Towards Predictive Runtime Modelling of Kubernetes Microservices

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Masters of Science(Research)

at

The University of Waikato

by

Stephen Burroughs

2021

Abstract

Kubernetes is one of the major container management platforms utilised by Cloud Service Providers offering to host applications and services. As cloud based services become more prevalent, platform providers are faced with an increasingly complex problem of trying to meet contracted performance levels. Providers must strike a balance between management of resource allocations and contractual obligations to ensure that their service is profitable, while offering competitive pricing rates for contracts. This research explores performance modelling of microservice application tenants within the Kubernetes container management platform. We present a self-adaptive architecture to achieve modelling at runtime. We establish the potential for automated classification of cloud systems, and utilise a hybridised modelling approach to verify system properties and evaluate performance. We achieve this through the modelling of components as Extended Finite State Machines in WATERS, from which we automate the generating of performance models using the PEPA syntax.

Acknowledgements

I would like to acknowledge my supervisors, Dr Robi Malik and Dr Panos Patros, for their guidance and enduring patience through the research process. I would also acknowledge the Oceana Researchers in Cloud-Adaptive Systems (ORCA) laboratory for providing the necessary space and equipment, and Flavio Stutz for the opensource application used during the experimental process within this research.

Contents

1 Introduction 1

2 Background 4

2.1 Self-Adaptive Systems 4

2.1.1 Control Theory 5

2.1.2 Waves of self-adaptation 6

2.2 Cloud Computing 8

2.2.1 Cloud Service Providers 9

2.2.2 Platform as a Service 10

2.2.3 System Targets 10

2.2.4 Containerization 10

2.2.5 Microservices 12

2.2.6 Load Balancing 13

2.3 Kubernetes 14

2.3.1 Minikube 17

2.4 Modelling 17

2.5 WATERS 18

2.5.1 Discrete Event Systems 18

2.5.2 Supervisory Control 19

2.5.3 Extended Finite State Machines 20

2.5.4 Verification 21

2.6 Queuing theory 22

2.6.1 Stochastic Processes 23

2.7 PEPA 24

2.7.1 Process Algebras 24

2.7.2 Syntax 25

2.7.3 Model Solving 26

3 Related Work 28

3.1 Classification of Cloud Systems 28

3.1.1 Modelling 29

3.1.2 Kubernetes 30

3.2 Self-Adaptive Systems 31
3.3 Modelling Approaches 32
3.3.1 General Background 32
3.3.2 Performance 33
3.3.3 Verification 34
3.3.4 Other Modelled approaches 34
3.4 Research Areas 35
3.4.1 Architecture 35
3.4.2 Modelling Context 36
3.4.3 Self-Adaptation 36

4 Proposed Self-Adaptive Design 37
4.1 Self-adaptation Requirements in Kubernetes 37
4.2 Proposed architecture 38
4.2.1 Monitoring 39
4.2.2 Analysis 39
4.2.3 Planning 40
4.2.4 Execution 41
4.3 Implementation 41

5 Classifying Aspects of Kubernetes 44
5.1 Cluster Elements 44
5.1.1 Pods and Deployments 45
5.1.2 Nodes 45
5.1.3 Controllers 45
5.1.4 Deployed Applications 46
5.1.5 Horizontal Autoscaling 46
5.2 Request Handling 47
5.2.1 Resource Constraints 47
5.2.2 Internal processes 49

6 Experimental Setup 50
6.1 Minikube setup 50
6.1.1 Deployed Application 51
6.1.2 Load Balancing 51
6.1.3 Scaling 52
6.2 JMeter Scripts 53
6.3 PodMetrics 54
6.4 Postprocessing 54

7 Tuning Model Parameters 56

7.1 Variations to Experimental Context . . . 56

7.1.1 Deployment Configuration . . . 56

7.1.2 Request Configuration . . . 61

7.2 System Classification . . . 68

7.2.1 Limits . . . 69

7.2.2 CPU Cost . . . 72

8 Modelling Approaches 75

8.1 Workload . . . 76

8.2 Component Classification . . . 77

8.2.1 Pod Queues . . . 78

8.3 WATERS representation . . . 79

8.3.1 Automata . . . 79

8.3.1.1 Components and Constants . . . 80

8.3.1.2 Pods . . . 83

8.3.1.3 Requests . . . 87

8.3.1.4 System . . . 88

8.3.2 Verification . . . 88

8.4 PEPA . . . 89

8.4.1 Automatic Model Generation . . . 89

8.4.1.1 System Metrics . . . 90

8.4.1.2 Rates . . . 91

8.4.1.3 Node Resource Constraints . . . 92

8.4.1.4 Deployment Queues . . . 93

8.4.1.5 System Equation . . . 96

8.4.2 Assumptions . . . 97

9 Model Evaluation 99

9.1 Analysis of Deterministic Model . . . 100

9.1.1 Model Development . . . 100

9.1.2 Evaluation . . . 100

9.2 Probabilistic Modelling . . . 103

9.2.1 Model Evaluation . . . 103

9.2.1.1 Deployment Scenarios . . . 105

9.2.1.2 Request configuration . . . 110

9.2.2 Understanding Results . . . 118

9.2.2.1 Deployment Scenarios . . . 118

9.2.2.2 Request Configurations . . . 121

9.2.3 Further Model Refinement . . . 124

9.2.3.1 Restart Delays . . . 124

9.2.3.2 Classification 125
9.2.4 Modelling at Runtime 126

10 Conclusions **127**
10.1 Appraisal of Research Questions 128
10.2 Threats to Validity 129
10.3 Further Future work 130

List of Figures

2.1 A MAPE-K Loop 6

2.2 Application Encapsulation in a VM-based Cloud 11

2.3 Containerised Cloud Management 12

2.4 Kubernetes Architecture 15

4.1 Proposed Architecture 39

4.2 Stages of Proposed Self-Adaptive Architecture 42

7.1 Observed Replica Status for Deployment Scenario A 58

7.2 Observed Replica Status for Deployment Scenario C 59

7.3 Observed Resource Utilisation for Deployment Scenario B, Request duration 50ms 62

7.4 Observed Resource Utilisation for Deployment Scenario B, Request duration 200ms 63

7.5 Observed Replica Status for Deployment Scenario B, Request Duration 50ms 65

7.6 Observed Replica Status for Deployment Scenario B, Request Duration 50ms 66

7.7 Observed Replica Status of Pod Replicas in Deployment Scenario B 70

7.8 Balanced Resource Utilisation of Pod Replicas in Deployment Scenario B 71

7.9 Resource Utilisation Pod Replicas in Deployment Scenario B . . 73

8.1 Our Hybridised Modelling Approach 76

8.2 WATERS Elements 80
8.3 WATERS Pod component 83
8.4 WATERS Unpacked Deployment 86
8.5 WATERS Request Component 87
8.6 WATERS System Controller 88

