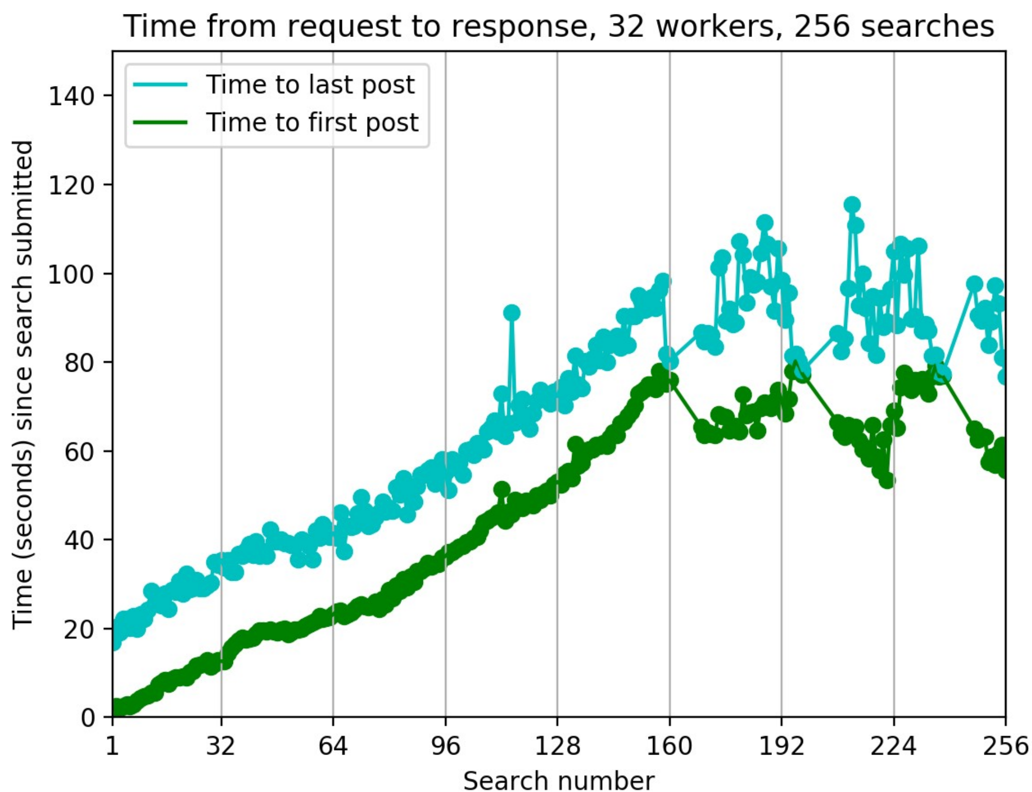

Figure 28: KPI comparison plot, 32 workers, 256 searches

As predicted, Figure 19: Single burst timeline, 32 workers, 128 searches was not exempt from timeouts, and simply hadn't run enough searches to trigger one. In Figure 23: KPI comparison plot, 32 workers, 128 searches, the final search before the first search timeout burst was search 160, with a TTFP of 75.9s, a timeout burst length of 8 searches, and a TTFP of 65.5s on the first successful search post-burst. These values align with the prediction made above for a first search timeout between search number 150 and 180 with a TTFP of 75-90s. The TTFP of 65.5s once the first timeout burst concludes is also consistent with the values listed in Table 2: Locations and timings of timeout bursts during 128-search trials, and the plot outlined in Figure 28: KPI comparison plot, 32 workers, 256 searches is consistent with all previous KPI comparison plots from Figures 24 through 26. The success of these predictions provides solid evidence that the behaviour of Phoenix can be suitably quantified through the use of these derived metrics.

These trials demonstrate that, with considerable resources, Phoenix can be brought into a stable state where timeouts will not occur. However, doing so requires a considerable investment of resources, with 64 concurrent workers being required to process searches at a rate of one per second. Moreover, a comparable experience can still be had at lower worker counts - at a worker count of 32, Phoenix can process one search per second for over two and a half minutes without suffering a timeout, though the TTFP does grow

significantly over that time period. Doubling the worker count may prevent timeouts from occurring, but also doubles the cost of running Phoenix over that period of time.

The 256-search trial also revealed an interesting behind-the-scenes fact about Phoenix: its job executors do not handle timeouts well. By the end of the test, every single worker process on Phoenix had exited and restarted at least once. More concerningly, 7 out of the 32 workers had failed so many times in quick succession that Supervisord had placed them in the `FATAL` state, indicating that no further attempts would be made to restart them. Once flagged as FATAL, worker nodes will not be restarted without manual intervention or a full reboot of the server instance, which itself generally requires manual intervention or a catastrophic failure on Phoenix. These nodes represent a nearly-permanent loss in processing power available to the node, and should constitute a significant concern for the Phoenix development team as they iterate on their monitoring and automated recovery processes. The testing procedure employed during this work resolves these lost workers after each trial, to ensure no contamination occurs between independent trials on the system.

4.4 On-off inputs

The single-burst trials performed thus far have provided a solid understanding of the behaviour of Phoenix under load. However, single bursts are not necessarily indicative of real-world usage of the platform. A more robust solution is the on-off workload strategy, which simulates repeated bursts of interaction with Phoenix with periods of downtime. These periods of downtime are of great interest, as they will allow the Phoenix job queues to catch up on queued data without new ingestion.

The first trial, in Figure 29: On/Off timeline, 24 workers, 20x10 searches, 30 second delay, documents an On/Off workload pattern of 10 search bursts of 20 searches each, with a 30-second delay between each burst. During this trial, Phoenix had access to 24 worker nodes.

