School of Computing
FACULTY OF ENGINEERING AND PHYSICAL SCIENCE

UNIVERSITY OF LEEDS

Quaranteeing QoS of Latency Critical Systems Using a Hybrid Scaling Mechanism

Christopher James London

Submitted in accordance with the requirements for the degree of BSc Computer Science

2020/21

40 credits

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Deliverables 1, 4, 5	Report	Assessor and Supervisor (03/05/21)
Deliverables 2, 3	Github Repository	Supervisor and Assessor (03/05/21)

Type of project: Exploratory Software

Summary

The microservice architecture is being increasingly used in place of traditional, monolothic architecture in order to facilitate the creation and maintenance of complex applications and systems. Applications developed using the microservice architecture separate the different functions of a system into many, small, concentrated components; these components are often scaled based on load in order to meet increase demand. These components exhibit complex relationships, which renders scaling a large application complex and difficult to manage. This project devises a working autoscaling mechanism that utilises these relationships to inform scaling decisions and demonstrates that an autoscaler that acknowledges and utilises these relationships can almost match the performance of traditional scaling mechanisms.

Acknowledgements

I would like to thank both of my supervisors, Professor Jie Xu and Dr Renyu Yang for their constant advice, guidance and support during my dissertation. Their expertise in this subject area has been invaluable and provided me with the means to conduct my own further research. I would also like to thank my assessor, Professor Karim Djemame for his support and advice during my progress meeting and feedback on the interim report. The papers provided in the feedback from my interim report provided some of they key motivation for my later project work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scaling modern applications has become an essential feature in the design, development, and maintenance of computer systems that are increasingly hosted in the cloud using microservice architectures [1]. The ability to leverage scaling techniques to service a sudden or persistent increase in demand on one or many components whilst meeting one or more agreed upon Service Level Agreements (SLA) is essential to a functioning microservice architecture.

The microservice architecture has become an influential architectural pattern in modern applications owing to the pertinence and popularity of cloud computing [2]. Microservices are often used to mitigate or alleviate the pitfalls often found with monolith systems such as conflicting functions, complex dependency tree management, technology lock-in, wide scope for bugs and most importantly for this project, limited scalability [2][3]. A single microservice "is a cohesive, independent process interacting via messages" that typically has a very narrow, small function as a part of a larger, distributed application [2]. This separation of concerns leads to a decreased scope for bugs to arise and a collection of more focused, individual components that are heterogeneous in terms of their general technology stack [4]. The autonomous nature of microservices also facilitates incremental development and the faster testing and deployment of new components to an existing system [2][5].

A distributed system consisting of multiple cohesive components also introduces some challenges in terms of component dependency; a single component may depend on another component in order to complete a given function [4]. This introduces some difficulties when designing and developing an application using the microservice architecture; particularly in terms of the ability to meet demand for a service that exceeds anticipated demand. Increased load on one component of a distributed system often "fans out" to other components, leading to a cascading affect in terms of load, propagating errors from one slow or faulty component to others. Given that microservice components are inter-dependent, a system is only as fast as its' slowest component and it is not uncommon for a small subset of components to be bottlenecks in a distributed system.

Microservices are often containerised, being deployed in very small VM-like environments that have access to a smaller number of resources than is present on the physical machine [2][4]. Containerisation of components facilitates the ability to scale specific parts of a wider application, a technique employed to mitigate against the effects of increased load [6]; scaling one or more components in a system can be used to adapt to changes in demand for a given service. Vertical scaling is the technique in which a container is taken down and redeployed with adjusted access to resources such as CPU cores, RAM size or disk space to adequately meet an increased demand [7]. Horizontal scaling is used to increase the number of replicas of a given component, leading to multiple identical components completing the given function independently and asynchronously of each other [8][9]. These new replicas can be brought up and torn down without affecting other replicas as and when developer-defined thresholds are reached [8], allowing the component to continue serving requests from other components in the system.

1.1 Aim

This project aims to devise an algorithm for scaling microservices based on the services that depend on them in order to minimize latency in microservice applications. An algorithm will be developed and then implemented in a custom autoscaler that is developed using engineering best practices. The results of using this autoscaler in a production-like environment will then be contrasted with traditional scaling mechanisms.

1.2 Objectives

- Develop an auto-scaling technique that scales a given application using metrics from other upstream components
- Compare and contrast the developed autoscaler in consistent load and short burst/spike scenarios
- Compare and contrast the developed autoscaler with a traditional autoscaling method used in industry
- Determine whether dependency-based autoscaling is feasible and viable

Chapter 2

Background Research

2.1 Containerisation

The need to virtualise resources and applications running in the cloud stems from the expense of running individual machines on bare metal servers [10]. Running applications on dedicated servers is not only wasteful in terms of the required computational power but also in terms of monetary cost [10]. Containerisation is a relatively modern alternative to traditional virtualisation using hypervisors [11]. Containers virtualise an OS, sharing the underlying kernel which makes containers far more lightweight than hypervisor implementations where a VM sees themselves as a true, physical machine in its own right with an individual kernel [12]. The lightweight nature of containers provides the means for increased performance over traditional VM's, which is a key factor when running tens, hundreds of even thousands of instances of a given application [11][12]. Container orchestration software such as Docker also have very simple mechanisms to mount and share directories and physical devices between containers which are incredibly useful in certain scenarios [11].

Isolation is an important feature provided by containers, encapsulating the implementation details of other containers and ensuring processes only interact when explicitly required to. Most modern container orchestration systems such as Docker run a single process/microservice in each container. Containerisation lends itself to the microservice architecture, where a larger system is comprised of many autonomous components that interact through predefined contracts.

2.2 Microservices

Microservices are small, atomic components of a larger system, individually they solve a very small and specific problem, often with very little scope for adaptation. Having a microservice whose sole role is to receive a piece of data simply to enrich it is a standard use case, there is no definition of how much work a microservice should do. In real-world applications, these services are composed of one another, calling other services for access to their data and functionality [2]; it is this composition which leads to large, complex systems.

2.2.1 Benefits

This style of application architecture alleviates some common downsides found with traditional monolithic systems, such as technology lock-in, dependency tree issues, and the inability to scale individual parts of an application to meet demand for a specific part of the system [2].

2.2.2 Performance Degradation and User Impact

Many modern systems, especially those offered via the Software as a Service (SaaS) business model by cloud providers such as Amazon Web Services are designed using the microservice architecture [13]. As previously discussed, scalability of these systems is relatively simple in technical terms; the real challenges arise when prioritising agreed upon SLA's, cost, energy efficiency and performance [13].

Managing costs and ensuring a given service or component of a service is not over-provisioned is in the interest of the owner of a system, but this needs to be balanced with the need for a given system to perform adequately within the bounds of SLA's [13]. A service that is under-provosioned will result in higher latency, decreased performance and bottlenecks in larger composite systems [13]. Because of this, we require a sufficient, intelligent method to scale services and components when required in order to ensure adequate performance, whilst also balancing the need to minimise cost and energy consumption.

2.3 Metrics

Applications deployed in the cloud require continuous monitoring, often by way of analysing system metrics such as CPU, RAM, disk and network usage in almost real-time; this data analysis is often used when making decisions with regards to changing environmental conditions, namely, adjusting the number of available replicas of a given service or adjusting the resources a given service has access to [12][14].

Typically, every atomic component of a given system will publish metrics related to their own performance to a metric store that can be used to gain insight into the historical and immediate performance of a given service. These metrics may differ slightly, including not only traditional system metrics such as CPU usage but also domain and application specific metrics such as latency or Kafka Topic Lag. The metrics used to determine which action to take is sometimes domain specific. It is feasible for some components to be adjusted based on CPU usage, other based on network I/O and sometimes metrics such as lag on a Kafka Topic may be a good indicator a given component is struggling to cope with a burst in load; context and semantics have a significant meaning in relation to how and when scaling should occur [15].

Collecting system-wide metrics (which are concerned with the overall performance of the system as opposed to individual components) is also common, but is less likely to be used to inform scaling decisions due to the lack of granularity and specificity of which part of the system is slow and may be acting as a bottleneck. If individual components are being scaled adequately, the entire system should be running satisfactorily.

2.4 Load

Load is generated by different sources, the type of load is dependent on the type of component; for web-services it may be in the form of individual requests being received, in streaming applications it may be the number of messages being placed on to an individual pub-sub topic. In essence, load can be thought of as the total number of individual units of work generated for a given component, where one HTTP request, Message, or database transaction are examples of individual units of work [16].

Load on a system is rarely constant, and changes over time or in response to external factors. In large microservice architectures it is standard for there to be a set of distinct component types such as databases, web-services, and streaming applications which are inherently different in their implementation and inner workings. Increased load on these components can lead to decreased response time, timeouts, requests being rejected due to rate-limits, and slow-downs due to locks and other restrictions which often lead to SLA violations for individual components or the entire system.

There are two types of load that we consider in the cloud [16]. Transactional load is best characterized with web services that service individual requests, databases that handle individual transactions and so forth; the load is not long running. A streaming application that reads off of a pub-sub message topic, handles a message and then places results in a database is an example of a batch-load application, the work is long-running; there are batches of data that are handled constantly by the application. [16].

2.4.1 Fan-out

Components are inter-dependent, and in order to fulfill their own requests make use of functionality and data exposed by other components in the wider system [15]. This leads to a 'critical path' being formed for a given function, comprised of the different components needed to fulfill a given user request [17]. It is when a component on this critical path becomes saturated that system-wide slow downs occur [17][4]. One of the drawbacks of this inter-dependency is the tendency for a single, slow component to fail-slow, impacting other faster components of the system which depend on it.

2.5 Scaling

A key benefit of using a microservice oriented system is the ease of which you can scale individual components in response to a change in demand, which is often difficult or even impossible with a monolithic system [2][3]. Whilst it is true that if a system is designed correctly the act of scaling can be incredibly simple, the decisions that go into determining when, how and what to scale are not always so clear and often depend on the implementation details of a given service [14]. Sometimes, scaling may not help with SLA violations due to limitations external to the system in question, external API calls which are rate-limited cannot be easily overcome, for example.

2.5.1 Strategies

Vertical Scaling

Vertical scaling is concerned with scaling an existing set of application instances by increasing the resources they have access to [9]. In some situations, simply increasing the allocation of physical hardware to a set of given instances is enough to prevent SLA violations and bring performance within a desired range.

Whilst vertical scaling can provide adequate increases in performance, there are some drawbacks. In particular, cold-starts are a significant issue when scaling vertically; in order to increase access to certain resource such as RAM and CPU cores, the instances need to be brought down, container resources adjusted, and then re-started. This can lead to a significant amount of time where there are no instances operating; there are some mitigation strategies such as bringing instances down gradually, but this then adds to spin-up time [18].

Horizontal Scaling

Horizontal scaling is an alternative method of scaling that does not increase access to physical resources, but instead spins-up more instances of the same application, usually by creating additional identical VM's/Containers running the application. Horizontal scaling is particularly useful in data-processing workloads where instances can work independent of each other [19].

One pitfall of horizontal scaling, is the lack of knowledge regarding diminishing returns when scaling horizontally; there is an open question as to when horizontal scaling does not provide significant performance increases [20]. It is also possible that a given component may not owe itself to scaling horizontally, especially where immediate data consistency is required.

2.5.2 Elasticity Policies

There are many different forms of elasticity. In general, elasticity can be considered "an automation on the concept of scalability with an aim optimize as best and quickly as possible" [12]. Systems that make use of scaling require some mechanism to identify what to scale, and when to scale it. Microservices can be considered synonymous with any other resource in a cloud network, and as such are governed in a similar way; especially in the context of an autoscaler. In order to determine whether there is a need to scale a given resource, an elasticity policy is defined and implemented by an autoscaler. This policy will define the thresholds needed in order to modify the provision of a given resource in a cloud system [16].