9.1 WATERS Simulator, Deployment Scenario B 101
9.2 Modelled and Observed Replica Status For Scenario A, Requests of 3mb and 100ms 106
9.3 Modelled and Observed Replica Status For Scenario B, Requests of 3mb and 100ms 107
9.4 Modelled and Observed Replica Status For Scenario C, Requests of 3mb and 100ms 108
9.5 Modelled and Observed Replica Status For Scenario D, Requests of 3mb and 100ms 109
9.6 Modelled and Observed Replica Status For Scenario B, Request Duration 50ms 111
9.7 Modelled and Observed Replica Status For Scenario B, Request Duration 200ms 112
9.8 Modelled and Observed Replica Status For Scenario B, Request Size 1mb 114
9.9 Modelled and Observed Replica Status For Scenario B, Request Size 5mb 115
9.10 Modelled and Observed Replica Status For Scenario B, Autoscaling 117
9.11 Observed Replica Status Across Experimental Rounds for Scenario A with 2 Pod Replicas, Requests of 3mb 100ms Duration . 119

Chapter 1

Introduction

Distributed cloud systems are growing in prominence. Cloud service providers are faced with an increasingly complex task of balancing costs associated with infrastructure and resource utilisation, with contractual obligations to clients in the form of Service Level Agreements [4]. The *Kubernetes* cloud management platform is used by *Platform as a Service* (PaaS) providers to manage client applications. It is a *self-adaptive* platform, that makes use of *control-theoretical* approaches to maintain expected levels of performance. This research explores a potential extension to Kubernetes in the form of predictive management. We present a self-adaptive architecture to perform classification of systems and produce subsequent performance models at runtime. We employ two modelling paradigms to explore the adaptation of such a system as it experiences different degrees of load. We make use of the *Waikato Analysis Tool for Events in Reactive Systems* (WATERS) to create initial models using Extended Finite State Machines (EFSM) [56]. These models reflect the configuration and behaviour of deployed applications across a shared resource pool within a local Kubernetes Cluster. We verify the behavioural properties of EFSM models using the WATERS model checking framework. We present an automated approach to classifying aspects of *microservice* applications. We present a tool to automatically generate performance models using a *Performance Evaluation Process Algebra* (PEPA) syntax [40]. Our modelling tool produces models that adhere to the same logic outlined by our EFSMs. We evaluate the performance of physically deployed systems against our predictive modelling approach. We have three research questions that we seek to answer through this research.

1. How accurately are we able to capture the behaviours of the Kubernetes management platform using EFSMs?
2. Can we classify aspects of microservice applications to generate equivalent performance models?
3. Is there potential for such models to be used for predictive management?

We hypothesize that our modelling approach using WATERS will capture the interactions between request handling and Kubernetes reasonably well, thus providing a mechanism to help inform PaaS providers of possible impacts of configuration changes at design time. We anticipate that our PEPA approach will contribute further insight into expected levels of performance, from which adaptation strategies can be developed.

Key contributions of this work are as follows.

- An outline of our proposed self-adaptive architecture.
- An EFSM model that can be easily adapted to represent different cluster and deployment configurations at design time.
- A generator tool that establishes a model in PEPA syntax to evaluate the performance of our modelled system at design time.
- Evaluation of our modelling process and observations of encountered limitations.

We evaluate our models against observed performance of applications deployed to local Kubernetes clusters managed by *Minikube*. We establish performance observations using *Apache Jmeter*.

We envision that this approach may show the potential for usage by *Cloud Service Providers* (CSPs) to inform decisions about the acceptance of new client contracts. Configuration of client applications can be added to an existing WATERS model to see the potential impact on the status of a cluster. PEPA models can be used to evaluate the expected performance of the new application under different loads, and the consequent impact on cluster configurations. Furthermore, we expect that future development of our modelling approaches will work toward automating the classification and subsequent generation of system models at runtime. Such an approach falls within the third and fifth waves of adaptation [84]. We establish performance models of systems based on known aspects of our application (Wave III). We use these models to provide guarantees of performance under uncertainty (Wave V) [15].

This thesis has the following structure: Chapter 2 discusses the background information related to the presented research. This includes information regarding *self-adaptive systems*, *cloud computing*, *Kubernetes*, and the *WATERS* and *PEPA* modelling paradigms. Chapter 3 presents previous work related to our research topic. Chapter 4 presents our self-adaptive modelling architecture. Chapter 5 discusses behavioural aspects of the Kubernetes container management platform that we seek to capture. Chapter 6 discusses our experimental methodology for evaluation of our modelled and actual clusters. Chapter 7 describes the methodology behind our automated system classification approach. Chapter 8 presents the development of our system models and generation tool. Chapter 9 evaluates the performance of our modelling approach. Chapter 10 presents conclusions and future development of our research.

Chapter 2

Background

This chapter presents the background information relevant to the content explored throughout this thesis. First, it introduces the notion of self-adaptation from a software engineering perspective. Following this, it discusses the structure of modern cloud systems and the contexts within which they operate. It then introduces containerization, microservices, and the Kubernetes container management platform. Finally, it presents modelling practices and tools used during the research process.

2.1 Self-Adaptive Systems

Increased complexity and reliance on software-based systems necessitates effective management strategies. The aim of such management is the avoidance of performance inconsistencies and the increase in efficiency with which resources are allocated. Systems that integrate principles of *self-adaption* establish system goals in the form of *setpoints*. Setpoints represent the behaviours that the system is expected to adhere to. This is often achieved through the quantification of goals that are measured against performance metrics. The conceptual model of a self-adaptive system within a software engineering context is comprised of four basic elements:

- The *environment* within which a system operates. This includes both physical and virtual infrastructures across which aspects of a system operate.

- The *managed system* is the functional application code for which performance goals are created. Performance of the managed system is effected by resource availability and consequent processing capacity.

- *Adaptation goals* are quantified performance expectations that the managed system should adhere to. Examples of such goals are the mean response time of a system under dynamic load conditions, or the average utilisation of allocated resources. Adaptation goals are conditions placed on these metrics: e.g. response time is never greater than X.

- The *managing system* is comprised of multiple *controlling* components that effect changes to the relationship between the managed system and its environment to meet adaptation goals.

To achieve effective management of complex managed systems, sensory information relating to observed performance of system is scraped at runtime. The implementation of self-adaptive management of systems follows a *control-theoretical* approach.