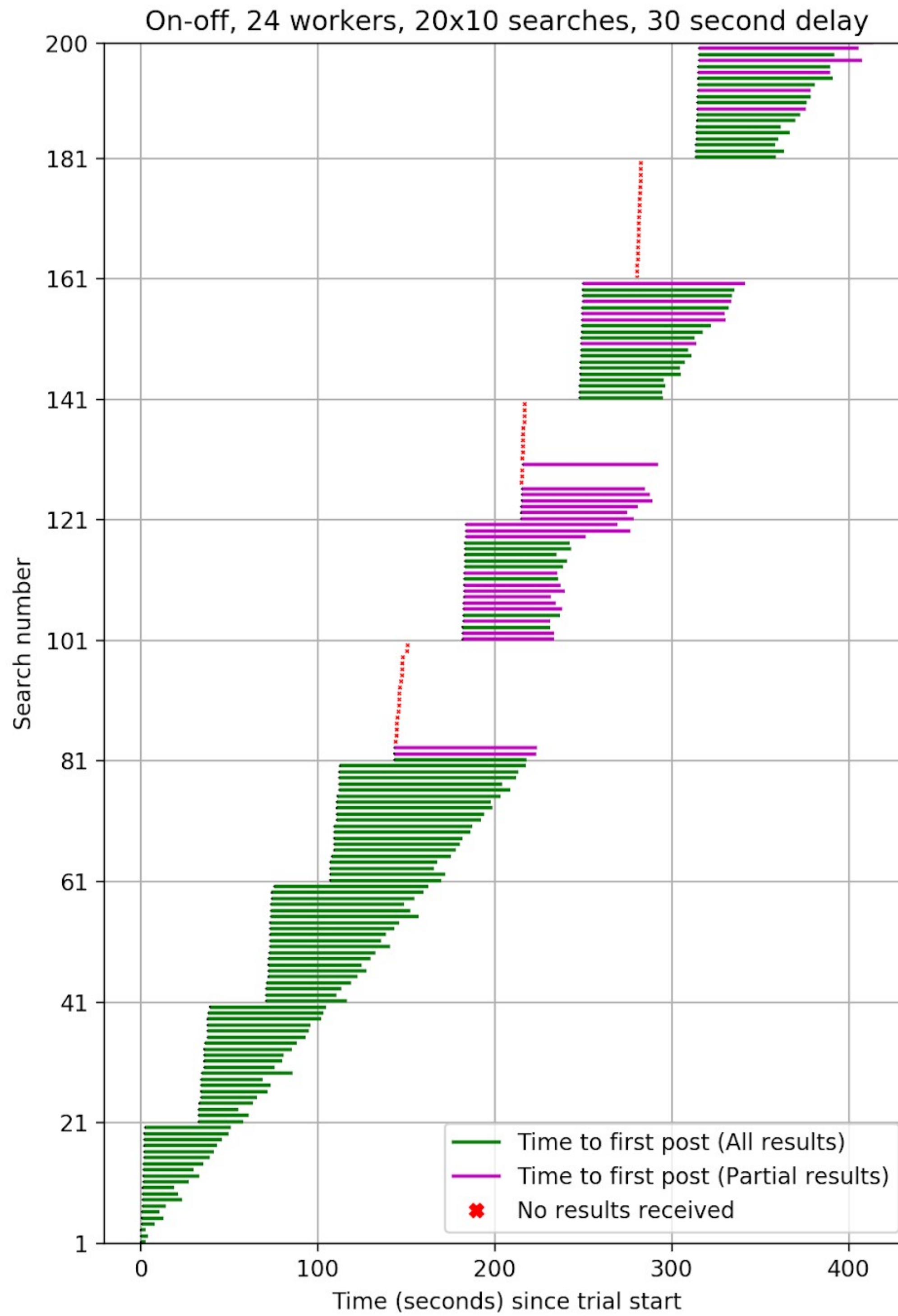

Figure 29: On/Off timeline, 24 workers, 20x10 searches, 30 second delay

The timeline graph in Figure 29: On/Off timeline, 24 workers, 20x10 searches, 30 second delay clearly demonstrates the 10 separate bursts of searches generated by the On/Off workload generation. The apparent consequence of these On/Off bursts in Figure 29 is that the searches in a burst tend to succeed or fail at generating results as a group, rather than showing independent failures. Bursts 5, 7, and 9 all suffered numerous timeouts, with 17/20, 13/20, and 20/20 search requests timing out respectively in each of these bursts, and only a single search across all three bursts successfully returning all expected websocket packets. The other seven bursts in Figure 29 suffered zero timeouts and only 7 partial results across 140 searches. In previous timeline graphs, such as Figure 23: KPI comparison plot, 32 workers, 128 searches, search timeouts were still clearly grouped together, but the use of an On/Off workload pattern has changed the variable-size timeout bursts from Table 2: Locations and timings of timeout bursts during 128-search trials into more rigid groups that align with the On/Off dispatch pattern. Figure 30: KPI comparison plot, 24 workers, 20x10 On/Off searches, 30 second delay below shows the KPI comparison graph for the 30-second trial, again using time relative to each individual search request rather than the time since the start of the trial to better show the escalation of TTFP/TTLP over the course of the test.

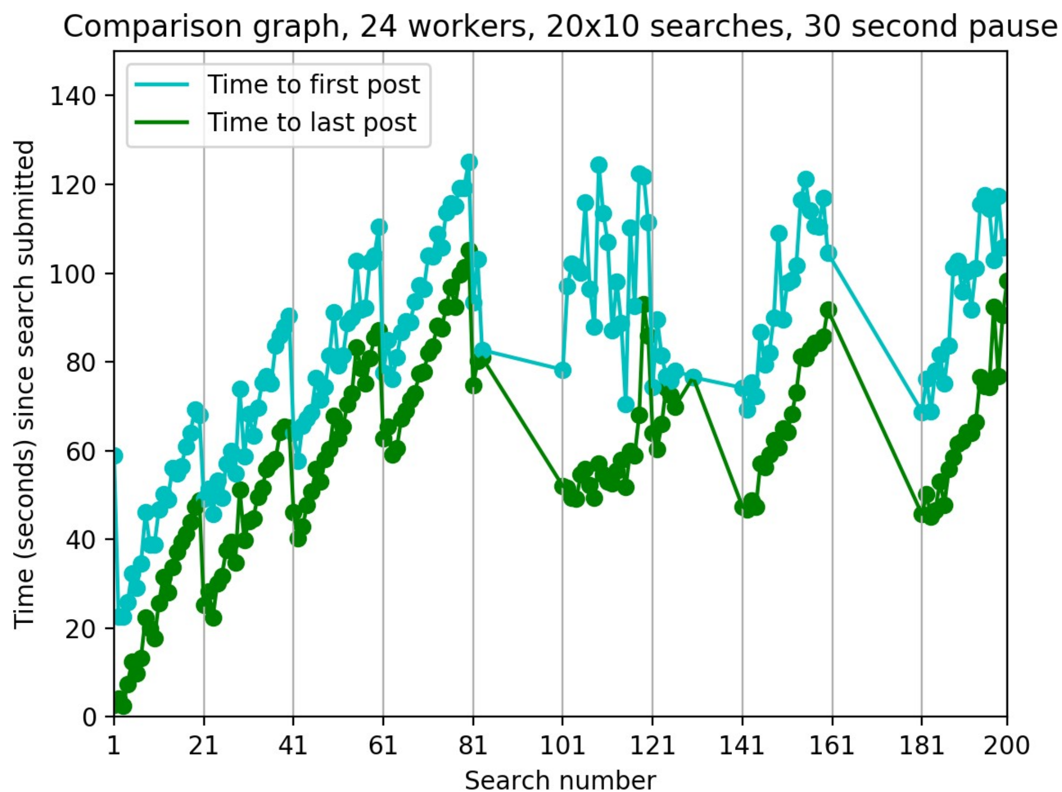

Figure 30: KPI comparison plot, 24 workers, 20x10 On/Off searches, 30 second delay