Elasticity policies have a variety of characteristics that define their behaviour such as their purpose, policy, method and so forth. There are two types of elasticity policy, predictive and reactive [12], reactive elasticity mechanisms respond to changes in the environment and load, whereas predictive methods attempt to preemptively scale resources before load can occur using methods such as time-series analysis and machine learning [12]. The purpose of elasticity can take different forms, in some cases elasticity may be employed in order to reduce costs, improve performance or increase availability. Increasing availability and improving performance involve ensuring a resource is scaled sufficiently such that an increase in load can be serviced and handled with minimal SLA violations [12].

2.5.3 Autoscalers

It is possible to adjust resource availability and performance using manual intervention, but it is often desirable to do this with as little human interaction as possible [14], which is where the need for autoscalers arise. In any automatic resource-provisioning system there exists a feedback loop which uses some input to determine the appropriate action(s) to take [16]. There are a handful of different approaches autoscalers can take, all implementing some elasticity policy, which are discussed below.

Schedule Based

Schedule-based autoscalers are a prominent implementation that use time as the key parameter for the scaler, leading to resources that are scaled on a schedule [12]. This type of implementation is effective in settings where the workload is predictable or there is significant historic data to infer when load is likely to increase [21]. Autoscalers of this type are powerful when load is accurately predicted, services that often act as a bottleneck can be adjusted in anticipation, removing the need to wait for a service to begins struggling before it is adjusted to handle increased load.
Whilst these autoscalers work well when increases in load are predictable or follow a trend, they are not at all effective with increases in load which do not follow a trend, or when load cannot be predicted. This significantly reduces the situations in which these types of autoscaler can be used with great effectiveness [21].

Rule Based

Whilst proactive schedule-based autoscalers use time as the key indicator to indicate when and how to scale a given service [12], rule-based autoscalers use a combination of rules and data such as performance metrics to determine if there needs to be an adjustment to the number of replicas or resources of a service [12][16][21]. The scaler can run in an infinite loop of monitoring metrics, analysing the data and scaling accordingly [12][16][21]. Scaling mechanisms that are rule-based are inherently reactive, they do not predict load on a service but instead react to changes as quickly as possible [12]

2.5.4 Self-determination

In traditional reactive scaling methods, components are assessed on an individual basis. Assuming load is unpredictable in some microservice application S, if component x begins to serve dependents y_1 and y_2 slower than usual, x may be scaled up according to the elasticity policy E of S. This only happens when x's metrics show a degradation in performance. Strategies that follow this general line of thinking are very common, and typically work well where a small delay is acceptable whilst instances are scaled horizontally or vertically, but may present a problem in latency-critical applications where cold-start time for new instances, or when a delay in the decision to scale presents a significant degradation of performance.

2.5.5 Predictive auto-scaling

Predictive auto-scaling is an explored research area within cloud computing; typically, prediction of future metric values for a given service is used to predicatively scale cloud resources in anticipation to meet potential future demand. Given some component x in system S with elasticity policy E and autoscaler A, A will attempt to predict the future values of metrics used to determine the scaling of x. If every component can be accurately scaled to meet future demand before a load-spike hits the application then, in theory, SLA violations should be less frequently experienced. The majority of predictive elasticity policies make use of time-series analysis, pattern recognition, and machine learning [12] to inform scaling decisions.

2.6 Summary

To summarise, this project will develop a hybrid, application-focused autoscaler that aims to improve performance, henceforth referred to as Scale Eye. Hybrid elasticity policies that combine both reactive and proactive approaches are an open research area that is yet to be explored in depth [12]. Scale Eye will implement a hybrid method that implements a reactive elasticity policy and combines this with the predictive scaling of other services.

The developed system should manage resources under load that is unpredictable, making use of metrics from upstream services that depend on a given service. Relationships between components are complex and as such, determining which services need scaling is inherently non-trivial [15]; a dependency graph will be used in order to assess the need to scale a given component in anticipation of future load. When determining which components to scale, we will not only consider the component showing a degradation in performance, but also its dependent services, scaling them in anticipation of the incoming load increase.

Chapter 3

Project Plan

In order to ensure this project is completed to a satisfactory standard and in good time, a solid plan needs to be established with key dates and deliverables. The project has a timeline of roughly seven and a half months, the majority of this time is likely going to be spent on research and design.

3.1 Methodology

There are various methodologies that would be suitable for this project. Agile [22] and Waterfall [23] are the two that were considered for this project, due to the experience the author has with both methodologies as well as their prevalence in software engineering. There are other methodologies, such as rapid prototyping that ultimately did not seem appropriate after surface research [24].

Agile

Agile is a modern methodology that places emphasis on rapidly producing results and iterating over previous deliverables to make changes and improvements. Agile focuses on adaptability and agility in development [25], allowing for rapid adaptations to the product being developed. Work is split into iterations, where each iteration is comprised of design, development and testing.

Whilst agile owes itself very well to software development in a setting where there are multiple stakeholders and external factors that may need to be accounted for during product development, this project is not singularly focused on software development. This project is also largely focused on literature review, analysis and experimenting with the environments being used. The project will need to be largely well defined before development can begin, and as such, the need to adapt to change is not a key concern in this case.

3.1.1 Waterfall

The waterfall method is a more structured, rigid methodology; the project is broken into individual tasks, and the person(s) responsible for project management define the time in which each task should be completed in. This methodology is largely being replaced by Scrum and Agile in the software engineering industry due to its rigid and bureaucratic nature [23][25]. However, given this project is being undertaken by a single person, and there is a distinct need to research and plan almost all aspects of the project before development and analysis can begin, it is the optimal choice in this case. There will likely be very minimal work that requires iterating on a previous deliverable, the rigid nature of the methodology is likely to be a positive feature in this project.

3.2 Timeline

Waterfall is the methodology chosen for this project. As such, a timeline of work to be completed has been created in the form of a Gantt Chart. There are seven phases, which all contain a set of tasks. Some of these tasks overlap due to the fact they can be completed in parallel, others overlap in order to allow some feedback from the newly beginning task to feed into the finalisation of another task; overlaps have been kept to a minimum wherever possible.

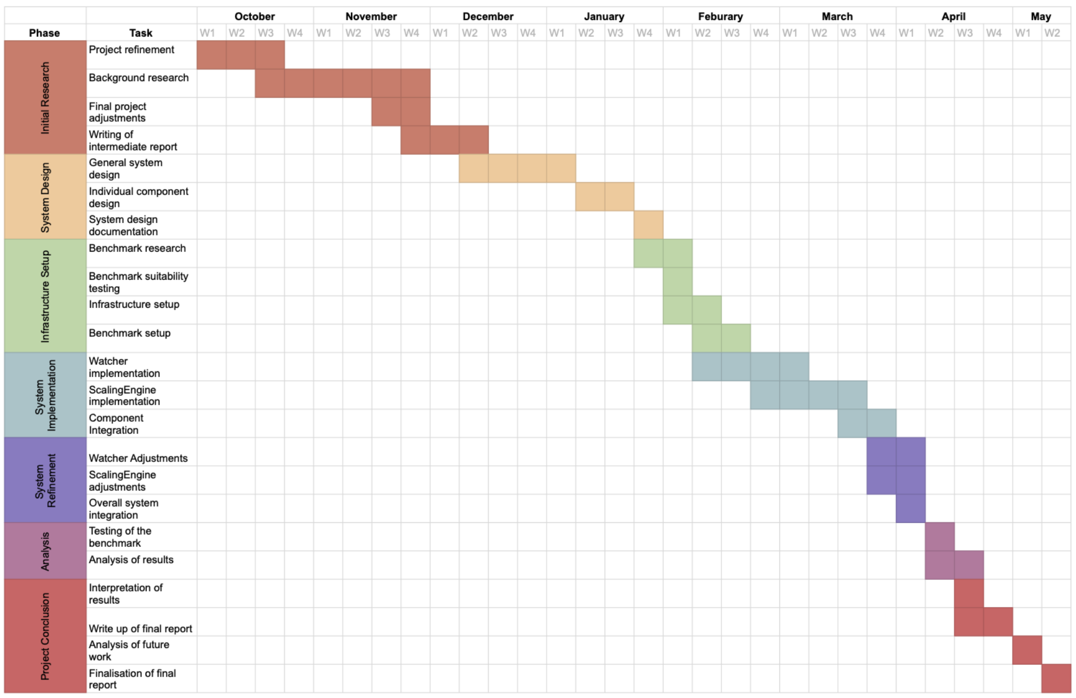
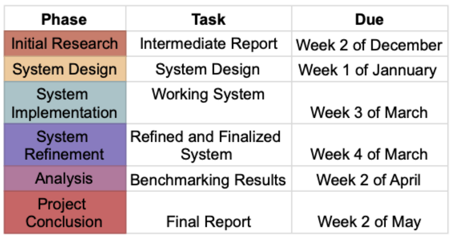


Figure 3.1: Project plan by week

3.3 Milestones

Below is a table of the key milestones for this project. They have been linked with the phases they correspond to. Achieving these milestones is essential in order to ensure the final project deadline is met in good time.



Phase	Task	Due
Initial Research	Intermediate Report	Week 2 of December
System Design	System Design	Week 1 of Jannuary
System Implementation	Working System	Week 3 of March
System Refinement	Refined and Finalized System	Week 4 of March
Analysis	Benchmarking Results	Week 2 of April
Project Conclusion	Final Report	Week 2 of May

Figure 3.2: Key Project Milestones

3.4 Code Management

Version control will be used throughout the project in order to manage the codebase. Different Git repositories will be used for the different components in order to have a granular view on the progress of the project. Continuous integration and deployment will be used in order to ensure code is correct and compiles.

3.5 Deliverables

The key deliverables for the project are:

1. An algorithm design for dependent auto-scaling
2. A functioning system that implements and utilises the aforementioned algorithm
3. The source code for the developed system
4. Results and analysis comparing and contrasting the results of the benchmark using
   (a) The developed system
   (b) Kubernetes's Horiztonal Pod Autoscaler (HPA)
   (c) No autoscaling mechanism
5. A final report discussing the project and the results as well as suggestions for future work

3.6 Success Criteria

This project has multiple objectives, and as such it is important to understand how success will be measured. Success will not be measured by the success or failure of the developed system, instead success will be determined by the results produced during the evaluation stage. The aim of this project is to understand if scaling services based on their dependencies is a feasible approach and if it could potentially rival or supersede traditional methods such as the Kubernetes HPA, if the developed algorithm and system provide valuable insight into this question, the project has been successful.

3.7 Legal, Social and Ethical Issues

3.7.1 Legal

This project has no major legal considerations, all programming libraries and API's are open-source and free to use, as are the technologies such as Kafka. All data is either open source or generated by the author for this project specifically and as such, is free to use.

3.7.2 Social

Global warming is one of the defining contemporary social issues of the century, with governments around the world poised to introduce legislation and social programs that aim to reduce the impact humans have on the environment. Typically, factories and other conventional infrastructure are attributed a large portion of the blame with regards to carbon emission output [26]. However, general consensus is changing to acknowledge the impact server farms, data centres and general computing infrastructure have on global warming; typically by way of electricity consumption; cloud data centres are responsible for roughly 10% of global electricity consumption [10] [26].

Adjusting resource replication and availability has a direct impact on electricity consumption [26], for every extra process created on a given server, the electricity consumption will increase, as will the heat output of the physical machine which in turn need active cooling. This creates a problem that autoscalers can solve, or exacerbate. Auto-scalers that are overly-optimistic with their elasticity policy are likely to under-provision resources and cause the violation of SLA's; whilst autoscalers that over-provision resources will lead to an unnecessary increase in electricity consumption and heat output. Balancing these two demands should be a major consideration in the implementation of this project.

3.7.3 Ethical

Ethically, there are no issues with this project. No other people are required to engage with this project and the system will not interact with any humans throughout normal operation besides configuration of the code. The issues relating to over-provisioning of resources and electricity usage are more social in nature, as discussed prior to this section.

3.7.4 Proffesional Issues

This project has been developed adhering to all standards set out by the British Computing Society. All external work has been accredited and all claims made are factually accurate. Code developed in this project will not interfere with any existing systems on which the code is run. There are no further professional issues identifiable with this project.