2.1.1 Control Theory

A *control-theoretical* approach to software engineering entails the implementation of controlling components. Such components observe and analyse the behaviour of a system at runtime. Adjustments are made to system parameters to effect a change in behaviour. Controllers act to minimise the error between observed behaviours and defined setpoints. This thesis focuses on the Kubernetes container management platform. The major benefit offered by this platform is its self-healing and adaptive nature. This is achieved through a control-theoretical [36] approach to management. Kubernetes utilises several *controller* components that work through automated control loops. Figure 2.1 depicts the logic of a *MAPE(K) loop* that is the basis system controllers:

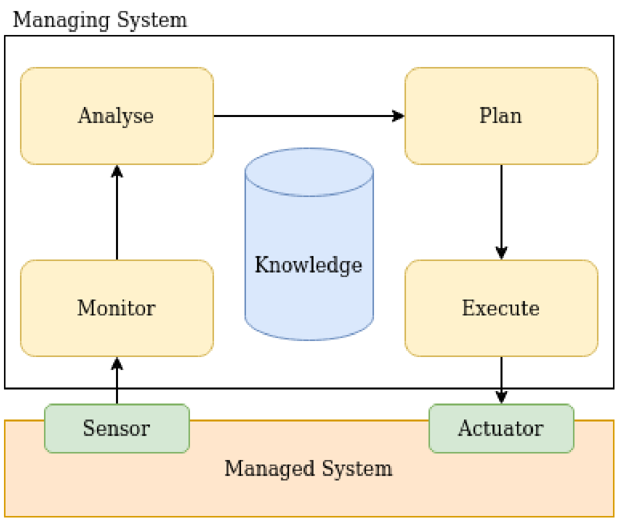


Figure 2.1: A MAPE-K Loop

- **Monitor** the system to evaluate an observed behaviour against an expected one.
- **Analyse** the current performance metrics that are produced through the monitoring process. Determine whether the specified system goals are adhering to specifications.
- **Plan** out a series of potential strategies to correct deviations from expected performance.
- **Execute** a correcting action to effect change in system behaviour.

The MAPE(K) loop is executed at run time. Comparisons are made between observed behaviour and specified setpoints. Controlling actions are taken to minimize the measured error between observed performance and setpoint values. To achieve this, a controlled system makes use of a **Knowledge** base that contains runtime information relevant to the management of the controlled system [15, 27].

2.1.2 Waves of self-adaptation

Software Engineering of self-adaptive systems has been developed through 6 waves of research [84].

1. **Automating Tasks** to support continuous management computing systems to mitigate human error when dealing with complex relationships between dynamic system components [50].

2. **Architecture-Based Adaptation** [52] to achieve abstraction and separation of concerns to offer general controlling strategies that are applicable across a range of systems and domains.

3. **Runtime Models** [9] to extend traditional model-driven engineering techniques to fit a self-adaptive context. Models abstract behaviours of underlying systems. Incorporation of models at runtime provides a mechanism to evaluate the impact and efficacy of adaptive strategies prior to implementation within the running system.

4. **Goal-Driven Adaptation** to achieve *fuzzy* behavioural aims rather than explicit requirements [6]. Goal-driven adaptation allows for temporary deviation from setpoint values that result from dynamic system behaviours. A good example of goal driven adaptation is a central heating system. When the desired temperature is increased, the managing system will implement controlling actions to achieve the new setpoint. The most effective strategy for this is to turn a heating component onto full power until the setpoint is reached. The cost associated with power consumption for this strategy is high. Similarly, maintaining an exact temperature may achieve setpoint adherence, but requires that heating and cooling processes are rapidly engaged. Such behaviour is inefficient and again has a high cost associated with power consumption. A goal-driven approach in this instance seeks to balance cost with setpoint adherence. Rather than meeting a desired temperature as soon as possible, controlling actions are taken that ensure the setpoint is eventually met at a lower cost.

5. **Guarantees Under Uncertainty** establishes a shift in the motivation for self-adaptation. Rather than improve upon the efficiency with which

systems are managed, this wave aims to minimise the impact of external *disturbances* on observed performance [21, 30]. Managing systems within this wave effect changes to achieve goal adherence when the managed system is exposed to changing conditions. An example of such a situation is the adjustment of resources assigned to an application for request handling when the incoming request rate is dynamic.

6. **Control-Based Adaptation** combines the principles of waves 2 and 5 to provide guarantees under uncertainty through the implementation of control theory as a self-adaptive framework [35]. The managing system acts to meet multiple system goals, with controlling actions following a mathematical structure that accounts for the magnitude of deviation from expected behaviours and the flow on effect across multiple aspects of the system.

Within this research, we present a self-adaptive managing architecture to extend the Kubernetes platform.

2.2 Cloud Computing

Coined by Google CEO Eric Schmidt in 2006, cloud computing describes a system that is hosted across multiple different *Physical Machines* (PMs) that are connected to each other through a network. Modern cloud systems can be seen as an evolution of earlier technologies, such as web based email clients and internet search engines [5], both of which rely on similar distributed systems [53].

The obvious advantage of cloud computing comes from the ability to displace a great deal of the cost associated with the infrastructure required to host a service. The cloud computing approach follows a client/server model, wherein an end user makes a request of an application hosted on the cloud infrastructure. Requests routed to the application for handling, before returning a response.

2.2.1 Cloud Service Providers

Many cloud systems are managed by *Cloud Service Providers* (CSPs). The costs associated with cloud system management and infrastructure can be significant. An alternative is to enter into a contract with a CSP. This relationship can be classified as a provider/client relationship, where clients are individuals or companies with an application to be hosted. The contracts between CPSs and clients form the basis of *Service Level Agreements* (SLAs). The degree of separation between a client and application management categorises CSPs into three main categories [37]:

- *Software as a Service* (SaaS) can be viewed as analogous to on-demand software. Software is hosted across a distributed cloud system, and is made available to clients on a subscription basis.

- *Platform as a Service* (PaaS) consists of hosting client services within a managed cloud platform. A client engaged with such a provider deals only with the data required for their service to run, while the PaaS provider manages the administration associated with running such a platform.

- *Infrastructure as a Service* (IaaS) is similar to PaaS, but without the management layer provided within a PaaS environment. Rather than a provider hosting and managing client services on a cloud platform, clients in an IaaS model manage their own platforms, but pay for the provided infrastructure.

A fourth cloud service model of *Runtime as a Service* (RaaS) also exists. In a RaaS model, clients pay for access to a runtime environment and are charged for resources on a utilisation basis. A potential extension of RaaS, *Application-Server as a Service* (ASaaS) [66] has been proposed as an alternative to these three standards. ASaaS builds on a *Runtime as a Service* model, wherein the proposed ASaaS approach improves the efficiency of resource utilisation in a RaaS context by sharing processes and libraries between tenants.

2.2.2 Platform as a Service

The focus of this research is the modelling of a cloud environment following a Platform as a Service model. Specifically, we evaluate resource utilisation of client services against observed performance. PaaS providers must compete with one another to provide a competitive price for the service provided. Contractual pricing rates seek to balance profitability with client demand. The PaaS provider has an interest in minimizing costs associated with operation of the platform, but must meet contracted levels of service. [22]

2.2.3 System Targets

From the PaaS provider perspective, service metrics are measured against *Service Level Objectives* (SLOs) [20]. Whenever a contract is entered into with a client, the service provider makes guarantees as to the expected performance of the client's hosted service as SLAs. SLAs typically involve performance metrics service response time or allocation of resources such as CPU or memory [49]. SLAs refer to the minimum standard of service expected of the provider, as established contractually with each respective client. To avoid SLA violations, CSPs establish SLOs with a higher standard of performance than that defined within SLAs. Actual performance is analysed to produce Service Level Indicators (SLIs). By taking corrective action when measured SLIs result in violation of SLOs, a provider is able to engage preemptive measures to avoid breaches of contractual SLAs.