The KPI comparison graph for the 30-second delay trial, Figure 30: KPI comparison plot, 24 workers, 20x10 On/Off searches, 30 second delay above, resembles the comparison graphs from the single-burst trials such as Figure 28: KPI comparison plot, 32 workers, 256 searches, but with a clear sawtooth effect breaking up the previously-

smooth linear growth discussed in Table 3: Time to first post rate of increase for 128-search trials, workers 8-32. Each ‘tooth’ of the sawtooth effect lines up with an On/Off burst, suggesting that each period of downtime from the workload generator provides Phoenix with time to process through its queued jobs, and so once new jobs begin to arrive in the next On/Off burst the queue is shorter and the first few searches are processed more quickly. Notably, bursts 6, 8, and 10 in Figure 30 have similar maximum TTFPs and TTLPs, with an initial time to first post of approximately 50 seconds, rising to approximately 100 seconds by the final search in the burst, despite being farther chronologically into the trial. This consistency suggests that the failed bursts are providing time for the system to clear its queues and return to a more consistent operating mode before carrying on with further successful searches.

The next two trials were run with the same environment, but with a 45- and 60-second delay between bursts, respectively. During the 30-second trial, in the first On/Off burst, the final post of the final search was received at 48 seconds, and so the 45-second delay is intended to provide almost enough time for Phoenix to clear its job queue before the next burst of requests. The 60-second delay should provide an abundance of time and more than fully allow Phoenix to complete all pending jobs before a new batch is delivered.