Chapter 4

System Design

Our proposed system is Scale Eye, a hybrid, distributed, scalable and fault-tolerant autoscaler that utilizes service relationships in order to inform scaling decisions in microservice applications. Scale Eye continuously monitors a given microservice application and its constituent parts, watching for metric anomalies. Scaling triggers are fully customisable and adaptable to different scenarios. The system will be event-driven and reactive, it will detect and react to specific events external to the system and produce different results based on the events it detects. Services that are not currently in violation of an SLA are proactively scaled in anticipation of load that is likely to cascade to them.

Scale Eye is composed of two distinct components, Watcher and Scaling Engine. These two components operate separately and independently of each other. Initially, Scale Eye was designed as a monolithic system which contained all required functionality in a single, deployable executable. After extensive research and design proposals, a system designed with the principles of a microservice architecture seemed most appropriate.

Scale Eye logically separates into two distinct components, each component fulfills a separate function. As such, extending, modifying and removing just one of these components is a possible future requirement, and deploying the two components separately made sense from a maintenance perspective. Given that Scale Eye is a microservice application, replicas of each component can be increased to ensure SLA violations are detected and serviced in good time; in addition to this, each component requires very different libraries and technologies, this requirement lends itself to the microservice architecture [2]. A monolithic design would alleviate some of the performance bottlenecks of Scale Eye, communication latency is the most significant bottleneck in the autoscaler. However, the primary bottleneck in the overall response time with respect to an SLA violation will be the actual act of the orchestration system deploying a new instance of an application; as such, Scale Eye's increased response time as a monolithic application would likely be insignificant.

An additional motivation for designing Scale Eye as a microservice system in addition to the inherent fault-tolerance, adaptability and simpler design is that the data generated and produced by Watcher may be useful in other contexts. Once the data is produced, it could be used as an extension to Scale Eye in the future that provides insights into SLA violations or utilises past data to predict future SLA violations.

4.1 System Architecture

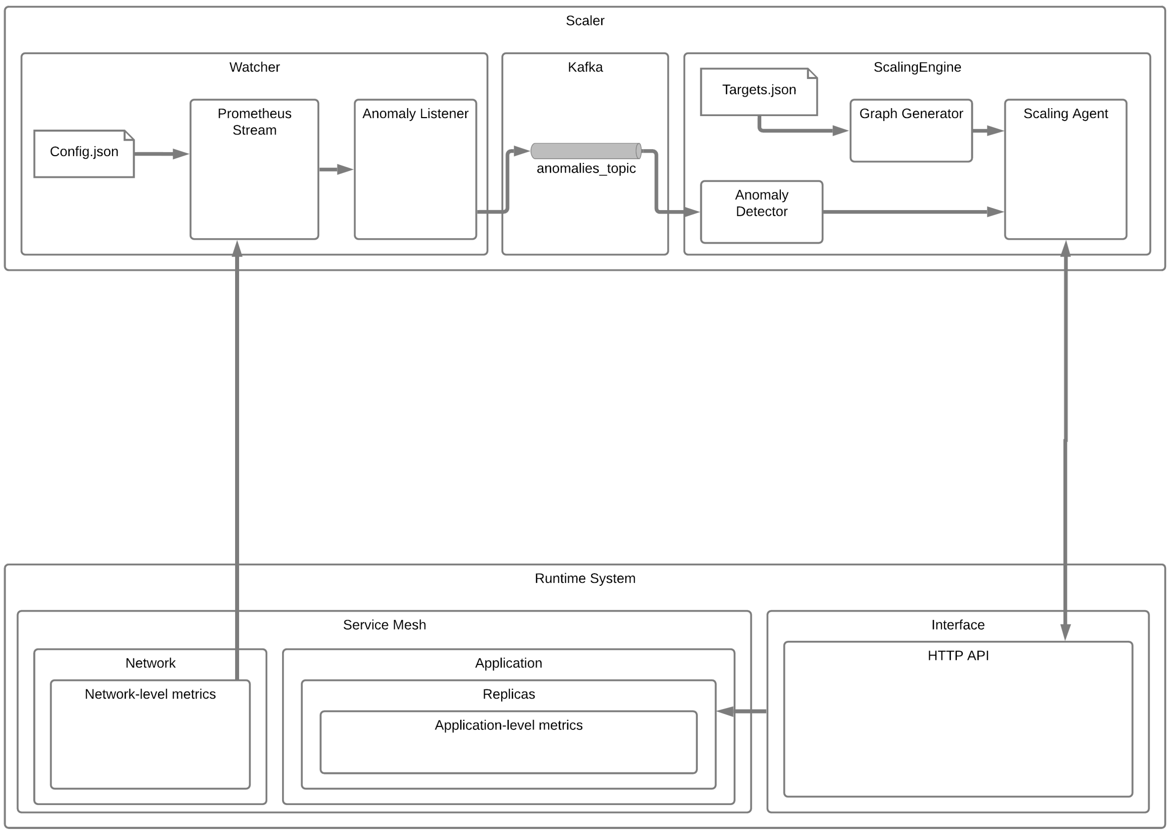


Figure 4.1: General System Architecture

Figure 4.1 illustrates the overall system architecture, demonstrating how the two distinct components will interact with each other and how each will interface with systems that are external to Scale Eye. As shown in Figure 4.1, a pub-sub messaging system will be used for all communication between Watcher and Scaling Engine. Direct socket communication could be used in order to facilitate communication between components, but this would require complex configuration and additional error handling in code. Additionally, designing the components as web services that communicate using HTTP requests would be possible, but this would also require extensive error handling, data serialisation and would also require a large amount of work to ensure response times were adequate. There are many pub-sub messaging systems that ensure communication is consistent, fault-tolerant and well managed; this led to the decision of using a pub-sub system over the additional methods listed.

Due to the decoupled nature of Scale Eye, components can be scaled in or out, modified, added to and otherwise adapted without other components needing to change, providing message schemas are adhered to. There could in theory be multiple instances of Watcher, all with slightly different implementation details in order to query different metrics stores that all produce to the same Kafka topic for Scaling Engine to consume.

4.1.1 Watcher

```
Data: target: ServiceName, function: String, query: String, Threshold: Int
queryValid ← queryValidator.validate(query);
if queryValid equals True then
    queryResult ← metricAPI.getResult(query);
    if function equals ScaleOut then
        if threshold < queryResult then
            action ← "out";
            anomaly ← KafkaMessageGenerator.generate(target, threshold, queryResult,
              action);
            produceKafkaMessage(anomaly);
        else
            pass
        end
    else
        if queryResult < threshold then
            action ← "in";
            anomaly ← KafkaMessageGenerator.generate(target, threshold, queryResult,
              action);
            produceKafkaMessage(anomaly);
        else
            pass
        end
    end
else
    log error
end
```

Algorithm 1: Anomaly Detection Algorithm

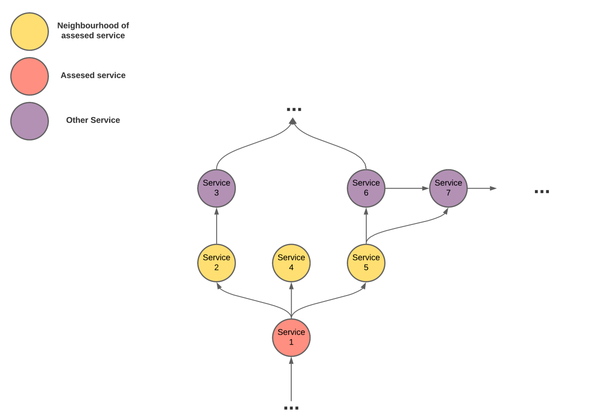
Watcher is the component responsible for detecting events that need to be responded to by Scale Eye, it is also where the elasticity policy for the autoscaler will be determined, based on the derivations made from user-provided target definitions and the configuration file provided for Watcher. The target definitions are essentially the rules the autoscaler will use to determine whether to scale a microservice. Watcher continuously queries a metric store or other time-series database with a set of predefined targets provided by the user. These targets contain query strings, application information and so forth, each target corresponds to a single microservice in the application(s) being watched. Watcher's principle component is the Anomaly Listener, this component utilizes the target data provided for each microservice along with the results from the metrics store in order to determine if a microservice is in breach of an SLA. When a target is provided in the configuration files for Watcher, a query, threshold and action to take are provided; if the result of the query exceeds the threshold, an anomaly message with the action to take is produced to the pub-sub system. Algorithm 1 demonstrates the pseudocode for the Anomaly Listener component.

4.1.2 Scaling Engine

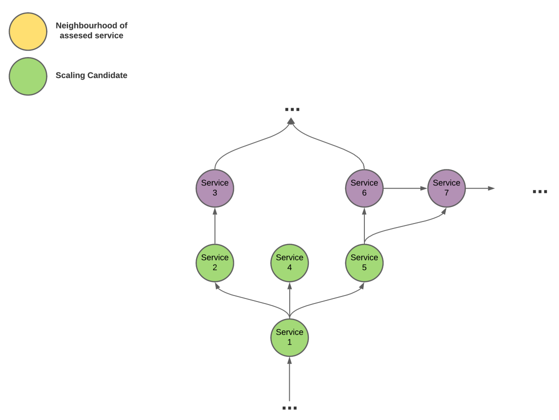
```
Data: function: ScaleOut || ScaleIn, targets: Set[ServiceName]
Result: None
for s in targets do
    currentReplicas ← api.getReplicas(s);
    scaleTargets ← graph.getDependencies(s);
    if function equals scaleOut then
        maxReplicas ← config.getMaxReplicas(s);
        if currentReplicas < maxReplicas then
            newReplicas ← currentReplicas + 1;
            deployment ← api.getDeployment(s);
            newDeployment ← deployment.replicas ← newReplicas;
            api.updateDeployment(newDeployment);
        else
            log warning
        end
    else
        minReplicas ← config.getMinReplicas(s);
        if minReplicas < currentReplicas then
            newReplicas ← currentReplicas - 1;
            deployment ← api.getDeployment(s);
            newDeployment ← deployment.replicas ← newReplicas;
            api.updateDeployment(newDeployment);
        else
            log warning
        end
    end
end
```

Algorithm 2: Dependent Scaling Algorithm

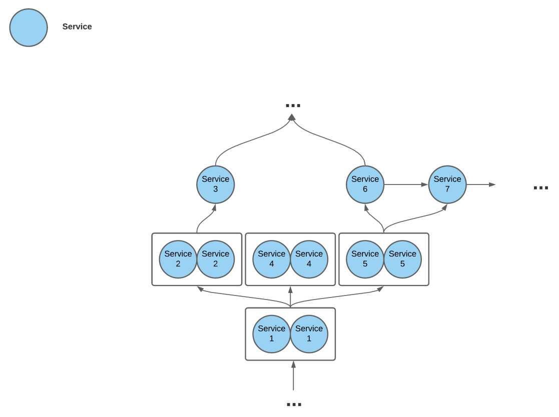
Scaling Engine is the component of Scale Eye that is responsible for reacting to messages produced by Watcher. Scaling Engine reads continuously from the pub-sub system, and uses the data from these messages in order to inform scaling decisions. Scaling Engine utilizes service definitions provided by the user that describe the different microservices being watched by Scale Eye including the maximum and minimum number of replicas and more importantly, the different relations to other services as demonstrated in Algorithm 2; if a service consistently calls another service, this should be highlighted in the dependencies definition provided by the user.



(a) Service 1 violates SLA, Calculate neighbourhood



(b) Scaling candidates marked



(c) Scaling occurs

Figure 4.2: The process of calculating scaling candidates and scaling out

The service definitions provided are used in the Graph Generator component of Scaling Engine, this represents a graph where every service is a node and edges represent a dependency on another service. When a message is consumed from Kafka, Scaling Engine will infer the dependent nodes in the graph for the service that the message is considering. With these services calculated, Scaling Engine will then attempt to scale all of these service (including the original service) in or out depending on the function specified in the message. This will all be completed without breaching the maximum or minimum replicas for each service. Figure 4.2 illustrates the general process of scaling a service and its dependent services, if service one is in violation of an SLA, Watcher will produce a message that will consequently be consumed by Scaling Engine. Scaling Engine will then use the devised algorithms to calculate the neighbourhood, mark them for scaling (adding them to the queue of scaling candidates) and then scale all of the services.