2.2.4 Containerization

Alongside the evolution of cloud systems, the methods of deploying applications across a decentralised pool of resources have continued to develop. Effective management of cloud systems requires efficient distribution of applications across the available infrastructure. Development of *Virtual Machine* (VM) based cloud systems greatly increased the ability for providers to tai-

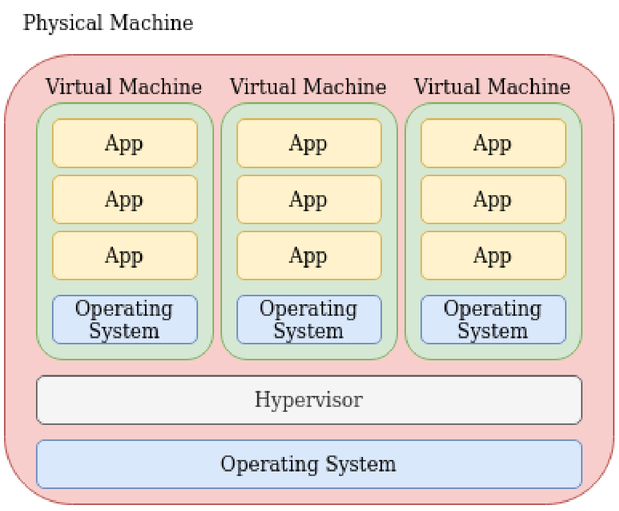


Figure 2.2: Application Encapsulation in a VM-based Cloud

lor their own platform architecture [8]. VM based cloud applications could be hosted on any PM that can meet the resource requirements. Figure 2.2 illustrates this style of architecture. Applications run within VMs running the required guest OS, while the VMs are managed by Hypervisors running on the host machine. Applications that run of different operating systems can be hosted on the same PM using VMs. While VM based architectures allow for flexible management of deployed services, they also have their own disadvantages.

- There is a high cost associated with running a hypervisor to manage VMs that greatly impacts efficiency of resource management.
- VMs have a time cost associated with booting up or shutting down the required host OS, which impacts the speed at which a system can reactively scale.
- Both of the above issues are exacerbated by the inability to share system resources between different VMs.

A *containerization* approach packages client applications along with any necessary dependencies into a lightweight wrapper that can be quickly created or destroyed [64]. This approach allows for the benefits of flexibility found within

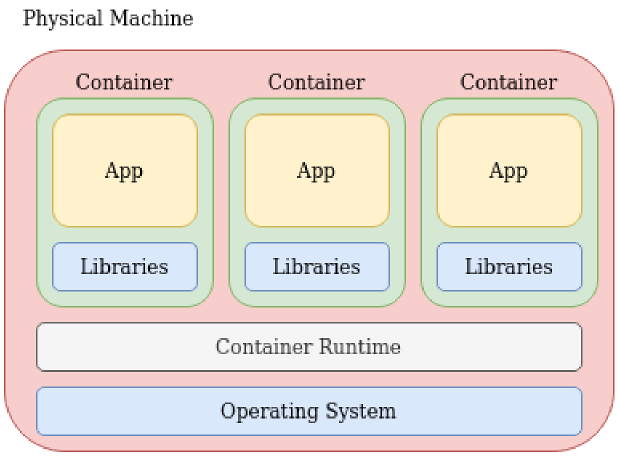


Figure 2.3: Containerised Cloud Management

VM-based systems, while also alleviating the cost associated with hypervisors. Hence, containerization allows for much more efficient management of multi-tenancy systems. This is achieved largely through the ability of multiple containers to share kernel resources across a physical host without the need for a hypervisor. As such, *containers* are able to be created and destroyed at a much faster rate than VMs and the resources required to operate an individual container unit are much less than that of a VM. Thus, container-based systems allow for a much greater degree of efficiency and reactivity when compared to an equivalent system that utilizes VMs [63].

2.2.5 Microservices

The lightweight nature of containerization is evident within *microsevice* architectures [3]. Applications that follow a microservice architecture package aspects of application behaviour as individual container images. These containers have a relatively low resource cost and consequent low capacity for request handling. A microservice architecture maintains expected levels of performance through *scaling* actions. The number of running containers associated with a given service are increased or decreased based on observed load to adjust the processing capacity of a given service under dynamic load conditions. Microservice architecture has the following benefits over a traditional monolithic approach [79]:

- Resource allocations can be managed efficiently as micro service containers scale up and down.
- The impact of individual container failures is decreased. Container failures impact a single aspect of an application, rather than impacting the application in its entirety. The proportional impact of each failure on performance is decreased. Services spread across many containers with small resource pools are impacted less by a container failure than a service spread across few containers.
- Resource requirements can be split into smaller amounts. Applications following a microservice architecture are well suited to deployment over distributed systems.

Within this thesis, we present modelling approaches based on *statespace* methods. Components within a microservice architecture can be modelled more effectively than monolithic applications using statespace methods. Complexity of models increases as the capacity of modelled components increases. As systems become more complex, the resulting statespace grows exponentially [82]. The modelling approach within this thesis is restricted to small applications that follow a microservice architecture to avoid statespace explosions.

2.2.6 Load Balancing

The increased flexibility associated with a containerized approach to cloud deployments, requires a *Load Balancing* service to ensure that workloads are distributed across all active containers in such a fashion that we end up with a relatively equal degree of stressing across all containers related to a service. The load balancer is tasked with redirecting incoming traffic to the containers that are associated with the application that is being engaged.

2.3 Kubernetes

Kubernetes is a container orchestration platform for adaptive clouds[1]. Designed by Google in 2014 and maintained by the Cloud Native Computing Foundation, Kubernetes manages distributed clusters through layers of component encapsulation. As such, it facilitates management of cloud based applications by abstracting away multiple levels of internal network management. Maintenance of the running cluster is achieved through controlling components that check the observed state of the cluster with expected configuration within a data store known as *etcd*. Kubernetes employs controllers to periodically check the observed configuration of the cluster against the expected configuration stored within etcd. These controllers then act to correct errors as they are encountered through controlling actions. Such a system can be described as *self-adaptive*. The units of encapsulation within Kubernetes are as follows [74]:

- **Pod** components encapsulate application containers. Each pod has an IP address, and contains one or more containers associated with a given application. Each pod has associated resource requests and limits. This thesis focuses on pods that contain a single container.