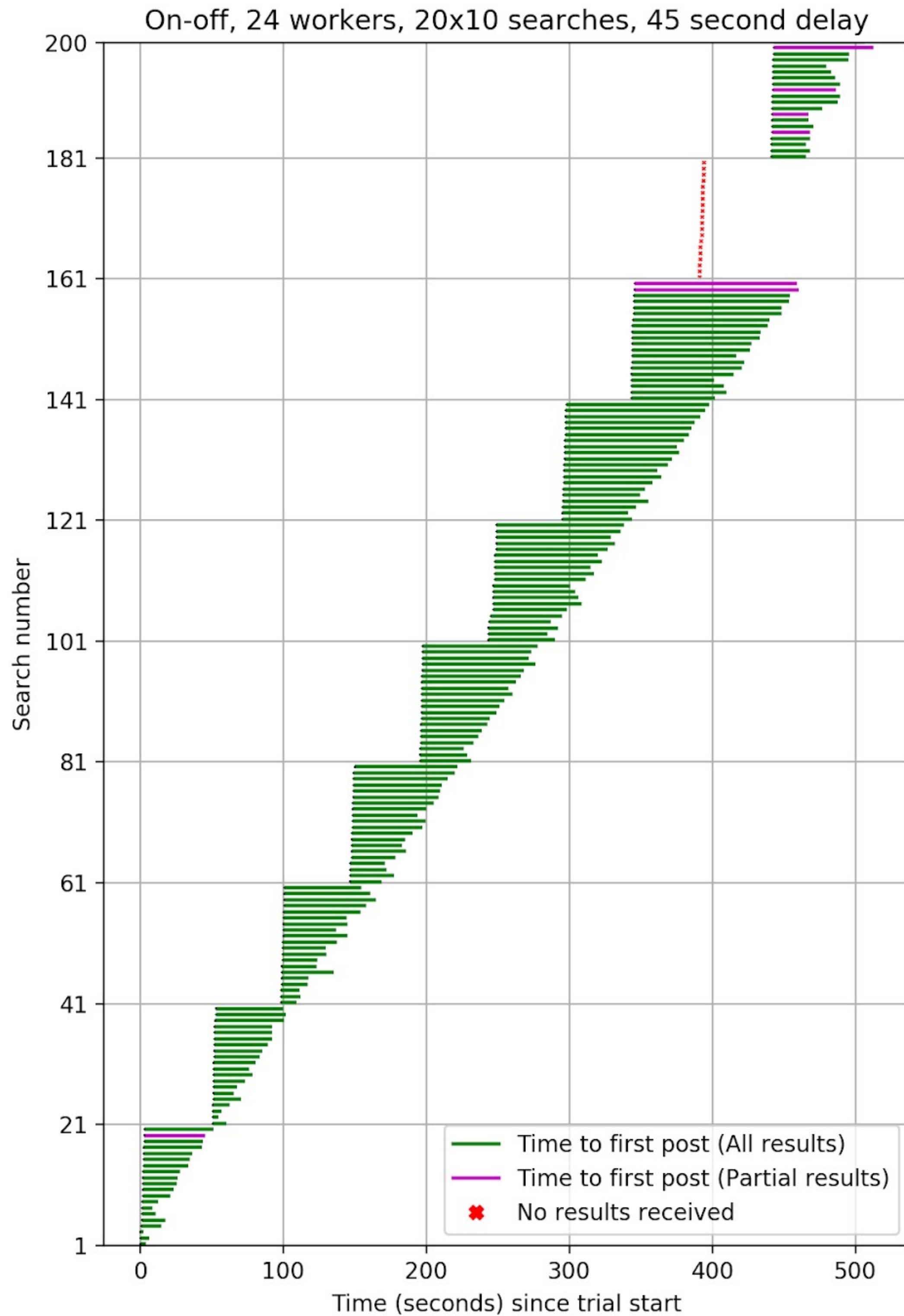

Figure 31: On/Off timeline, 24 workers, 20x10 searches, 45 second delay

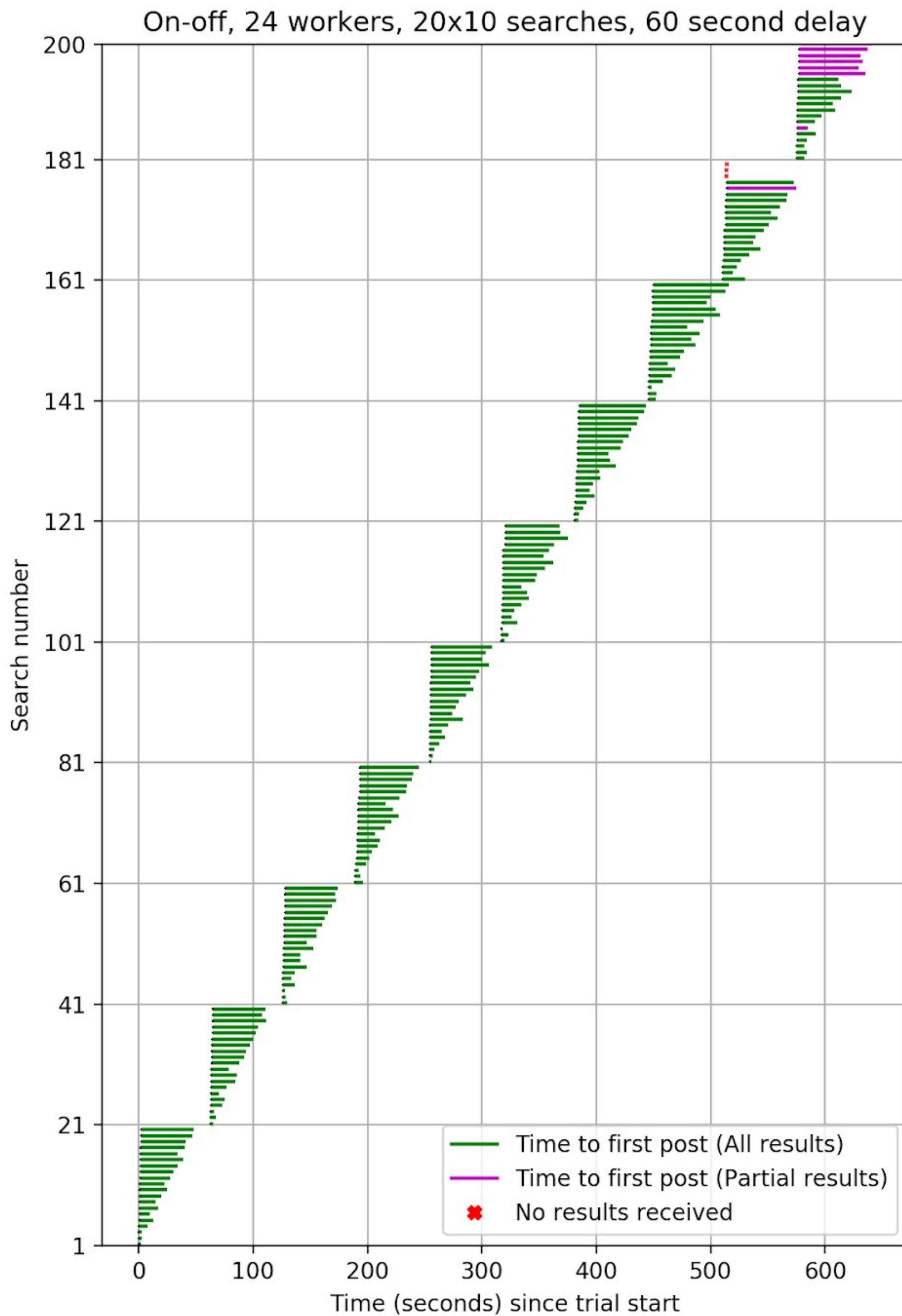

Figure 32: On/Off timeline, 24 workers, 20x10 searches, 60 second delay

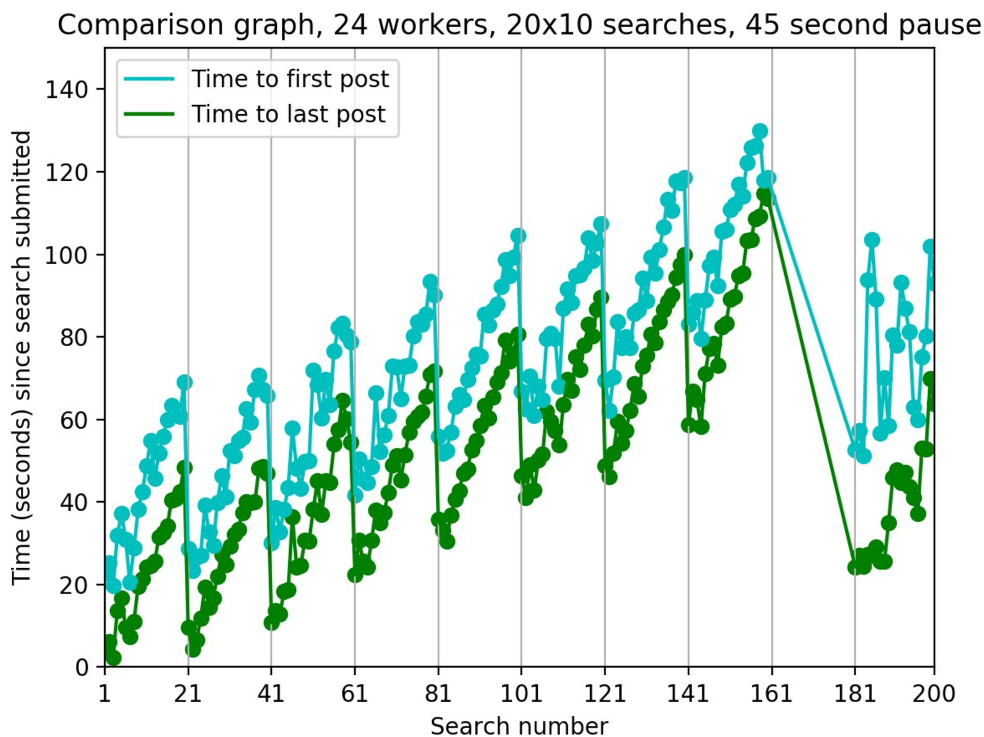

Figure 33: KPI comparison plot, 24 workers, 20x10 On/Off searches, 45 second delay