In terms of how services are selected for scaling, any service that is a dependency of service in breach of an SLA detected by Watcher will be scaled by one. Bayesian methods were considered to inform the services that should be scaled [15], but in order to keep this system simple and focused on the theory being described in this paper, they will not be used in Scaling Engine. The proposed method of scaling is primitive, but should be sufficient to see improvements in performance as this is the same method of scaling used by the Kubernetes HPA.

Chapter 5

System Implementation

5.1 Choice of Technology

Scale Eye will be developed in Scala. Scala is a modern language that runs on the JVM and allows for both object oriented and functional programming [27]. Scala has been chosen due to the fact it is mature, fast, and running on the JVM allows for easy bundling, containerisation and usage of the application components. Python and Java would have been equally viable, but Scala has been chosen due to the libraries available and easy integration with some of the technologies needed as well as its speed advantage over Python. All code is written in a functional style using libraries such as Cats which provides many quality of life improvements taken from Haskell [28] and FS2 that allows us to easily define our applications as a stream [29]. Kafka "is an open-source distributed event streaming platform" [30], messages can be rapidly produced to and consumed from Kafka topics and is the technology of choice for the pub-sub system. Kafka is a distributed application and is fault-tolerant [30]; moreover, it supports quality controls such as schema registry and schema enforcement.

5.2 General Code Design

In general, the code written for Scale Eye has been written in a functional style. Functional Programming is a software development pattern which places more emphasis on functions and immutability [31] in place of classes and representing real-world objects as is often found in Object Oriented code. Functional code is often easier to modularise and test due to the usage of higher-order functions, well-defined types and data structures [31].

Exception handling has been the error-handling method of choice in object oriented code and is a widely accepted method by which to handle unexpected input or computation in a system [32]. In Functional code, where asynchinocity and concurrency are prevelant, exceptions do not provide the most value in terms of handling errors [32]; instead, Monadic error handling with Monads such as Either, Option and Try will be used to handle unexpected or undesireable input and behaviour. This will prevent the system from requiring the context often found with exception-based error handling and instead handle errorneous behaviour and input gracefuly, treating the error types as types in their own right. The only case where exceptions will be used or thrown are in situations where the code for the system must halt immediately, such as due to JVM memory access exceptions.

5.3 Watcher Implementation

```
kafka-config {
  bootstrap-server = "192.168.86.168:9092"
  consumer-group = "watcher_v1"
  anomaly-topic = "anomalies_v3"
}
target-definitions {
  source = "targets/mainTargets.json"
}
prometheus-config {
  host = "192.168.86.52:30003"
  api-endpoint = "api/v1/query"
}
http-config {
  max-concurrent-requests = 50
}
application-metric-processing-config {
  stream-sleep-time = 240
  stream-parallelism-max = 150
}
```

Listing 5.1: Example Watcher Config

When Watcher first starts, a configuration file is loaded and parsed which contains values that will be used to configure the parallelism, timeout periods, Kafka connection details and more. These values will impact how many metric queries are executed in parallel by Watcher as well as the period of time between metric queries for each target. The value *stream-sleep-time* determines how many seconds there are between metric queries for each microservice, and impacts the elasticity policy of Scale Eye. An example of this configuration file can be found in Listing 5.1.

```
package domain

case class MetricTarget(
    name: String,
    prometheusQueryString: String,
    threshold: String,
    appName: String,
    function: String
)
```

Listing 5.2: Metric Target Case Class

Once configuration has been loaded and passed to the relevant components of the application, metric targets are loaded and passed to the stream component of the application. Listing 5.2 demonstrates a MetricTarget case class, this is used to represent each microservice being watched in the application; it contains data such as the microservice name, the metric query string (tailored for Prometheus in this case), the threshold and the action to take if the metric breaches the threshold specified. One microservice may have more than one target definition, one for scaling in and one for scaling out, for example. A sequence of these targets are generated by reading the relevant definition file provided by the user. See Appendix A for an example target definition file.

```scala
private[stream] def prometheusStream(
  watchList: Seq[MetricTarget]) =
KafkaProducer.stream(producerSettings).flatMap { producer =>
  (Stream.awakeDelay[IO](streamConfig.streamSleepTime.seconds) >>
    Stream
      .emits(watchList)
      .covary[IO]
      .parEvalMapUnordered(streamConfig.streamParallelismMax)(retrieveMetrics)
      .parEvalMapUnordered(streamConfig.streamParallelismMax)(detectAnomalies)
      .unNone
      .map(generateKafkaMessage)
      .parEvalMapUnordered(streamConfig.streamParallelismMax) { record =>
        producer.produce(record).flatten
      }).repeat
}
```

Listing 5.3: Watcher Stream

The fs2 stream illustrated in Listing 5.3 is the backbone of Watcher, it defines the process that every MetricTarget will go through, every target will be passed to the retrieveMetrics function which will utilise the metric-store connection details from the configuration file and the query string provided for this metric target to retrieve a singular value. This target and the retrieved metric is then passed to the detectAnomalies function, which is where the Anomaly Listener component is implemented. This function will use the retrieved data to determine if the threshold has been breached, if it has, it will signal that an anomaly has been detected. The next stage in the stream is generateKafkaMessage where for every anomaly detected, a Kafka message is created containing information such as the microservice name, the function to execute (scale-in or scale-out) and other metadata such as the time of detection. Finally, the messages that have been created are produced to the Kafka topic specified in the configuration file.

The processes outlined will be repeated indefinitely after the function runForever is called at the root of the application until either a fatal error occurs or the application is purposefully killed. The time between stream runs is determined in the configuration file. The messages produced by Watcher will be consumed down-stream by other components such as Scaling Engine, this implementation of Watcher is specific to Prometheus as this is the metric store chosen for this project, but this could easily be adapted to any other metric store or data source, and as specified earlier in this report, multiple distinct Watcher implementations could be working in unison to produce anomaly messages.

5.4 Scaling Engine Implementation

```
stream-config {
  bootstrap-server = "192.168.86.168:9092"
  consumer-group = "engine_v2"
  topic = "anomalies_v3"
  stream-sleep-time = 5
  stream-parallelism-max = 20
}
service-definitions {
  path = "definitions/services.json"
}
http-config {
  max-concurrent-requests = 10
}
# Not loaded by typesafe config. Read by Akka, disables Skuber logs.
akka {
  loglevel = "OFF"
}
```

Listing 5.4: Example Watcher Config

A configuration file is provided to Scale Engine that determines how many anomalies may be processed concurrently, the number of HTTP requests that can be sent in a given second and so forth. This is standard configuration found in many applications. An example of this file can be found in Listing 5.4

Once configuration has been loaded for Scale Engine, service definitions are loaded and processed. These definitions are specified in a file provided by the user, every microservice in the application should be specified in this file with data such as maximum and minimum replicas for this service and more importantly, the different services it depends on the complete a request. These definitions are loaded by Scale Engine in order to understand the different microservices it may scale whilst running. See Appendix B for an example service definition file.

```
class ServiceDependencyGraph(
  relationships: Map[String, Seq[TargetDependency]]
) {
  def inferTargets(serviceName: String): Seq[TargetDependency] =
    relationships.getOrElse(serviceName, Seq.empty[TargetDependency])
}

object ServiceDependencyGraph {
  private val log: Logger = LoggerFactory.getLogger(getClass.getSimpleName)
  def apply(
    serviceDefinitions: Seq[ServiceDefinition],
    serviceMaxReplicaMap: Map[String, Int],
    serviceMinReplicaMap: Map[String, Int]
  ): ServiceDependencyGraph = {
    val relationships: Map[String, Seq[TargetDependency]] =
      mapServicesToDependencies(serviceDefinitions, serviceMaxReplicaMap,
          serviceMinReplicaMap)

    new ServiceDependencyGraph(relationships)
  }

  def mapServicesToDependencies(
    serviceDefinitions: Seq[ServiceDefinition],
    serviceMaxReplicaMap: Map[String, Int],
    serviceMinReplicaMap: Map[String, Int]
  ): Map[String, Seq[TargetDependency]] =
    serviceDefinitions.map { service =>
      service.serviceName -> service.dependencies.map { dependency =>
        val maxReplicas = serviceMaxReplicaMap.get(dependency.serviceName) match {
          case Some(value) => value
          case None =>
            log.error(
              s"Could not retrieve maximum replicas for ${dependency.serviceName}.
                  Defaulting to 1"
            )
            1
        }

        val minReplicas = serviceMinReplicaMap.get(dependency.serviceName) match {
          case Some(value) => value
          case None =>
            log.error(
              s"Could not retrieve minimum replicas for ${dependency.serviceName}.
                  Defaulting to 1"
            )
            1
        }

        TargetDependency(dependency.serviceName, maxReplicas, minReplicas)
      }
    }.toMap
}
```

Listing 5.5: Graph Generator Component

When the service definitions are loaded, a logical graph is constructed using maps, every microservice will be a key in three maps, one for the maximum replicas, one for minimum replicas and one for dependent services; Listing 5.5 illustrates this. These maps form the basis for understanding which services to scale later on in the application.

Once the graph has been generated, the backbone of the application, an fs2 stream can be started. This stream takes the graph and context data such as replica maps and then begins consuming from Kafka. Scale Engine will consume messages as soon as they appear in Kafka.

When a message is read is will be processed using a selection of different functions, just like in Watcher. Firstly, the message is processed and validated. Once the message is validated, scaling candidates are created by using the graph to infer the dependent services of the microservice specified in the message. Once all the services are retrieved scaling candidates are created by obtaining the maximum and minimum replicas for each candidate, a request to Kubernetes is the executed which retrieves the deployment details of this microservice. The candidate and its associated deployment data is then passed to the Scaling Agent component of the application that determines if the service should be scaled based on the maximum and minimum replicas, whether the deployment is valid and so forth. Even if the subject of the original message read from Kafka is not scaled due to constraints such as maximum replicas, the dependent services may still be scaled; they are treated equally.

Scaling occurs by modifying the retrieved Kubernetes deployment details and updating it using the HTTP API. Kubernetes will receive this deployment modification and adjust the number of pods accordingly. Cluster scaling which actively adjusts the number and size of VM's that containers run is an alternative method of adapting to changes in load [33]. The focus on this project is on scaling the replicas of a given application, however.

Chapter 6

Evaluation

In order to determine the feasibility of the scaling algorithm and the implemented system, a suitable microservice application will be deployed to an instance of Kubernetes. With this application deployed, an array of different loads will be used that mimic real-world usage scenarios. The results of the load testing will be used to evaluate the efficacy of the developed system in contrast to the application being deployed with no autoscaler and the application being deployed with the Kubernetes HPA.

6.1 Benchmark Application

The Google Cloud Platform Online Boutique (GCPOB) is a production-like microservice application composed of ten individual services [34], the application represents a typical e-commerce application. Figure 6.1 demonstrates the overall architecture of GCPOB. The application source code utilises five different programming languages and many frameworks, and as such represents a real-world heterogeneous application. Communication between the different microservices is completed using gRPC. Each of the different services in the online boutique are deployed as separate Kubernetes deployments and as such are all deployed in distinct Kubernetes pods.