- **Deployment**s establish expected configuration of application components. Pod resources and replicasets are controlled by deployment configurations. Traffic is spread between pods in a deployment using an external Load Balancer.

- **Replica Set**s determine how many pod replicas of a given deployment are desired at a time.

- **Namespace**s can be used to link related deployments or establish *resource quotas* for specific applications

[1]https://kubernetes.io/

- **Nodes** are the physical machines that a Kubernetes *Cluster* spans across. These are referred to as *Masters* and *Workers*.
- **Cluster** refers to the entirety of the system, and encapsulates all nodes and deployments.

When part of an application needs to be accessible to an end-user, the relevant deployment is exposed as a *service*. The internal IP of each pod in a deployment is mapped to the exposed service as an *EndPoint* within etcd. The

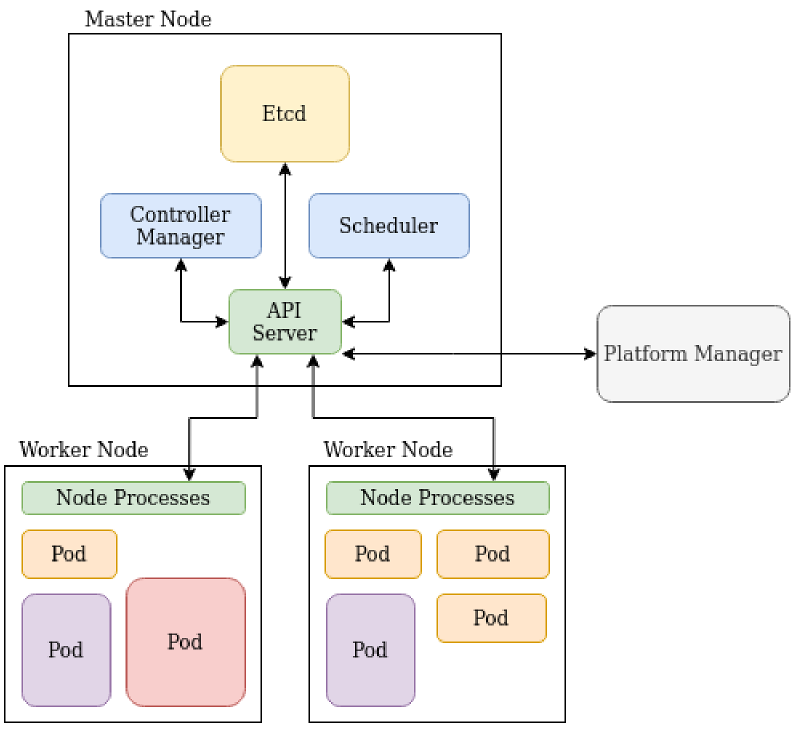


Figure 2.4: Kubernetes Architecture

infrastrucure of a Kubernetes Cluster is made up of Nodes. Worker nodes are the physical machines making up the resource pool onto which pods can be deployed. Each Worker has a *Kubelet* that facilitates communication between master components and worker nodes, several pods that encapsulate containerized applications, a Container Runtime, and a KubeProxy that facilitates internal cluster networking. Master nodes are reserved for cluster management components:

- **Kube-APIServer** facilitates communication between components within a cluster and exposes front-end APIs.

- **Kube-Controller-Manager** is responsible for keeping the real-time cluster architecture consistent with outlined configurations. A *Node-Controller* keeps track of current node performance and failures. A *Replication-Controller* keeps track of the number of deployed pods against each relevant replica set. An *Endpoints-Controller* makes sure that any exposed service is mapped to the appropriate pod within the cluster.
- **Kube-Scheduler** assigns pods to available nodes as they are created.
- **Etcd** is the distributed data store used within Kubernetes. When controller components of the cluster are engaged, they check the observed state of the cluster against the expected configuration found within etcd. Adjustments are made to the physical cluster as required to match the expected state.

The Kubernetes platform offers self-adaptive management of deployed applications. The self-adaptation within Kubernetes is achieved through the controlling components which all follow a MAPE-K loop, with etcd comprising the shared knowledge base. The platform architect applies new configurations in the form of new deployed applications or alterations to the running cluster environment. These configurations are uploaded and stored within etcd. This management approach provides two functionalities that are explored within this thesis:

- **Horizontal Autoscaling** can be implemented within deployments to establish a replica set that adjusts based on resource utilization [45]. Autoscaling allows for the adjustment cluster resources dedicated to running a particular deployment. Horizontal autoscaling achieves this by scaling the number of pod replicas between a set minimum and maximum number to meet resource utilisation goals.
- **Self-Healing** is a natural byproduct of the Kube-Controller-Manager.

> If a pod crashes, the discrepancy against the expected configuration is noticed. The Replication-Controller sends a request for a new pod which is scheduled onto an available node. The Endpoints-Controller ensures that the new pod is linked to the service, and removes the link for the old one.

A request can enter into the cluster at any physical node in the system. The request is aimed at a specific service hosted within the cluster. The load balancer is then engaged to route the request. It determines the route by checking the resource utilisation across all of the active pod replicas associated with the services' deployment, found by checking against the endpoint objects stored within etcd. The request is then directed to the pod with highest availability.

2.3.1 Minikube

Minikube is a tool that allows for the running of a single node Kubernetes cluster within a VM or container on a local machine. The controlling components are deployed within a "$kube - system$" namespace, which mimics a master node in an ordinary cluster. This local environment is the context within which experiments are run in this thesis.

2.4 Modelling

Within any system, processes take place that transform the system itself from an initial state to a number of given potential outcomes. The collection of all possible outcomes resulting from *actions* across the different components within a system is referred to as the statespace of the system. The actions undertaken by components along with the associated rate at which they occur are viewed as *transitions*. Model checking algorithms are based on an underlying framework known as the Specification Problem [28]: Given a system M with specification h, determine whether the behaviour of M fits h. The application for this approach within the context of a Kubernetes architecture is as follows:

- Given a Kubernetes Cluster with a known load, how accurately can a model reflect observed performance against an established SLO?
- Given a Kubernetes Cluster with known performance, how accurately can a model predict observed behaviours when load scales or new tenants are introduced?

This thesis focuses on two modelling approaches, the first utilises the Waikato Analysis Tool for Events in Reactive Systems (WATERS) which allows for creation and verification of models using modular automata to represent discrete event systems. The second utilises Performance Evaluation Process Algebra (PEPA) to model expected performance of Kubernetes clusters modelled as stochastic processes represented by Markov Chains.

2.5 WATERS

The Waikato Analysis Tool for Events in Reactive Systems (WATERS)/Supremica is a tool developed to provide an environment within which large discrete event systems can be modelled and analysed [2]. It utilises supervisory control theory [75] from which a complex system can be represented using modular components with shared events.