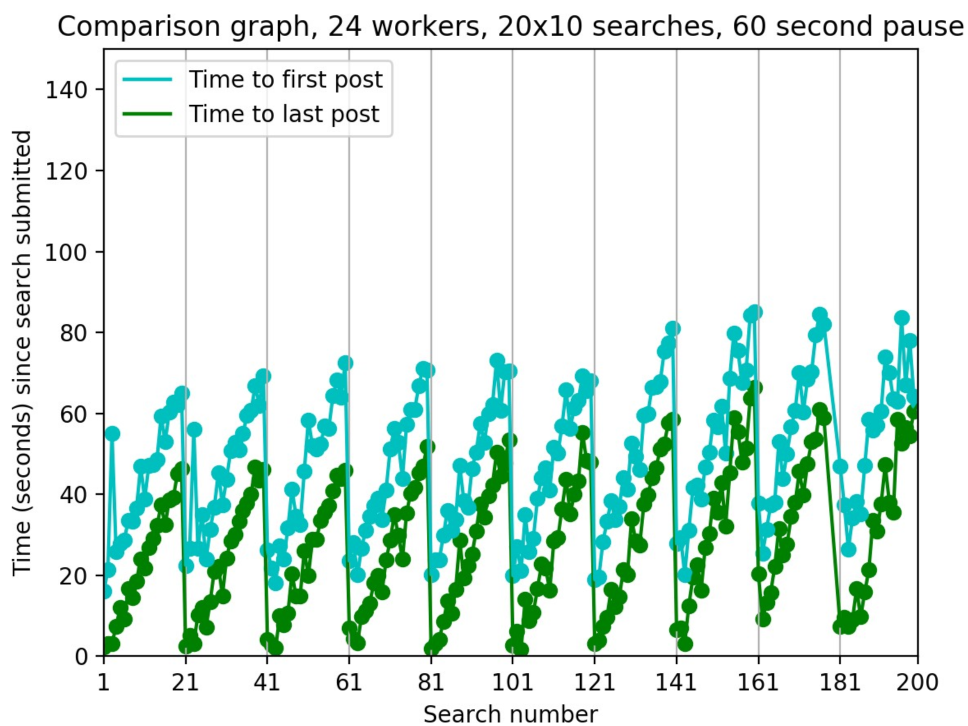

Figure 34: KPI comparison plot, 24 workers, 20x10 On/Off searches, 60 second delay

At 45 seconds between bursts, in Figure 31: On/Off timeline, 24 workers, 20x10 searches, 45 second delay, the system still times out during 1 of the 10 bursts, but the remaining bursts are mostly stable, with no other timeouts and only 3 searches failing to generate the full set of websocket packets. Figure 33: KPI comparison plot, 24 workers, 20x10 On/Off searches, 45 second delay tells a much less positive story, however, with the KPIs rising rapidly throughout the trial. Even after the timeout period, the next burst has a minimum time to first post of 24 seconds, well above the expected reasonable limit for isolated searches. Interestingly, the minimum time to first post for the 5th burst, at 30 seconds, already exceeds the minimum value for the 10th burst.

With 60 seconds between bursts, Figure 32: On/Off timeline, 24 workers, 20x10 searches, 60 second delay has no full bursts timing out and only 3/200 total searches failing to return any results. Figure 34: KPI comparison plot, 24 workers, 20x10 On/Off searches, 60 second delay, however, demonstrates that the KPIs are still slowly rising between bursts, with the final burst having a minimum time to first post of 7 seconds. In a similar fashion to Figure 23: KPI comparison plot, 32 workers, 128 searches, Phoenix is still suffering a progressive slowdown even though no searches are actively timing out, even with the system receiving an average of 1 search every 3 seconds, a third of the average frequency during the single-burst trials.

4.5 System Performance Breakpoints

From these results, it is clear that the time to first post and time to last post of search requests during a period of sustained input follow a consistent tri-modal behaviour.

1. When the rate of searches ingested is sufficiently low relative to worker capacity, system KPIs are flat. All search queries are processed as though they were run in isolation, with all searches receiving their first posts in 2-3 seconds and their last posts in 18-26 seconds, representing the fastest possible throughput of the Phoenix data retrieval pipeline. See the timings present in Figure 7: Isolated single search packet dispatch timings, and see Phoenix operating in mode 1 in Figure 26: KPI comparison plot, 64 workers, 128 searches.

2. When the rate of searches ingested is high relative to worker capacity, the job queue begins to fill. Time to first and last post increase with each subsequent search at a linear rate that grows larger as the ingested workload scales up relative to the worker count. Customers perceive the slower performance and suffer a reduced user experience on the application, potentially firing off additional requests rather than waiting patiently for results to eventually arrive. See Figure 23: KPI comparison plot, 32 workers, 128 searches for an example of mode 2 operation.

3. When load is high relative to worker capacity, and the job queue has filled to the point that time to first post exceeds approximately 70-100 seconds, search requests begin to time out. Fewer or no posts are sent for these requests, effectively preventing Phoenix users from meaningfully interacting with the system. The timeouts will continue for a period of time until the queue is

sufficiently flushed, at which point searches will resume executing in mode 2. See Figure 17: Single burst timeline, 16 workers, 128 searches and Figure 21: KPI comparison plot, 16 workers, 128 searches for a timeline and KPI comparison chart of Phoenix transitioning from mode 2 into mode 3 and back multiple times.

During mode 1, response times are at the minimum possible under the application. Any further resource allocation by Echosec constitutes entirely wasted expense that provides no value to the customer, as the system's response time is bounded by the delays induced by third-party data providers that are external to Phoenix. Moreover, retaining performance at this level is prohibitively expensive. As demonstrated, the system requires approximately 64 workers to sustain operation in mode 1 while receiving inbound searches at a rate of one per second (Figure 26). Handling On/Off bursts, during which searches arrive at an even faster rate, would require additional capacity to avoid the KPI increases of mode 2 occurring during the bursts themselves, even if the system as a whole is able to remain stable during the burst. See Figure 34: KPI comparison plot, 24 workers, 20x10 On/Off searches, 60 second delay for an example of this, noting the mode 2 behaviours during each burst as the TTFP / TTLP trend upwards before resetting for the next On/Off burst. Resource scaling to meet demand is possible for gradual shifts in throughput, but Echosec's erratic spikes in user demand suggest that scaling must be done reactively, as opposed to a proactive approach scaling based on time of day or day of week, and any scaling that takes longer than 3 to 20 seconds to complete will not complete quickly enough to preserve time to first or last post, respectively, for users in a given input burst. As such, retaining performance in mode 1 seems prohibitively expensive for Echosec.

During mode 3, some user requests fail to receive data entirely. This potential for data loss means that customers cannot count on their requests returning all desired data, which in turn means that customers must retry their search requests. Retried search requests will further increase the queue delays, keeping the system in mode 3 and leaving customers in a state of consistently-poor user experience. As data loss is not an acceptable solution, Echosec cannot permit Phoenix to rest in mode 3 on a consistent basis. As mode 3 is unacceptable due to user experience and mode 1 is unacceptable due to costs, the conclusion is that Phoenix should be pushed to operate in mode 2, and that the points of greatest concern for Echosec should be the performance within mode 2 and the transition points to modes 1 and 3.

The transition point from mode 2 to mode 3 occurs when search requests begin to time out. This point depends on multiple factors: the number of worker nodes running on Phoenix, the number of searches being delivered per second, and the number of searches expected to be received in a row. This third point, the number of consecutive searches expected, is perhaps the most interesting of the three. As shown throughout this work, Phoenix will remain in mode 2 with worsening customer-facing KPIs until a TTFP threshold is passed and mode 3 begins. Echosec has the opportunity to make a decision about what KPI levels are acceptable to their customers and to decide how many concurrent searches they wish to support, and the combination of these two factors will

dictate the resource levels that Echosec should optimally provide as well as the rate at which those resources should scale up as Echosec's customer base expands.