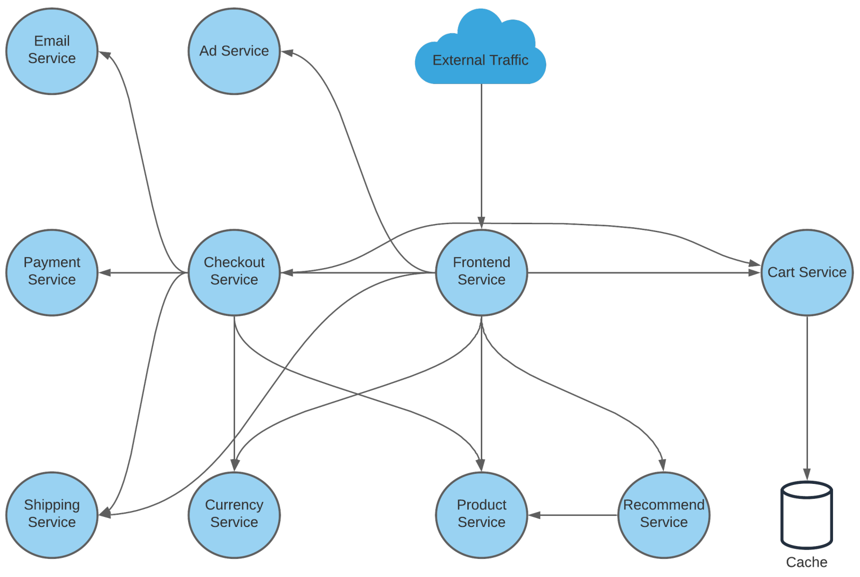


Figure 6.1: Online Boutique Architecture

6.2 Deployment, Environment and Configuration

Testing Scale Eye requires a runtime system that manages, monitors and facilitates the deployment of containerised applications and allows developers an interface to manually modify deployed resources, such as application replicas. Kubernetes [35] is the system of choice due to its prevalence in industry and well-documented API. Kubernetes does not provide sufficient metrics for monitoring the deployed applications for this project, many modern applications do report performance metrics but the validity of these metrics cannot be guaranteed as it is largely dependent on developer knowledge [15]. To supplement Kubernetes's and application metrics, the Istio service-mesh [36] has been installed to provide networking metrics including P90, P95, and P99 latency, HTTP request success rate and more between the deployed components. Istio achieves this using a high-performance proxy that captures requests and stores metrics in a time-series database [15].

The Kubernetes instance will run on Ubuntu Server 20.10 which makes use of eight gigabytes of RAM and an eight-core CPU. Minikube [37] will be used to create a Kubernetes cluster with Docker [38] being used as Minikube's backbone to create two worker and one main node; two of the eight gigabytes of RAM and two CPU cores will be allocated per node. The Istio service mesh will be deployed on top of the Kubernetes instance.

Scale Eye itself will be deployed to a different system. Scale Eye will be deployed using Docker with each component in a distinct Docker container. The system used to deploy the autoscaler is comprised of an six core CPU with sixteen gigabytes of RAM.

6.2.1 Scale Eye Configurations

Scale Eye will be deployed with two distinct configurations, encapsulating two distinct approaches to scaling the individual service components. As well as the configuration below, the metric queries for each service will also be defined but remain the same for both configurations. Accurately predicting valid thresholds for metrics such as CPU usage, RAM usage and higher-level metrics such as latency is notoriously hard to predict [12]. Incorrect prediction of the correct values to trigger scales can lead to system instability and lacklustre performance of an autoscaler [12]. A best attempt has been made to provide valid and accurate metric thresholds to trigger scaling for every service, both the Watcher target definitions and Scale Engine service definitions used in this analysis can be found in Appendices B and C respectively.

Configuration One

The first configuration will be more pro-active and reactionary, metrics will be pulled every twenty seconds. P99 latency will be the metric of choice for all scaling target queries. This deployment should detect violations very quickly and react accordingly.

Configuration Two

The second configuration will be less reactionary, and will have a delay of five minutes between metric pulls; this prolonged delay should prevent anomalies from being detected more than once [15].

6.3 Assessment

Evaluating the performance of the Scale Eye will be completed by simulating real-world user load on GCPOB, the frontend component of the application will be the destination for all requests. No direct requests will be sent to any other services, but will instead be sent by the frontend as demonstrated in Figure 6.1. Stress tests will be conducted in the following scenarios:

- Without an autoscaler
- With the Kubernetes HPA
- Using Scale Eye with system configuration one
- Using Scale Eye with system configuration two

6.3.1 Test Definitions

All tests will have a random delay between each request of between five-hundred milliseconds and one second, to replicate the behaviour of real users. Users will not all be created at once, but will be added incrementally. Every test will last for sixty minutes. Prior to conducting a test, all Kubernetes deployments will be scaled to zero replicas and re-scaled back to one replica to ensure every test begins with fresh application pods for each service.

Standard Test

The Standard test will simulate the constant load of one-hundred users. These users will send one request per iteration, a single request to the homepage.

Standard Spike Test

The Standard Spike test will simulate the constant load of one-hundred users with occasional bursts of traffic. The constant users will send three requests per iteration, first visiting the homepage, then a predefined product page before finally sending a request for the checkout page. With the constant load running, every ten minutes a burst of fifty users will send a single request to the homepage each for one minute before stopping until the next burst.

Heavy Test

Heavy test will simulate the constant load of five-hundred users. These users will send three requests per iteration, first visiting the homepage, then a predefined product page before finally sending a request for the checkout page.

Heavy Spike Test

The Heavy Spike test will simulate the constant load of five-hundred users with occasional bursts of traffic. The constant users will send three requests per iteration, first visiting the homepage, then a predefined product page before finally sending a request for the checkout page. With the constant load running, every ten minutes a burst of one-hundred users will send a single request to the homepage each for one minute before stopping until the next burst.

Apache Jmeter "is a Java application designed to load test functional behavior and measure performance" [39]; Jmeter has a variety of functionality that will be used to simulate HTTP requests, report metrics such as P99 latency and stress-test the different micro-services. The load exerted by Jmeter unto the deployed application will be synthetic in nature, synthetic load is often used in the benchmarking of scaling mechanisms [12].

6.4 Analysis

Latency on a microservice is not constant, different users will experience different latency from the same request due to external factors, as such the mean latency on a service may not be sufficiently descriptive. In place of the mean latency, the ninetieth (P90), ninety-fifth (P95) and ninety-ninth (P99) percentile latencies will be considered. The ninetieth percentile latency will be a singular value that users that fall into the bottom ten percent of users (the ninetieth percentile) are likely to experience.

Whilst P90, P95 and P99 latency all provide some level of insight into application performance, P90 will provide the most detail as to the performance that most users will experience; As such, will be the most relevant in contrasting overall performance [15]. P95 and P99 are more susceptible to "jitters", whereby very extreme cases of latency will have more of an impact; only the slowest one percent of requests are likely to experience the levels of latency reported by P99 [15]. System performance in relation to latency is the primary metric of interest in this evaluation, however, system crashes will also be used to assess overall application stability over time due to the cost of restarting and re-creating Kubernetes pods. During analysis, the following metrics will be used to compare and contrast the performance of each scaling mechanism:

- Frontend P90 latency
- Frontend P95 latency
- Frontend P99 latency
- Total number of kubernetes pod crashes/forced restarts

6.4.1 Results

No Scaling Mechanism

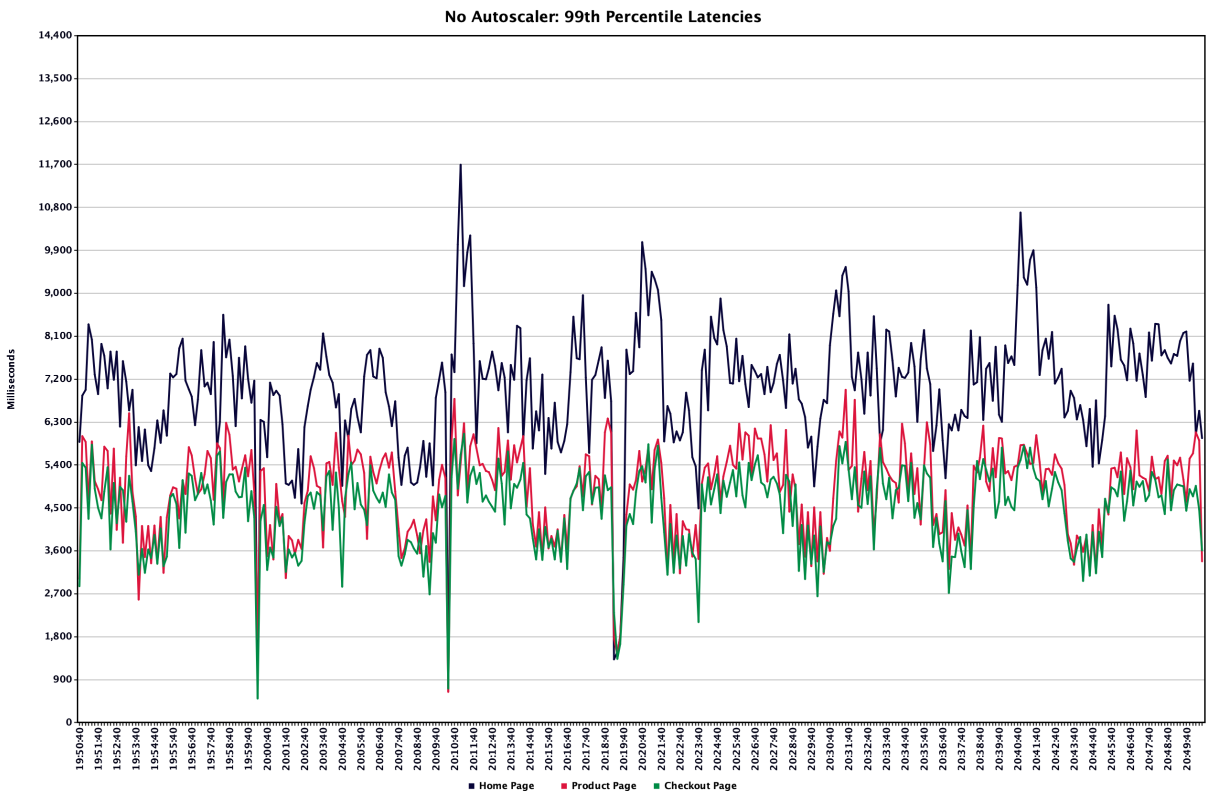


Figure 6.2: P99 during Heavy Spike for No Scaling Mechanism

The lack of an autoscaler removes the ability for the system to properly adapt to changing workloads. The application performed well during the Standard test with a consistent P90 latency of just over one thousand and two hundred milliseconds; whilst slow, this did not render the frontend unusable. The application did experience some slow-down when faced with load spikes in the Standard Spike test and had a P90 latency of roughly ten thousand milliseconds in both the Heavy and Heavy Spike tests as demonstrated in Figure 6.2 which illustrates the fact that GCPOB being deployed without a scaling mechanism does render it unusable when confronted with inconsistent load. When handling a constant load, the P90 latency reported remained consistent, but high. There were no pod crashes during any of the four tests. The results of this test show that whilst the system could not handle load well at all, it does remain stable in the sense that there were no crashes.