2.5.1 Discrete Event Systems

Many systems cannot be easily or accurately represented via mathematically defined relationships, but rather as occupying one of a given finite series of states at any one time. Such systems can be referred to as *discrete event systems* and modelled through automata [33]. A good example of such a situation would be modelling the potential behaviour of a production factory. In such a situation, we are concerned about whether a particular manufacturing component is idle, working, or finished. In such a context, the focus is on whether the required component is in an acceptable state at the point at which it is needed. Likewise if we examine a traffic system, the aspect of focus is that two opposing signals are not both green at the same time. In both of these examples, the focus of model checking is to determine whether the system functions as expected or enters states classed as *forbidden*, while transitions are symbolic rather than numerical. WATERS provides a tool to explore these systems through modelled automata, which represent an abstraction of components within the system. The interaction between these components can then be used to model supervisory control or potential blocking aspects of a system.

2.5.2 Supervisory Control

Supervisory control can be used to model complex discrete event systems. Each system component is modelled individually. A supervisor component models system behaviours that must be adhered to. The synchronisation of transitions between system components and a supervisory establish controlled behaviours [23]. WATERS allows the user to model systems using components known as *plants* and *supervisors*. Modelled behaviour of components is controlled by synchronising transitions across plants and supervisors. Each plant or supervisor is a *Finite State Machine* (FSM) representing the states of an individual component and the transitions between each of said states along a labeled pathway. Thus, each FSM can be represented as a 5-tuple [57] $G = (\Sigma, Q, \rightarrow, Q^i, Q^m)$, where:

- Σ represents a finite *alphabet* of events that exist within the system. This is a combination of the sets Σ_u of uncontrollable and Σ_c of controllable events.
- Q is the set of possible *states*.
- $\rightarrow$ represents transitions between states of a system along a labeled *event* pathway.

- $Q^i \subseteq Q$ represents the subset of *initial* states of the system
- $Q^m \subseteq Q$ represents the subset of *marked* states in the system

States marked as accepting within a component represent what might be considered as an *idle* stage, wherein the component is ready to be engaged. The events within a system can be shared across components and are classified as *controllable* or *uncontrollable*. Controllable events can be blocked by a system's specifications, while uncontrollable events must always be able to occur. Where components share the same actions, WATERS uses synchronous composition in line with the theory of Communicating Sequential Processes [13]. The shared event must be consistently controllable or uncontrollable across all components in which it exists, the transition in one component can then only occur if it is also enabled in each other component. At the point at which a transition does occur in a component due to a shared event, all other components with the event will also experience the relevant transition. Our WATERS modelling approach makes use of this synchronous property. We map the effect of transitions across multiple components through shared events.

2.5.3 Extended Finite State Machines

WATERS allows for control of systems by extending traditional Finite State Machines through the incorporation of *Variables*, *guards*, and *actions* [62].

- Variables allow for modelling of system limits or behaviours [81]. Variables are integer values that are dynamically adjusted by transitions between states. Variables are bounded within a defined range. These variables can be referenced by guards and actions.
- Guards enable or disable transitions based on a logical formula.
- Actions update variables as events occur. Actions are limited by the defined variable range.

A system model within WATERS is thus represented by component plants and controlling specifications. Plants model the behaviour of actual components through states and transitions. Specifications are viewed as controllers that enforce the limits of the system that interacting components must act within by associating guards with given events. The system is considered controllable if the given specifications define behaviour for all instances of uncontrollable events across all plants in a system [56]. The WATERS modelling approach within this research establishes the current state of cluster components through tracking of variable values. Components contain logical guards defined within plants and a system specification. In this manner, we track the state of resource availability and utilisation across components within the system. Furthermore, analysis of the generated state space provides insight into the competition for resources between deployments as they scale. Thus, we establish possible cluster configurations as components react to incoming request actions.

2.5.4 Verification

Once the plants and specifications of a system are established, WATERS allows for verification [60] of the model properties through the following methods:

- *Controllability* determines whether the system meets the definition of control as previously discussed.
- *Conflict* checking determines whether the system can finally reach a state marked accepting.
- *Deadlock* checking finds any states from which a system cannot transition out of.
- *Control Loop* checking finds any potential *livelocks*, wherein a system is stuck in a repeating loop of controllable transitions.
- *Property* checking allows for predictive modelling of system behaviours, by checking the model against known behaviours.

This type of analysis is useful for determining reachability of a state given a well defined model, but does not provide useful information about expected system performance. This is an inherent property of any Discrete Event System (DES). Such systems model transitional behaviour between system states without regard to the rate at which a given transition might take place. Our modelling approach makes use of the verification provided by the WATERS framework. Properties of our WATERS system model are verified. We then transpose the logic of our model into a PEPA syntax. Our PEPA model supports activity rates for transitions. We evaluate the performance of our PEPA model.

2.6 Queuing theory

A job in any given computer system has three main states. It arrives at the system, it is worked by the system, and it leaves the system. With this basic model, the departure time of any job can be easily predicted with given knowledge of the arrival time and the resource and I/O requirements of the job. As more jobs are added concurrently to the system, the competing nature of jobs with each other for resources results in performance delays as resources are used and freed. When these jobs come in the form of user input across a distributed system, the rate at which jobs arrive in a system can be viewed as inherently stochastic. Queuing theory [38] can be applied to such a system, given knowledge of resource costs and an exponentially distributed arrival rate, to predict expected performance levels across a system design. We utilise queuing theory to model capacity of pods through observed levels of concurrency. Our models can thus be viewed as queues, where each position reflects a granular level of concurrent active requests.

2.6.1 Stochastic Processes

Stochastic Processes are a sequence of random variables representing generalized behaviour of a system. Such processes are best suited to representing systems for which certain behaviours can be assumed. Arrival rates on incoming jobs are typically constant over a time period, but the arrival time of each individual job is unpredictable. Stochastic processes can be modelled using *Markov Processes* to represent the evolution of the system state space at any given time step [54, 31]. Markov processes are well suited to modelling stochastic processes as they possess a *memoryless* property with regard to transition rates. This means that the probability of being in a state at time t is dependent on the state at time $t-1$ but independent of all states prior. Rather than using specific numeric rate values, Markov processes utilise a mean value to represent a rate with an exponential distribution. Such an approach means that long term trends will be well represented as the system evolves, even if the actual rate fluctuates around the expected mean. Discrete Time Markov Chain (DTMC) represents a system wherein transitions occur at discrete time steps. Continuous Time Markov Chains (CTMC) allow for transitions regardless of time steps. This thesis focuses on CTMC representation of systems. The memoryless nature of transitions within the chain means the rate at which a transition occurs is independent of how many transition have occurred prior, or how long the system has been in the current state. As such, the exponential variable can be viewed as analogous to a Poisson Distribution, where:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

- X is a discrete random variable representing the number of times a given event occurs over a time period.
- k is a positive integer representing the observed number of events that has occurred.
- λ is the expected value of X.