While resource levels determine the slope of the system in mode 2, the transition point to mode 3 can be controlled by altering the queue timeout delay on pending search requests. Increasing the timeout interval will push back the transition point, allowing more searches to be performed before mode 3 begins, but also increasing the maximum time to first post and last post observed by users. Echosec must decide what timings are acceptable to their customers, drawing from both existing literature and from customer surveys, to determine the maximum delays that are acceptable.

Notably, in Table 2: Locations and timings of timeout bursts during 128-search trials, the maximum TTFP before mode 3 began was 107.5 seconds. This value is far in excess of the 3 seconds recommended for a responsive web applications [19], and while Echosec may be willing to support a longer duration due to the nature of their platform, the leap from 3 seconds to over 100 seconds is large. More importantly, as shown by Table 2, the TTFP resets to a lower value after Phoenix exists a timeout burst, but currently this value at ~65 seconds is still far in excess of what a customer would likely consider acceptable. By shortening their queue timeout values, Echosec would cause more users on Phoenix to suffer timeouts, but would also cause the application's KPIs to reset to something more acceptable to humans after the timeout burst completes, which would increase the average number of customers who experience positive KPIs.

4.6 Summary

In chapter 4:Results, Phoenix was tested against static workloads under varying conditions in order to determine how the three quality-of-service metrics were affected. In 4.1:Isolated input, single search requests were sent to Phoenix and a baseline was found for these metrics when the system was under no other load. In 4.2:Single burst input, increasing search count, multiple trials were run with an increasing number of searches sent at a regular interval. During 4.2, the time to first post and time to last post was observed to grow at a linear rate for a period of time with all websocket packets being broadcast. After this period of linear growth, however, Phoenix was observed to begin timing out search requests for a period, with partial or no websocket packets being broadcast, and Phoenix eventually began cyclically alternating between sending results with increasing TTFP/TTLP and timing out results.

In 4.3:Single burst input, increasing worker count, the search count was held constant across each trial as the number of job worker processes was altered. Higher worker counts were shown to reduce the rate of increase of TTFP/TTLP for consecutive searches and to increase the number of consecutive searches that can be sent before the first timeout occurred.

In 4.4:On-off inputs, the search request generation pattern was changed to alternate between bursts of search requests and periods of downtime, following the On/Off technique. During these trials, search metric behaviour was observed to cluster based on

the bursts of search requests, with some bursts suffering timeouts and subsequent bursts entirely succeeding.

Finally, in 4.5:System Performance Breakpoints, the behaviour of Phoenix was determined to be tri-modal, based on the results seen previously. The second of these three modes, where TTFP/TTLP are increasing while all searches are returning all results, was determined to be the most valuable to Echosec based on both customer experience and associated resource costs. This section also made recommendations on adjustments to the timeout values currently being used by Phoenix, to increase the percentage of customers that receive search results within an acceptable period of time.

5 Conclusions and Future Work

This chapter summarizes the objectives, results, and limitations of this work. Future efforts that should be undertaken by Echosec will be recommended, both to implement improvements proposed and to investigate new research opportunities uncovered as a result of this work.

5.1 Objective Evaluation

The objective of this thesis was to determine suitable quality-of-service metrics for Phoenix, to examine the behaviour of these quality-of-service metrics under varying operating conditions, to show that these metrics shift between various modes of operation, and to propose a means of controlling the mode of operation in which Phoenix is operating. The first of these objectives was completed by chapter 3.3:Key Performance Indicators, while the remainder were executed throughout chapter 4:Results.

The conclusion was that Phoenix exhibits tri-modal behaviour, with the current mode of operation depending on the volume of search requests relative to the worker count and the length of the time to first post. Echosec was encouraged to hold Phoenix in the second mode, controlling the rate of growth of Phoenix's customer-facing performance metrics by adjusting the worker count and altering the transition point into the third mode by changing the timeout period on queued jobs.

Further changes were recommended based on observations from chapter 3.1:Phoenix Architecture Details, where flaws in the queueing structure of Phoenix were identified that cause current timeout values to be incorrectly applied, leading to search requests spending longer queued than is intended. As the transition point between operating modes 2 and 3 is dependent on this timeout value, making these changes will further the ability of Echosec to control this transition point.

5.2 Limitations

The primary limitations of this work stem from the use of the static test harness. This test harness fully and accurately simulates the average-case response from each of the data sources used by Echosec, but does not account for exceptional cases. For instance, a data provider may suffer an internal error and begin timing out requests, greatly increasing the delay before the associated HTTP calls are received, or a change to API behaviour may cause all calls to that API to result in an error. These temporary effects may result in behaviour not seen during this trial. This limitation could be resolved by improving the behaviour of the test harness by incorporating state-based changes to the query data behaviours.

Additionally, this work is limited by the number of trials executed. Executing single trials provided significant insights into the behaviour of the system and its clear tri-modal structure, but more decisive conclusions about these behaviours could be reached from repeated execution of the trials, such as more information about how timeouts in mode 3 are distributed. Several trials, including trials in mode 1, demonstrated increasing levels of noise over the duration of the trial, yet no solid conclusions may be drawn about this

noise with the confidence levels present on the current single-trial runs. With increased confidence, this noise could be demonstrated as irrelevant, or could indicate a previously-unidentified non-stationary behaviour in the system.

Finally, the tests done in this work focused around single periods of extreme usage. These spikes are the moments of greatest concern to Echosec and of highest risk to Phoenix's customer-facing KPIs, which is why the decision was made to emphasize these spikes. However, Phoenix as a whole operates at all times, not just during these bursts of throughput. Investigating periods of relative downtime would provide more insights into how the system recovers from spike periods, and more information about how quickly dynamic scaling can remove worker capacity.

5.3 Future Work

Future work on this subject should resolve the limitations discussed above. Most notably, increasing the volume of trial data via automated re-execution and capture of trials would allow for a discussion on the ergodicity of the Phoenix platform's KPIs. Repeated trials would also improve confidence in the tri-modal behaviour of Phoenix, and would provide more precision on the conditions of the mode transition points.

Expanding these extended trials to consider longer periods of downtime would also be valuable. This could be combined with the repeated trials proposed immediately above, with monitoring of the system continuing during periods of downtime between repeated activity spikes to collect data on the behaviour of the system over hours or days of periodic use. These data would more accurately reflect the production usage of Phoenix over realistic periods.

Any increased testing of this variety would require improvements to the test harness to ensure the Phoenix platform remains operational. During trials, individual Phoenix workers were observed to fail, requiring manual intervention after each trial to reset the state of the system before testing could continue. During these single trials, workers failing was accepted as part of the platform's behaviour, with any consequences of these trials being reflected in the results received, but testing for hours or days would require the state of these workers to be restarted when outages are detected to avoid the platform losing large parts of its capacity.