Horiztonal Pod Autoscaler

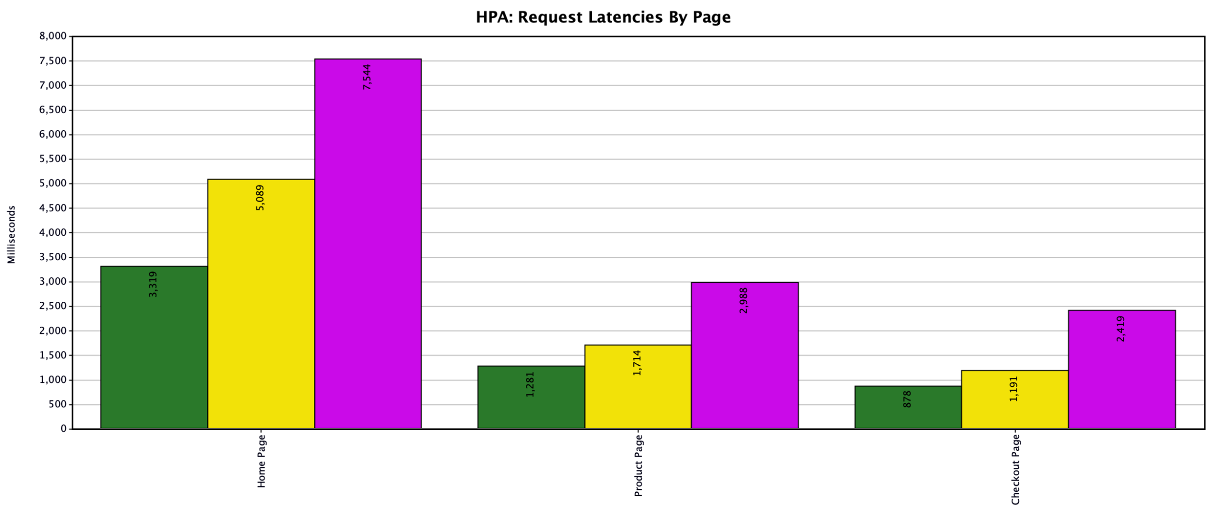


Figure 6.3: Heavy Spike Latency By Page for HPA

The Horiztonal Pod Autoscaler was used in order to ascertain the performance levels when using a traditional scaling mechanism. All deployments in the application were configured to autoscale based on CPU usage. CPU usage was chosen as it easily and readily available to Kubernetes through the in-built Kubernetes metrics server, and it is a metric commonly used in industry to scale deployments, replicas will be scaled out once CPU usage amongst all pods in a given deployment reaches 50%.

Overall, the HPA deployment saw significant improvements over no scaling mechanism in terms of P90 latency in all tests, with the exception of test one where the decrease in latency was minimal, likely due to the fact the autoscaler was not being used as the default configuration of one replicas was sufficient to service the load of one hundred concurrent users. Figure 6.3 illustrates the marked decrease in P90, P95 and P99 latencies in comparison to no scaling mechanism.

The worrying metric is the number of pod restarts, in total there were forty-one pod restarts in the Heavy Spike test. Due to the large number of restarts when using HPA the test was conducted again and similar results in terms of latency were found and a similar number of pod restarts also occurred (thirty-nine).

Scale Eye with Configuration One

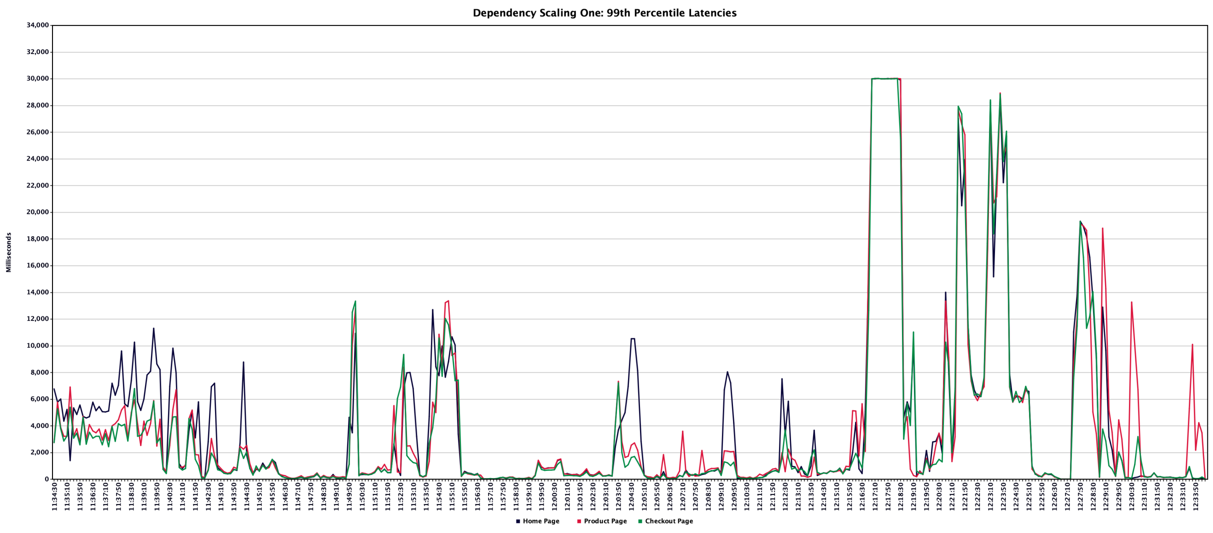


Figure 6.4: P99 Latency during HeavySpike test shows instability

Scale Eye deployed with configuration one performed inadequately in some tests, the application performed well for a period of time, but in some cases performed worse than when the application was deployed with HPA. P90 latency was reduced in both Heavy tests. P99, P95 and P90 latency was generally more stochastic during the test as seen in Figure 6.4. As well as the P99, P95 and P90 latency being unstable and unpredictable, there were fifteen pod crashes in test four; this is likely due to the frequency in which the system was changing the number of service replicas. The system was generally unstable but did have fewer pod crashes than HPA in the Heavy Spike test.

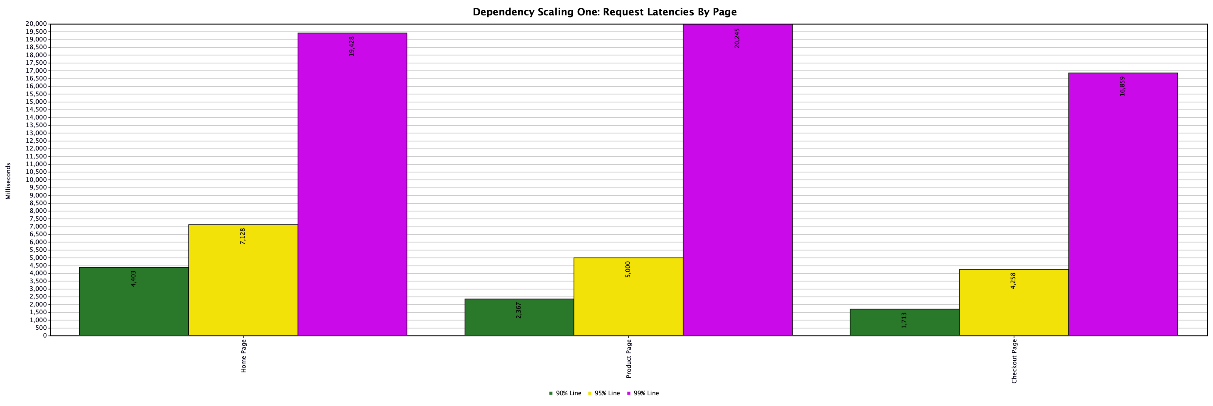


Figure 6.5: P99, P95 and P90 latency by page in The Heavy Spike test for Scale Eye Config 1

During all four tests, latency breached the SLA's defined in Watcher configs and never recovered. Figure 6.5 illustrates the system instability, with a very high P99 latency and much lower P90 latency, this was caused by the system was experiencing severe jitters and random timeouts.

Developed System with Configuration Two

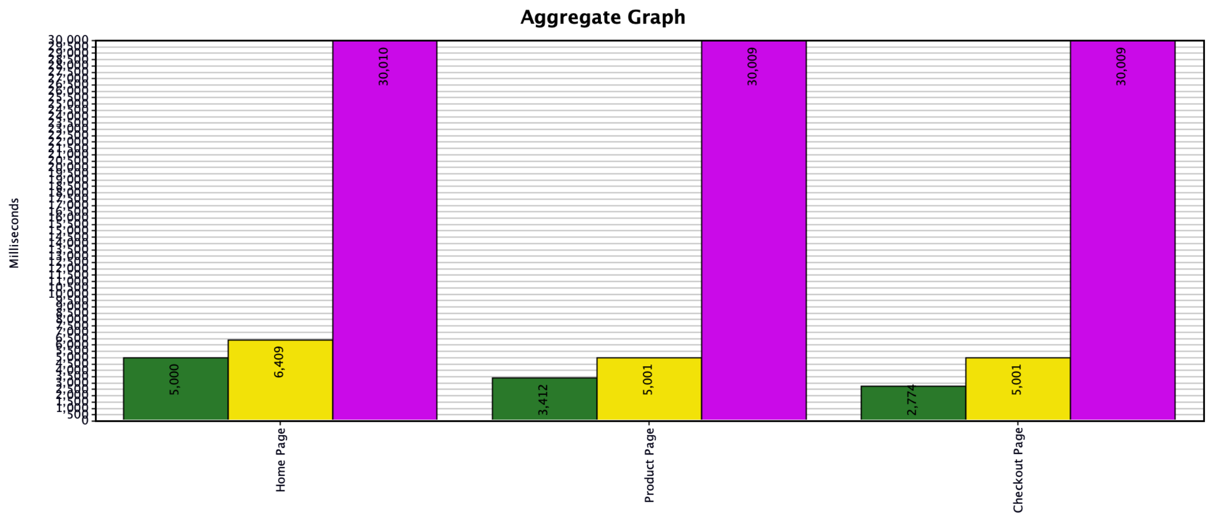


Figure 6.6: P99, P95 and P90 latency by page in The Heavy Spike test for Scale Eye Config 2

Scale Eye with configuration two performed sporadically and violated all SLA's consistently. During the Heavy Spike test, a P90 latency of five thousand milliseconds was experienced, with P99 latency reaching as high as thirty thousand milliseconds due to system crashes and instability. This shows Scale Eye could not handle the load exerted by the Heavy Spike test with configuration two. There were eighteen pod crashes during the same test.

During the Standard test, Scale Eye with configuraiton two displayed a roughly equivalent level of performance to the other three systems, with P90 latency of one-thousand, two hundred and fifty milliseconds. There were no crashed during the Standard test.

Five minute detection windows are not uncommon, and can sometimes be adequate for detecting anomalies in system load in order to trigger scaling [15]. In this case, the prolonged detection windows were not adequate and prevented the scaling mechanism from properly detecting anomalies in good time, before the system began to struggle.

6.4.2 Discussion

	No Scaling Mechanism	HPA	Scale Eye 1	Scale Eye 2
Standard	1255ms	1211ms	1272ms	1250ms
Standard Spike	1517ms	1233ms	1743ms	1627ms
Heavy	8776ms	3433ms	2858ms	9127ms
Heavy Spike	9229ms	3319ms	4403ms	5000ms

Table 6.1: P90 latencies for each scaling mechanism and test

In order to properly understand the impact of the scaling mechanisms being anaylsed, the system was first deployed without a scaling mechanism. The application performed as expected in this scenario, with a low P90 latency in the standard test where the number of concurrent users was relatively low, the load spikes in test two did present the system with further challenges, during both Heavy tests the system remained stable but with P90 latency that was very high, as demonstrated in Table 6.1. The lack of any pod crashes does show the system handled the load without any catastrophic failures. Deploying the application with the HPA presented some significant performance increases of roughly 90% in the Heavy Spike test in terms of P90 latency on the home page when compared with no scaling mechanism. The system handled many more requests before struggling, the maximum number of replicas for the front-end service was reached quickly into the test and rarely scaled in below the maximum replicas; roughly ten minutes lapsed after each spike before HPA began to scale back in again. It is also worth noting that during the Heavy and Heavy Spike tests, the P90 latency remained roughly equal which suggests that the spikes did not impact the performance of GCPOB when being deployed with HPA, this shows HPA was able to adapt to these load spikes in time. Despite the performance improvements the system was far less stable, with roughly forty pod restarts during test four, this can likely be attributed to HPA scaling in and out too frequently. It is possible some tuning of HPA parameters could increase system stability as opposed to HPA being fundamentally flawed.