Within the context of a Kubernetes Cluster, λ can be seen as equivalent to the expected number of incoming jobs per second; the actual value may vary between 0 and infinity, while the exponentially distributed variable has a mean at the expected average. Markov chains can be visualised as Finite State Automata, much like those found within the WATERS modelling context. The benefit of Markov chains is the ability to associate a rate λ with a transition action, allowing for the evaluation of likely outcomes within a non-deterministic context [16]. The previously mentioned memoryless nature of Markov processes means that the steady state probability of transitioning from any given state to another in the chain can be represented as a singular matrix with dimension $n \times n$ where n is the size of the state space. Each row within the matrix is associated with a the system being in a single state, with the entries in the row depicting the probability of transitioning out of the current state into the state associated with the column. Performance Evaluation Process Algebra makes use of this underlying matrix to analyse the expected behaviour of a defined Markov process, performing a steady state analysis to determine the probability of being in any given state of the process at any given time.

2.7 PEPA

Performance Evaluation Process Algebra extends a traditional approach to Process Algebras by associating probabilities with system actions and utilising a compositional approach to model building [41]. These qualities make it suitable for analysis of dynamic systems.

2.7.1 Process Algebras

Process algebras offer mathematical theories that allow for the modelling of concurrent systems via algebraic expressions [7]. Each component within a system is represented as a separate process. Interactions that alter the state of a process can be modeled from a set $\mathcal{A}$ of system *actions*. In traditional process algebras, probability is abstracted away in favour of non-deterministic choice [42]. Thus, actions can be viewed as analogous to the instantaneous transitions within WATERS. PEPA extends this traditional functionality by using *activities*, which comprise an *action-rate* pair: $a = (\alpha, r), \alpha \in \mathcal{A}$. Each activity is then given an exponentially distributed duration as a function of r, with a mean time of $\frac{1}{r}$. The likelihood of a given activity within a process taking place within a given time can be given as a function of time: $f_a(t) = 1 - e^{-rt}$, where r is the associated rate and t is the time step. Such an approach allows for the generation and analysis of the underlying Markov process, which employs such exponential variables.

2.7.2 Syntax

PEPA models adhere to the following language:

$$P ::== (a, r).P \mid P + Q \mid P \underset{L}{\bowtie} Q \mid P/L \mid A$$

- **Prefix** $(a, r).P$ denotes the basic mechanism through which interactions are modelled within PEPA. A component undertakes an activity consisting of an action and associated rate, after which it behaves as component P.

- **Choice** $P+Q$ denotes the competing nature of processes. Multiple activities being enabled simultaneously results in multiple possible outcomes. This can be viewed as a logical OR, with the outcome probabilities determined by the given activity rates.

- **Cooperation** $P \underset{L}{\bowtie} Q$ allows for parallel composition of processes across a system. This provides functionality through which to represent synchronising processes where L is the *cooperation* set of actions between processes P and Q. This means that where P experiences an action in L, Q must experience the same action. The rate of the activity is determined by the lowest rate, as fast processes must wait for slower ones to finish. If L is empty then $P \underset{L}{\bowtie} Q$ represents parallel composition, where there is no interaction between components. This is also represented as $P||Q$

- **Hiding** P/L denotes that actions in set L for process P are not visible to other components. This is useful where detail about specific processes can be abstracted away and only the rate is relevant

- **Constant** $A \overset{def}{=} P$ are the names assigned to evolving processes. A constant is an abstraction representing the possible activities undertaken from a particular state in a given process. Each constant represents a state in the associated component.

Where a particular component has multiple potential activities associated with the same action available at the same time, each activity is viewed as independent. While WATERS approaches modelling through visual representation of components as FSAs, PEPA has a focus on performance evaluation. As such, component states and transitions within the model are represented textually as process algebra. The resulting model is less intuitive than that of the WATERS approach, but allows for a much greater degree of evaluation due to the analysis of the underlying Markov process. While components can be easily defined and segmented as different plants within WATERS, there is no encapsulation of components within PEPA. Instead, all constants that are linked through an activity can be viewed as potential states of the same component. As such, the potential transitions from any given state within a component is represented by the choice between defined activity prefixes that lead to the next state.

2.7.3 Model Solving

When an activity completes, PEPA uses preemptive resampling with a restart to determine the next activity from the available choices for each component state. This means that each activity within the system as a whole is treated equally, with activity weighting remaining consistent regardless of how much time has passed since it was enabled. Thus, the exponential rate distribution associated with transitions is independent of time. A derivation graph represents all of the potential interactions between different states in a process due to the enabled activities. The initial node of the graph represents the combination of all system component initial states. A race condition is used to determine the probability of each activity branch from each node in the graph. The likelihood of each activity occurring is calculated as a ratio of the associated rate against the sum of the rates all enabled activities. Each selection is independent and only one activity can fire at any timestep. By associating each node in the graph with a system state, and each activity branch between nodes in the graph with a given action, an underlying Markov process is identified that represents the states and transitions within the system. Analysis of this Markov process can be performed to find the steadystate probability associated with each state of the modelled system.

The research presented within this thesis uses queuing theory to model behaviours of microservices within a Kubernetes cluster using WATERS and PEPA. WATERS is used for initial creation and verification of models. An automated generation tool is used to transpose the logic of our WATERS model into a PEPA syntax for performance evaluation.

Chapter 3

Related Work

Within this chapter, we present a summary of previous research toward modelling of cloud computing systems. We examine existing approaches to modelling cloud systems across different architectural contexts, along with previous work that relates to the modelling approaches taken within this thesis.

We classify our selected papers into the following key sections:

- Classification of Cloud Systems.
- Self-Adaptation.
- Modelling Approaches and Efficacy.

3.1 Classification of Cloud Systems

The use of cloud-based distributed systems is a recent phenomenon and as such, the standard of implementation has evolved rapidly in recent years. Hence, to classify and understand the behaviours of such a system, one must have familiarity with the history of cloud environments and functional aspects therein.

Dillon et al [24] Introduce key underlying concepts within the context of cloud computing, specifically focusing on the history of Grid Computing and service models found within a cloud environment. The presented work contextualises the relationships between Service-Oriented Computing (SOC), Cloud

Computing, and Grid Computing, with particular interest in the constraints experienced by service providers when attempting to achieve established SLOs. Gibson et al [37] present and discuss the differences between cloud service models from the perspective of a service provider. The presented work establishes a clear distinction between cloud contexts using Platform, Infrastructure, and Software as a Service. Magulari et al [55] present an approach to classifying capacity of cloud systems based on resource utilisation and implements an approach that utilises queuing theory to evaluate the impact of load balancing on resource allocation. Ardagna et al [4] provide a robust overview of existing approaches to modelling QoS within cloud systems. The present paper discusses the different approaches of workload vs. system modelling, and the different applications of each for evaluating QoS with regard to SLAs. Boniface et al [10] present a good overview of QoS management of cloud systems across Software/Platform/Infrastructure as a Service. Patros et al [67, 70] use *CloudGC* to benchmark the observed performance of deployed applications under different *Garbage Collection* policies. Further work [69] proposes the use of *Elastic GC* to minimise the impact of GC processes on application performance.