5.4 Recommendations

Echosec should adjust their timeout values to better reflect their desired response times. Even assuming that Echosec customers who have paid for access to the service are more willing to wait for data than the average website user, the current literature's recommendation of 3 seconds or less is an order of magnitude separated from the 60 second timeout currently used in Phoenix. This timeout value causes search requests on Phoenix to remain in queue far longer than they should, likely encouraging customers to retry their searches and therefore further increase the queue length and expected delays.

Additionally, Echosec should investigate the use of timeouts on additional search requests. The current timeout value does not correctly expire searches older than 60

seconds because, as discussed previously, the `default` queue used for all `ExecuteSocialQueryAbstract` jobs is shared with the `QueryProviders` jobs. Since the `QueryProviders` jobs occur before the `ExecuteSocialQueryAbstract` jobs, and because the `QueryProviders` jobs do not have a timeout set, search requests may take considerably longer than the expected 60-second timeout window to either resolve or time out. This flaw limits the ability of Echosec to meaningfully adjust their request timeouts in a controllable manner. Echosec could consider adding a secondary timeout to `QueryProviders` jobs and dividing the duration between these two timeouts. Alternately, Echosec could investigate options for maintaining a continuous timeout that persists throughout the lifespan of a search request, rather than each individual job in the processing pipeline being assigned a separate timeout value.

Monitoring improvements should be made to better detect and respond to worker outages on Phoenix. This was mentioned as a requirement for future work on the system at production scales, but even without running future trials, this work has revealed that Phoenix has a strong risk of worker nodes failing permanently in production environments that should be resolved.

Bibliography

[1] Twitter Inc, “Q4 and Fiscal Year 2018 Letter to Shareholders,” *Twitter, Inc Financial Information*, 07-Feb-2019. [Online]. Available: https://s22.q4cdn.com/826641620/files/doc_financials/2018/q4/Q4-2018-Shareholder-Letter.pdf. [Accessed: 13-Mar-2019].

[2] Amazon Inc, “EC2 Instance Pricing,” *Amazon Web Services*. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/. [Accessed: 13-Mar-2019].

[3] T. Otwell, “Laravel,” *Laravel - The PHP Framework For Web Artisans*. [Online]. Available: https://laravel.com/. [Accessed: 05-Nov-2018].

[4] I. Fette, Google, Inc., A. Melnikov, and Isode Ltd., “RFC 6455 - The WebSocket Protocol,” Internet Engineering Task Force, Dec. 2011.

[5] A. Bahga and V. K. Madisetti, “Synthetic Workload Generation for Cloud Computing Applications,” *JSEA*, vol. 04, no. 07, pp. 396–410, 2011.

[6] P. Barford and M. Crovella, “Generating representative Web workloads for network and server performance evaluation,” *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 151–160, Jun. 1998.

[7] Cheng-Shang Chang, “Stability, queue length, and delay of deterministic and stochastic queueing networks,” *IEEE Trans. Automat. Contr.*, vol. 39, no. 5, pp. 913–931, May 1994.

[8] G. Florin and S. Natkin, “Necessary and sufficient ergodicity condition for open synchronized queueing networks,” *IIEEE Trans. Software Eng.*, vol. 15, no. 4, pp. 367–380, Apr. 1989.

[9] J. Campos, G. Chiola, and M. Silva, “Ergodicity and throughput bounds of Petri nets with unique consistent firing count vector,” *IIEEE Trans. Software Eng.*, vol. 17, no. 2, pp. 117–125, Feb. 1991.

[10] V. Cortellessa, A. Di Marco, and C. Trubiani, “An approach for modeling and detecting software performance antipatterns based on first-order logics,” *Softw Syst Model*, vol. 13, no. 1, pp. 391–432, Feb. 2014.

[11] K. M. Chandy and J. Misra, “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs,” *IIEEE Trans. Software Eng.*, vol. SE-5, no. 5, pp. 440–452, Sep. 1979.

[12] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms,” *Softw. Pract. Exp.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[13] J. Espadas, A. Molina, G. Jiménez, M. Molina, R. Ramírez, and D. Concha, “A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures,” *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 273–286, Jan. 2013.

[14] Y. O. Yazir *et al.*, “Dynamic Resource Allocation in Computing Clouds Using Distributed Multiple Criteria Decision Analysis,” in *2010 IEEE 3rd International Conference on Cloud Computing*, Miami, FL, USA, 2010, pp. 91–98.

[15] P.-J. Maenhaut, H. Moens, B. Volckaert, V. Ongenae, and F. D. Turck, “Resource Allocation in the Cloud: From Simulation to Experimental Validation,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, Honolulu, CA, USA, 2017, pp. 701–704.

[16] L. V. Casaló, C. Flavián, and M. Guinalíu, “The role of satisfaction and website usability in developing customer loyalty and positive word-of-mouth in the e-

banking services," *Intl Jnl of Bank Marketing*, vol. 26, no. 6, pp. 399–417, Sep. 2008.

[17] S. Kim and L. Stoel, "Apparel retailers: website quality dimensions and satisfaction," *Journal of Retailing and Consumer Services*, vol. 11, no. 2, pp. 109–117, Mar. 2004.

[18] G. Sahi, "User Satisfaction and Website Usability: Exploring the Linkages in B2C E-Commerce Context," in *2015 5th International Conference on IT Convergence and Security (ICITCS)*, Kuala Lumpur, Malaysia, 2015, pp. 1–4.

[19] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: a platform for high-performance internet applications," *Oper. Syst. Rev.*, vol. 44, no. 3, p. 2, Aug. 2010.

[20] J. C. Fernandes, K. A. Swannie, N. J. Turner, M. R. Anderson, and J. P. Jubinville, "Digital publication monitoring by geo-location," 9704169B2, 11-Jul-2017.

[21] "Guzzle," *Guzzle official documentation*. [Online]. Available: http://docs.guzzlephp.org/en/stable/. [Accessed: 05-Nov-2018].

[22] "Jaeger Tracing," *Jaeger: open source, end-to-end distributed tracing*. [Online]. Available: https://www.jaegertracing.io/. [Accessed: 05-Nov-2018].

[23] "Apache Cassandra," *Apache Cassandra*. [Online]. Available: http://cassandra.apache.org/. [Accessed: 05-Nov-2018].