Scale Eye with configuration one experienced higher latency, more stochastic behaviour and fifteen pod crashes in the Heavy Spike test. In addition to the system instability, during the Heavy Spike test, P90 latency increased by 28% in comparison the the same latency when using HPA. After analysing logs from Scale Eye and Kubernetes it appears the stochastic behaviour, over-subscription of resources and contradictory scale actions likely attributed to this increase. Kubernetes never had a chance to properly create, initialize and add a pod to the load balancer before Scale Eye requested a pod be removed or another added during spikes. For example, in two iterations of the scaler (which were twenty seconds apart), the frontend was scaled up, and then scaled back down. This sporadic behaviour would explain the large spikes in P99 latency and consistently higher P95 and P90 latency, as the load-balancer would be continuously adapting to accommodate new numbers of replicas, it was also the case that some pods would crash due to the sheer number of changes in the number of deployed replicas. Anomalies could also have been picked up in multiple scaler iterations, leading to the scaler over-provisioning a resource only to later scale-in the same resource as suggested in [15]. Whilst during the Heavy Spike test, Scale Eye with this configuration was slower, during the Heavy test Scale Eye had an improvement of 18% in terms of P90 latency when compared with HPA. This suggests that Scale Eye struggles with load spikes, as opposed to load that increases gradually. This could be due to the fact that configuration one polls the metric store every twenty seconds, which may lead to the system over-provisioning resources too quickly, only to scale the same resources back in once the spike has finished; the pods may not have all started by the time the spike had finished, due to the rapid scale out. Some tuning of the parameters and configuration for Scale Eye could likely address this.

Scale Eye with Configuration Two performed worse than HPA in every test. There were increases of 165% in the Heavy test, the system simply could not keep up with the load being exerted onto GCPOB. The incredibly high latencies were largely due to the system struggling from early in the test, and not being able to recover after being scaled up. This could be a fault in the design of GCPOB, but it is unlikely. The prolonged detection windows appear to be the issue, the scaler could not adapt quickly enough to new load, spikes in particular, and this rendered the application unusable.

Overall Scale Eye violated a number of SLA's defined in the Watcher configuration, but did perform adequately in the the Heavy test, where it outperformed HPA with configuration one. In the Standard and Standard Spike test, Scale Eye with configuration one performed almost as well as HPA, with minor increases in P90 latency that would likely fall within or just outside of a margin of error. However, in the Heavy Spike test Scale Eye struggled with both configurations. This suggests that is it with spiked loads that the autoscaler struggles. This could be remedied using a more intelligent method of determining the number of optimal replicas, using reinforcement learning or Bayesian Optimisation [15]. It is also possible that the tuning of configuration files would adequately improve performance, the determination of metric thresholds to trigger scales was arbitrarily chosen, and there is active research in how to determine these thresholds [12].

Chapter 7

Future Work

The project has been a success in terms of the defined objectives and aims. There is, however, work that could be done to further improve the project and better determine if dependent scaling is feasible in production environments.

7.1 Algorithmic Improvements

The algorithm itself works as expected, there are adaptations that could be made to make it feasible for realistic production-like workloads. Firstly, the method by which microservices are scaled is naive and unlikely to work in large, complex deployments. In place of scaling each service by a constant factor, making use of an approach such as a Bayesian Optimization [15] to determine the optimal number of replicas for a service under a given load could provide extensive performance improvements. This should also alleviate some of the issues highlighted in the evaluation such as over-subscription of resources.

Another improvement for the algorithm would be to allow for more granular definitions of relationships. Currently, the algorithm only takes into account relationships that are linear in fashion and does not consider the fact that some services depend on multiple services to varying degrees [15]. Adapting the algorithm to use new configuration that defined how loosely or tightly coupled two services are to inform scaling decisions would likely improve efficiency.

7.2 Scale Eye

Configuration is the main point of contention with Scale Eye, whilst the configuration options were sufficiently granular in some cases, it could have been extended in others. Allowing the user to specify a number of seconds to wait in order to allow a deployment to finish being modified by Kubernetes may provide a mechanism to stop anomalies being detected more than once [15]. Futhermore, the metric queries provided in configuration are not optimal, and could be tweaked to be more appropriate; determining these metric thresholds is not an easy problem [12], moving to an automated means of determining the optimal threshold using past data could be a potential solution [12].

7.3 Testing

The tests conducted in this project in order to evaluate performance were sufficient and relevant, they provided valuable insight into the feasibility of the algorithm that has been developed and the accompanying autoscaler. The tests in this project followed a pattern, they were either entirely consistent (with a degree of randomness in terms of time between requests) or consistent with timed load spikes. In the real world, load does not always adhere to a pattern [40]; testing Scale Eye in a scenario such as this would provide valuable insight into further improvements that could be made.

Utilising different infrastructure is a potential improvement in the testing process. In this project, a single server was used with modest system hardware, testing Scale Eye on a Kubernetes deployment that consisted of multiple physical nodes that could handle many more requests would be a good indicator as to how the algorithm would work in a better equipped deployment, it would also remove some of the performance degradation factors such as pod crashes.

Finally, adapting the HPA to utilize latency in place of CPU usage may provide a more fair comparison, the aim of using the HPA in this project was to provide a baseline with a commonly used autoscaler in industry using configuration that is also commonly used, but in future it would be interesting to see how HPA works when using P99 latency as its metric of choice and whether it still performs more efficiently than Scale Eye.

Chapter 8

Conclusion

Microservice architectures form the basis for many of the most widely-used and essential computer systems to date. The separation of concerns, scaling granularity and simplified codebases microservices facilitate will ensure the architectural pattern will remain prevalent for some time [2]. Improving on the ability to scale a given application such as GCPOB is essential, guaranteeing Quality of Service when load increases and ensuring costs are minimized wherever possible [2].

Relationships between microservices are inherently complex, a microservice is almost always co-dependent on another in some way [17][15], understanding these relationships and adapting a system based on the inferences that can be made from them could provide the means to a more efficient scaling mechanism, as illustrated in Scale Eye, a rule-based autoscaler. Scale Eye is a hybrid, decentralized autoscaler that implemented a small set of simple algorithms that utilised dependent scaling, scaling a microservice not only based on its own performance but on the performance of microservices that are upstream, allowing a microservice to be scaled up before load can cascade to it.

Scale Eye's performance with respect to P90 latency matches the performance of existing contemporary methods such as Kubernetes's HPA, with minor increases in latency in some test scenarios. This suggests that Scale Eye is well positioned to provide sufficient real-world performance. Average P90 latency in scenarios where load increases gradually were sufficient and matched Kubernetes's HPA. However, in scenarios where load increases gradually with additional load spikes, Scale Eye did exhibit inferior P90 latency in deployed applications when compared with HPA. In future, contrasting Scale Eye with another predictive autoscaler such as MicroScaler [15] may demonstrate if this is a pitfall of Scale Eye and hybrid autoscalers, or a strong point of HPA. The different Scale Eye parameters impact the performance of the watched applications significantly, as demonstrated in the test scenarios; further developing the different configurations is also likely to bring performance more in-line with HPA in spike scenarios.

Scale Eye has shown that hybrid, dependent scaling has the potential to be equally as efficient as traditional scaling implemented by systems such as the Kubernetes Horiztonal Pod Autoscaler. During evaluation, pitfalls of Scale Eye and the devised algorithm were highlighted, namely the naive approach to calculating optimal replicas of a microservice and detecting anomalies more than once. These issues are not specific to Scale Eye, but could be alleviated and rectified in the future using methods such as Bayesian Optimisation or reinforcement learning to assess the optimal number of replicas for a service [15]. Hybrid elasticity policies are not common and as such, are not yet well researched [12]; Scale Eye demonstrates hybrid methods do have the potential to work in real-world scenarios and are comparable to existing autoscaler implementations such as Kubernetes's HPA and MicroScaler [15].

Dependent scaling has the ability to provide a resilient, efficient means of scaling a system with complex relationships, using microservice relationships as a key factor in scaling decisions. Whilst this project has shown it is possible to achieve an acceptable level of performance using dependent scaling on a system such as GCPOB with minor adaptations, it is still unclear how this would perform on a system with very few, or very simple microservice interactions.

8.1 Self-Appraisal

This project is something I found incredibly interesting, it allowed me to combine the fundamental theoretical concepts taught throughout my degree with the engineering experience I gained whilst on my fifteen month placement at Expedia Group. The subject area is not something I had extensive experience with, besides the standard experience many software engineers would have.

The project timeline was largely adhered to, there were no major setbacks in terms of timeline which I do principally attribute to the extra time I have to work on the project due to the COVID-19 pandemic. In normal circumstances I do see certain parts of the project that would been likely to overrun, namely the background research portion of the project.

The biggest challenge I faced was not to do with engineering, design or implementation, it was preparation. This is the first large individual project I have undertaken, and I am used to being given a brief or set of work to complete where the research is largely completed for me; even at work, the product teams would prepare work for the engineers. I noticed at various points of my implementation that I had overlooked very significant details, such as by which mechanism I would scale deployments, how would metrics be retrieved from applications and so forth. I learned the true value of preparing research and properly understanding the domain before beginning implementation. Not rushing this research phase would also have been very useful, as I looked over minor details in some papers that would have been very useful in my project.

Overall, I do believe this project has been a success. I ended the project with what I believe to be a good understanding of the relevant literature, as well as the engineering practices used to implement the theory outlined in the aforementioned literature. All deliverables have been produced, the algorithms and analysis are in this report, and the codebase is provided as a Git repository; the code is to a standard I would be happy to publish in industry. The analysis phase highlighted some of the shortcomings of my design, and outline why I believe this approach is not optimal in its current form. During the undertaking of this project I have learned the value in understanding the domain and having a thorough plan before beginning implementation; I also see the value in properly defining an expected timeline and accounting for potential setbacks.

References

[1] Dubey, A Roy, N and Gokhale, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. Washington, DC, USA, 2011. IEEE.

[2] Dragoni, N, Giallorenzo, S, Lafuente, A, Mazzara, M, Montesi, F, Mustafin, R, and Safina, L. *Microservices: Yesterday, Today, and Tomorrow.* Springer International Publishing, 2017.

[3] Stein, E. A Calculus of Communicating Systems, Lecture Notes in Computer Science 92. *ZAMM-Journal of Applied Mathematics and Mechanics.* 1982, **62**(9), p499.

[4] Ma, S, Fan, C, Chuang, Y, Lee, W, Lee, S, and Hsueh, N. Using Service Dependency Graph to Analyze and Test Microservices. *2018 IEEE 42nd Annual Computer Software and Applications Conference, 23-27 July 2018, Tokyo, Japan.* [Online]. Tokyo, Japan: IEEE, 2018, pp 81-86. Available from: https://ieeexplore.ieee.org/document/8377834.

[5] Jamshidi, P, Pahl, C, Mendonça, NC, Lewis, J, and Tilkov, S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software.* 2018, **35**(3), pp 24-35.

[6] Kang, H, Le, M, and Tao, S. Container and Microservice Driven Design for Cloud Infrastructure DevOps. *2016 IEEE International Conference on Cloud Engineering, 4-8 April 2016, Berlin, Germany.* [Online]. Berling, Germany: IEEE, 2016, pp 202-211. Available from: https://ieeexplore.ieee.org/document/7484185.

[7] Rattihalli, G, Govindaraju, M, Lu, H, and Tiwari, D. Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes. *IEEE 12th International Conference on Cloud Computing, 4-9 July 2011, Washington, USA.* [Online]. Washington, USA: IEEE, 2011, pp 716-723 [01 March 2021]. Available from: https://ieeexplore.ieee.org/document/6008775.

[8] Idziorek, J. Discrete event simulation model for analysis of horizontal scaling in the cloud computing model. *Proceedings of the 2010 Winter Simulation Conference, 5-8 December 2010, Baltimore, USA.* [Online]. Baltimore, USA: IEEE, 2010, pp 3004-3014. Available from: https://ieeexplore.ieee.org/document/5678994.

[9] Rzadca, K, Findeisen, P, Swiderski, J, Zych, P, Broniek, P, Kusmierek, J, Nowak, P, Strack, B, Witusowski, P, Hand, S, and Wilkes, J. Autopilot: Workload Autoscaling at Google. *Proceedings of the Fifteenth European Conference on Computer Systems.* 2020, **16**(1), pp 1-16.

[10] Scheepers, MJ. Virtualization and containerization of application infrastructure: A comparison. *21st twente student conference on IT.* 2014, **21**(1), pp 1-7.

[11] Dua, R, Raja, AR, and Kakadia, D. Virtualization vs Containerization to Support PaaS. *2014 IEEE International Conference on Cloud Engineering, 11-14 March 2014, Boston,*

USA. [Online] Boston, USA: IEEE, 2014, pp 610-614. Available from: https://ieeexplore.ieee.org/document/6903537.

[12] Al-Dhuraibi, Y, Paraiso, F, Djarallah, N, and Merle, P. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing.* 2018, **11**(2), pp 430-447.

[13] Zheng, T, Zheng, X, Zhang, Y, Deng, Y, Dong, E, Zhang, R, and Liu, X. SmartVM: a SLA-aware microservice deployment framework. *World Wide Web.* 2019, **22**(1), pp 275-293.

[14] Toffetti, G, Brunner, S, Blöchlinger, M, Dudouet, F, and Edmonds, A. An Architecture for Self-Managing Microservices. *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, 01 April 2015, Bordeaux, France.* [Online]. Bordeaux, France: ACM, 2015, pp 19-24 [01 March 2021] Available from: https://dl.acm.org/doi/10.1145/2747470.2747474.

[15] Yu, G, Chen, P, and Zheng, Z. Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. *2019 IEEE International Conference on Web Services, 8-13 July 2019, Milan, Italy.* [Online]. Milan, Italy: IEEE, 2019, pp 68-75. available from: https://ieeexplore.ieee.org/document/8818401.

[16] Ghanbari, H, Simmons, B, Litoiu, M, and Iszlai, G. Exploring Alternative Approaches to Implement an Elasticity Policy. *IEEE 12th International Conference on Cloud Computing, 4-9 July 2011, Washington, USA.* [Online]. Washington, USA: IEEE, 2011, pp 716-723 [01 March 2021]. Available from: https://ieeexplore.ieee.org/document/6008775.

[17] Gan, Y, Zhang, Y, Cheng, D, Shetty, A, Rathi, P, Katarki, N, and et al. *An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud Edge Systems.* Association for Computing Machinery, New York, NY, USA, 2019.

[18] Novak, JK, Kasera, SK, and Stutsman, R. Cloud Functions for Fast and Robust Resource Auto-Scaling. *2019 11th International Conference on Communication Systems Networks, 7-11 January 2019, Bengaluru, India.* [Online]. Bengaluru, India: IEEE, 2019, pp 133-140. Available from: https://ieeexplore.ieee.org/document/8711058.

[19] Hussein, AA. A survey on vertical and horizontal scaling platforms for big data analytics. *International Journal of Integrated Engineering.* 2019, **11**(6), pp 138-150.

[20] Qingye, J, Lee, YC, and Zomaya, AY. The Limit of Horizontal Scaling in Public Clouds. *ACM Trans. Model. Perform. Eval. Comput. Syst.*. 2020, **5**(1), pp 1-16.

[21] Wadia, Y, Gaonkar, R, and Namjoshi, J. Portable Autoscaler for Managing Multi-cloud Elasticity. *2013 International Conference on Cloud Ubiquitous Computing Emerging Technologies, 15-16 November 2013, Pune, India.* [Online]. Pune, India: IEEE, 2013, pp 48-51 [01 March 2021]. Available from: https://ieeexplore.ieee.org/document/6701474.

[22] Cohen, D, Lindvall, M, and Costa, P. An introduction to agile methods. *Advances in Computers.* 2004, **62**(3), pp 1-66.

[23] Despa, ML. Comparative study on software development methodologies. *Database systems journal.* 2004, **5**(3), pp 37-56.

[24] Jones, TS and Richey, RC. Rapid prototyping methodology in action: A developmental study. *Educational Technology Research and Development.* 2000, **48**(2), pp 68-80.

[25] McCormick, M. *Waterfall vs. Agile methodology.* [Online]. 2012. [01 December 2020]. Available from: http://mccormickpcs.com.

[26] Fernández-Cerero, D, Fernández-Montes, A, and Jakóbik, A. Limiting Global Warming by Improving Data-Centre Software. *IEEE Access.* 2020, **8**(1), pp 44048-44062.

[27] EPFL. *Scala Lang.* [Online]. 2020. [01 December 2020]. Available from: `https://www.scala-lang.org`.

[28] TypeLevel. *Cats.* [Online]. 2020. [01 December 2020]. Available from: `https://typelevel.org/cats/`.

[29] Typelevel. *FS2.* [Online]. 2020. [01 December 2020]. Available from: `https://fs2.io/index.html`.

[30] Apache. *Apache Kafka.* [Online]. 2020. [01 December 2020]. Available from: `https://kafka.apache.org`.

[31] Hughes, J. Why Functional Programming Matters. *The Computer Journal.* 1989, **32**(2), pp 98-107.

[32] Nagy, G. Comparing Software Complexity of Monadic Error Handling and Using Exceptions. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, 2024 May 2019, Optatija, Croatia.* [Online]. Optatija, Croatia: IEEE, 2018, pp 1570-1580. Available from: https://ieeexplore.ieee.org/document/8757213.

[33] Ayalew, TM, Johan, T, Elmroth, E, and Pierre, G. An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud. *CloudCom 2020 - 12th IEEE International Conference on Cloud Computing Technology and Science.* 2020, **11**(2), pp 1-9.

[34] Google. *GCP Online Boutique.* [Online]. 2020. [01 January 2020]. Available from: `https://github.com/GoogleCloudPlatform/microservices-demo`.

[35] Kubernetes.io. *Kubernetes.* [Online]. 2020. [01 December 2020]. Available from: `https://kubernetes.io`.

[36] Istio. *Istio Service Mesh.* [Online]. 2020. [01 February 2020]. Available from: `https://istio.io`.

[37] Minikube. *Minikube.* [Online]. 2020. [08 February 2020]. Available from: `https://minikube.sigs.k8s.io/docs/`.

[38] Docker. *Docker Containers.* [Online]. 2020. [01 February 2020]. Available from: `https://www.docker.com/resources/what-container`.

[39] Apache. *JMeter*. [Online]. 2020. [12 February 2020]. Available from: `https://jmeter.apache.org`.

[40] Yan, M, Liang, X, Lu, Z, Wu. J, and Zhang. W. HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM. *Applied Soft Computing*. 2021, **105**(1), pp 107-216.

Appendices

Appendix A

External Material

Code Repository URL: https://github.com/Sicarius154/ScaleEye
Code Build and Deployment Guide: https://github.com/Sicarius154/ScaleEye

Appendix B

Watcher Target Definition

```
{
  "proposedName": "Frontend P99 Latency Scale Out",
  "proposedPrometheusQueryString": "label_join((histogram_quantile(0.99,
      sum(rate(istio_request_duration_milliseconds_bucket{destination_workload=\"frontend\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)) / 1000) or histogram_quantile(0.99,
      sum(rate(istio_request_duration_seconds_bucket{destination_workload=\"frontend\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)), \"destination_workload_var\", \".\",
      \"destination_workload\", \"destination_workload_namespace\")",
  "proposedThreshold": "3.0",
  "description": "The 99th percentile of latency on the frontend",
  "proposedAppName": "frontend",
  "function": "out"
}
{
  "proposedName": "Frontend P99 Latency Scale In",
  "proposedPrometheusQueryString": "label_join((histogram_quantile(0.99,
      sum(rate(istio_request_duration_milliseconds_bucket{destination_workload=\"frontend\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)) / 1000) or histogram_quantile(0.99,
      sum(rate(istio_request_duration_seconds_bucket{destination_workload=\"frontend\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)), \"destination_workload_var\", \".\",
      \"destination_workload\", \"destination_workload_namespace\")",
  "proposedThreshold": "0.9",
  "description": "The 99th percentile of latency on the frontend",
  "proposedAppName": "frontend",
  "function": "in"
}
{
  "proposedName": "Email P99 Latency Scale Out",
  "proposedPrometheusQueryString": "label_join((histogram_quantile(0.99,
      sum(rate(istio_request_duration_milliseconds_bucket{destination_workload=\"emailservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)) / 1000) or histogram_quantile(0.99,
      sum(rate(istio_request_duration_seconds_bucket{destination_workload=\"emailservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)), \"destination_workload_var\", \".\",
      \"destination_workload\", \"destination_workload_namespace\")",
  "proposedThreshold": "3.0",
  "description": "The 99th percentile of latency on the email service",
  "proposedAppName": "emailservice",
  "function": "out"
}
{
  "proposedName": "Email P99 Latency Scale In",
  "proposedPrometheusQueryString": "label_join((histogram_quantile(0.99,
      sum(rate(istio_request_duration_milliseconds_bucket{destination_workload=\"emailservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)) / 1000) or histogram_quantile(0.99,
      sum(rate(istio_request_duration_seconds_bucket{destination_workload=\"emailservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)), \"destination_workload_var\", \".\",
      \"destination_workload\", \"destination_workload_namespace\")",
  "proposedThreshold": "0.9",
  "description": "The 99th percentile of latency on the email service",
  "proposedAppName": "emailservice",
  "function": "in"
}
{
  "proposedName": "Payment P99 Latency Scale Out",
  "proposedPrometheusQueryString": "label_join((histogram_quantile(0.99,
      sum(rate(istio_request_duration_milliseconds_bucket{destination_workload=\"paymentservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)) / 1000) or histogram_quantile(0.99,
      sum(rate(istio_request_duration_seconds_bucket{destination_workload=\"paymentservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)), \"destination_workload_var\", \".\",
      \"destination_workload\", \"destination_workload_namespace\")",
  "proposedThreshold": "3.0",
  "description": "The 99th percentile of latency on the payment service",
  "proposedAppName": "paymentservice",
  "function": "out"
}
{
  "proposedName": "Payment P99 Latency Scale In",
  "proposedPrometheusQueryString": "label_join((histogram_quantile(0.99,
      sum(rate(istio_request_duration_milliseconds_bucket{destination_workload=\"paymentservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)) / 1000) or histogram_quantile(0.99,
      sum(rate(istio_request_duration_seconds_bucket{destination_workload=\"paymentservice\",
      reporter=\"source\"}[1m])) by (le, destination_workload,
      destination_workload_namespace)), \"destination_workload_var\", \".\",
      \"destination_workload\", \"destination_workload_namespace\")",
  "proposedThreshold": "0.9",
  "description": "The 99th percentile of latency on the payment service",
  "proposedAppName": "paymentservice",
  "function": "in"
}
```

Listing B.1: Watcher Target Definition

Appendix C

Scale Engine Service Definition

```
{
  "serviceName": "frontend",
  "dependencies": [
    {
      "serviceName": "adservice",
      "scaleFactor": 1
    },
    {
      "serviceName": "checkoutservice",
      "scaleFactor": 2
    },
    {
      "serviceName": "recommendationservice",
      "scaleFactor": 1
    },
    {
      "serviceName": "checkoutservice",
      "scaleFactor": 1
    },
    {
      "serviceName": "productcatalogservice",
      "scaleFactor": 1
    },
    {
      "serviceName": "currencyservice",
      "scaleFactor": 1
    },
    {
      "serviceName": "cartservice",
      "scaleFactor": 1
    },
    {
      "serviceName": "shippingservice",
      "scaleFactor": 1
    }
  ],
  "maxReplicas": 10,
  "minReplicas": 1
}
{
  "serviceName": "checkoutservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "cartservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "adservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "productcatalogservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "recommendationservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "shippingservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "currencyservice",
  "dependencies": [],
  "maxReplicas": 5,
  "minReplicas": 1
}
{
  "serviceName": "paymentservice",
  "dependencies": [],
  "maxReplicas": 3,
  "minReplicas": 1
}
{
  "serviceName": "emailservice",
  "dependencies": [],
  "maxReplicas": 3,
  "minReplicas": 1
```

```
}
```

Listing C.1: Scale Engine Service Definition