Appendix A – Single-Burst Workload Generation

```
#!/usr/bin/python3

import requests
from bs4 import BeautifulSoup as BS
import re
import json
import time
import sys

# This script generates a single burst of input data to the
application.
# It does so by sending one request per second
# (technically one request followed by a one-second sleep)
# for a fixed number of cycles.

# Step 0: setup.
session = requests.Session()

targetWorkloadSize = 4
if len(sys.argv) >= 2:
      targetWorkloadSize = int(sys.argv[1])

# Step 1: get a valid JWT token from the login page.
print('Accessing login page...')
introResponse = session.get('http://app.echosec.test/login')
print(introResponse.url)
csrfToken = None
for meta in BS(introResponse.text,
'html.parser').find_all('meta'):
    if 'name' in meta.attrs and meta.attrs['name'] == 'csrf-
token':
        csrfToken = meta.attrs['content']

if csrfToken is None:
```

```
    print('ERROR: No CSRF token found. Aborting.')
    exit()
print('CSRF token: ' + csrfToken)

# Step 2: log into the application using our JWT token and known
development credentials.
print('\nExecuting secure authentication...')
loginPayload = {'email': 'nick@echosec.net', 'password':
'secret', '_token': csrfToken}

loginResponse = session.post('http://app.echosec.test/login',
data=loginPayload)
print(loginResponse.url)

# Extract the XSRF token.
xsrfToken = None
matches = re.search('window.Echosec.*"jwtToken":"(.*?)"',
loginResponse.text)
if matches:
    xsrfToken = matches.group(1)

if xsrfToken is None:
    print('ERROR: No XSRF token found. Aborting.')
    exit()
print('XSRF token: ' + xsrfToken)

# Step 3: begin executing search requests.
print('\nBeginning workload generation up to %s requests...' %
targetWorkloadSize)
searchParameters = {'token': xsrfToken}
searchPayload = {'keywords':
['puppy'],'address':'','address_bounds':'','author_id':'','author
_query':'','providers':
['fourchan','flickr','foursquare','medium','news','okru','pintere
st','reddit','tumblr','twitter','vk','vimeo','wikipedia','youtube
```

```
'],'start_at':None,'end_at':None,'images_only':False,'video_only'
:False}
# Specifying Content-Type header directly because apparently I've
only got 'requests 2.2.1' installed, despite Pip supposedly
bringing in 2.9.1. Probably an installation/module path issue
that I don't want to debug.
searchHeaders = {'Origin': 'http://app.echosec.test', 'Content-
Type': 'application/json'}

for i in range(0, targetWorkloadSize):
    print('\nPosting search ' + str(i))
    searchResponse =
session.post('http://app.echosec.test/api/v1/search',
data=json.dumps(searchPayload), params=searchParameters,
headers=searchHeaders, allow_redirects=False)
    print('Response code: ' + str(searchResponse.status_code))
    time.sleep(1)

# Step 4: Cleanup.
print('\nDone! Dispatched %s requests.' % targetWorkloadSize)
```

Appendix B – On/Off Workload Generation

```
#!/usr/bin/python3.5

import requests
from bs4 import BeautifulSoup as BS
import re
import json
import time
import copy
import asyncio
from concurrent.futures import ThreadPoolExecutor


# Workload generator that uses an on-off pattern, generating
bursts of requests with downtime in between.
# Requires Python 3.5 to support the async HTTP pattern used.
# Asynchronous code borrowed from https://hackernoon.com/how-to-
run-asynchronous-web-requests-in-parallel-with-python-3-5-
without-aiohttp-264dc0f8546


# Fetch a single search.
def fetchSearch(instanceNumber, session, searchParameters,
searchPayload, searchHeaders):
    searchResponse =
session.post('http://app.echosec.test/api/v1/search',
data=json.dumps(searchPayload), params=searchParameters,
headers=searchHeaders, allow_redirects=False)
    print('Instance %s response code: %s' % (instanceNumber,
searchResponse.status_code))
    return searchResponse


# Run a group of searches asynchronously.
async def runSearchesAsynchronous(session, searchParameters,
searchPayload, searchHeaders, searchesPerBurst):
    with ThreadPoolExecutor(max_workers=searchesPerBurst) as
executor:
        loop = asyncio.get_event_loop()
```

```
        tasks = [
            loop.run_in_executor(
                executor,
                fetchSearch,
                *(i, session, searchParameters, searchPayload,
searchHeaders)
            )
            for i in range(0,searchesPerBurst)
        ]
        for response in await asyncio.gather(*tasks):
            pass


def main():
    # Step 0: setup.
    session = requests.Session()


    # Step 1: get a valid JWT token from the login page.
    print('Accessing login page...')
    introResponse = session.get('http://app.echosec.test/login')
    print(introResponse.url)
    csrfToken = None
    for meta in BS(introResponse.text,
'html.parser').find_all('meta'):
        if 'name' in meta.attrs and meta.attrs['name'] == 'csrf-
token':
            csrfToken = meta.attrs['content']


    if csrfToken is None:
        print('ERROR: No CSRF token found. Aborting.')
        exit()
    print('CSRF token: ' + csrfToken)


    # Step 2: log into the application using our JWT token and
known development credentials.
    print('\nExecuting secure authentication...')
```

```
    loginPayload = {'email': 'nick@echosec.net', 'password':
'secret', '_token': csrfToken}

    loginResponse = session.post('http://app.echosec.test/login',
data=loginPayload)
    print(loginResponse.url)

    # Extract the XSRF token.
    xsrfToken = None
    matches = re.search('window.Echosec.*"jwtToken":"(.*?)"',
loginResponse.text)
    if matches:
        xsrfToken = matches.group(1)

    if xsrfToken is None:
        print('ERROR: No XSRF token found. Aborting.')
        exit()
    print('XSRF token: ' + xsrfToken)

    # Step 3: begin executing search requests.
    print('\nBeginning workload generation...')
    searchParameters = {'token': xsrfToken}
    searchPayload = {'keywords':
['puppy'],'address':'','address_bounds':'','author_id':'','author
_query':'','providers':
['fourchan','flickr','foursquare','medium','news','okru','pintere
st','reddit','tumblr','twitter','vk','vimeo','wikipedia','youtube
'],'start_at':None,'end_at':None,'images_only':False,'video_only'
:False}
    searchHeaders = {'Origin': 'http://app.echosec.test',
'Content-Type': 'application/json'}

    # On-off search pattern.
    searchesPerBurst = 20
    totalBursts = 10
    timeBetweenBursts = 30 # Seconds
```

```
    # Begin the async insanity!
    startTime = time.time()

    loop = asyncio.get_event_loop()
    for currentBurst in range(0, totalBursts):
        print('\nBeginning burst %s at time %s' % (currentBurst,
time.time() - startTime))
        future =
asyncio.ensure_future(runSearchesAsynchronous(session,
searchParameters, searchPayload, searchHeaders,
searchesPerBurst))
        loop.run_until_complete(future)
        print('Finished loop %s, sleeping.' % currentBurst)

        time.sleep(timeBetweenBursts)
        print('Finished loop %s including sleep.' % currentBurst)

    # Step 4: Cleanup.
    print('\nDone!')

main()
```