3.1.1 Modelling

With regard to modelling, existing research seeks to characterise cloud behaviours for the purpose of modelling performance and system verification. Existing research explores elastic scaling of applications within a cloud environment. Performance of vertical and horizontal scaling algorithms is modelled [51, 77], along with predicted adherence to SLOs [12, 80].Brebner et al [12] focus on performance prediction based on anticipated time durations. Souri et al [80] discuss the impact of elastic scaling on SLO adherence. Further existing research. analyses resource utilisation patterns across applications deployed to a cloud environment [78, 71, 25]. Shawky et al [78] presents an approach to modelling resource allocation within the context of an Aneka PaaS framework,

while Ding et al [25] explore the impact of heterogeneous resource pools within an IOT context. Patros et al [71] present further analysis of resource interference within a cloud environment that houses multiple scaling tenants.
We seek to differentiate our research from existing work through the development of runtime modelling strategies.

Existing research explores methods for classifying performance of cloud applications under dynamic load [73]. Patros et al [65] explore a potential approach to mitigating performance issues in multitenant clouds through sharing of dynamically compiled artifacts. Further research explores approaches to avoid SLO violations through a combined approach of autoscaling and prioritisation of request handling [68]. Calheiros et al [14] present the CloudSim toolkit to simulate the performance of a VM based cloud under dynamic load, others explore the efficacy of load balancing approaches across distributed systems [11, 55]. Zhu et al [86] establish performance benchmarking for *Node.js* applications under different scaling conditions.

Work presented within this thesis compliments and improves upon existing research by automating parts of the system classification and modelling processes. Previous work presented so far establishes methodologies for classification of cloud performance. We differentiate our research from extant work by presenting a framework that automates aspects of this classification process.

3.1.2 Kubernetes

The following works relate to performance and behaviours of applications hosted in a Kubernetes cluster. Casalicchо et al [18] evaluate the efficacy of multiple autoscaling algorithms with regard to observed performance, while van Zijl et al [83] model the logic of a horizontal autoscaling component to establish SLO compliance. Evangelidis et al [32] establish a model for verification of autoscaling policies using the PRISM model checker, with an aim of guaranteeing performance under uncertainty. Podolskiy et al [72] present

a machine learning approach to optimise scaling of deployed applications. All of the above works present research toward performance modelling of Kubernetes autoscaling behaviours. Podolskiy et al implement self-adaptative strategies, with the focus is on vertical scaling through machine learning techniques. Medel et al [59] present an approach to modelling the internal components of a cluster through Extended Petri Nets. This approach was later extended [58] to evaluate the impact of internal pod configurations on performance.
We identify the development of a framework for classification of microservice application performance as a gap within existing research. Furthermore, with the exception of Podolskiy et al, existing research does not explore self-adaptive implementation of modelled strategies. Podolskiy et al implement self-adaptation to achieve vertical autoscaling, rather than horizontal. They approach this through machine learning, rather than performance modelling.

3.2 Self-Adaptive Systems

We present the following works that relate to development and implementation of self-adaptive strategies.
Jiang et al [44] explore machine learning strategies for predicition-based provisioning of cloud tenants. Esfahani et al [29] present the POISED framework for mitigating the effects of uncertainty in self-adaptive systems. Jiang et al [43] present A Self-Adapative Prediction (ASAP) system for dynamic resource provisioning of VM clouds. Cedillo et al [19] propose a framework for producing runtime models of cloud systems. Heinrich et al [39] present work toward implementation of runtime prediction modelling that reflects dynamic changes of cloud environments. Calinescu et al [17] present adaptive model learning methods to predict non-functional system properties using discrete time markov chains. Johnson et al [48] present an approach to formalise cloud resource usage as probabilistic patterns. They synthesise Markov Decision Processes to evaluate probable resource utilisations.

The works of Jiang et al [44, 43] implement self-adaptive frameworks to improve performance. These works focus on machine learning techniques rather than performance modelling. Cedillo et al and Heinrich et al present frameworks for producing runtime models. These approaches illustrate the potential of self-adaptive management, but are distinct from the modelling approaches within our research. Existing research involving Markov Decision Processes establish the potential for probabilistic performance modelling, but implementation is achieved through the PCTL language using the PRISM model checker.

3.3 Modelling Approaches

The papers within this section reflect current modelling approaches for cloud based systems. The approaches taken can be broadly split into two categories. Those that focus on modelling performance, and those that focus on system verification. Furthermore, we also present relevant papers that establish the required understanding of the modelling approaches undertaken within this research.

3.3.1 General Background

Hillston et al [41, 40] discuss the development of process algebras and introduce the PEPA modelling language. Presented work evaluates the efficacy of previous approaches such as existing process algebras and petri nets for modelling computer systems. Hillston discusses the timed extension of existing frameworks and provides a good overview of approaches to modelling stochastic systems. Hillston presents the PEPA modelling language as syntax for modelling stochastic processes, along with concise documentation of the PEPA syntax and the construction and evaluation of the underlying Continuous Time Markov Chain (CTMC) that represents the modelled system.

Malik et al [57, 56] present an approach to modelling systems using WATERS. These works provide an overview of the WATERS/Supremica modelling framework with discussion included regarding the notion of supervisory control. Teixeira et al [81] discuss the representation of system components as Extended Finite State Machines, and establish how such models can be analysed to verify system behaviours. Further insight into the modelling environment within WATERS, along with various system models and accompanying documentation.

3.3.2 Performance

Current research into performance modelling of cloud systems establishes several frameworks. Calheiros et al [14] discuss issues relating to cloud performance under dynamic load, and introduce CloudSim as a simulation tool for evaluating expected performance. Other frameworks model performance through mathematical approaches [12, 55]. Brebner [12] establishes mathematical relationships to predict elasticity characteristics of cloud applications. Maguluri et al [55] model expected performance of stochastic cloud systems across different load balancing algorithms.

We find that several modelling approaches have been developed for cloud systems using the PEPA syntax [78, 11, 25, 77, 26]. Shawky et al [78] model resource allocation within the context of the Aneka PaaS framework. Initial modelling is achieved by outlining the behavioural logic within the syntax of Stochastic Process Algebra, from which a PEPA model is derived to evaluate performance. Bravetti et al [11] explore the impact of load balancing and scaling on a distributed system. The focus of the presented research is modelling the distribution of load across component populations and does not account for internal component behaviours of scaling of tenants. Ding et al [25] use PEPA to model predicted response times for various services across a heterogeneous infrastructure, with the aim of predicting response times across different cloud contexts. Further work models resource utilisation in VM clouds [26]. Sha et al [77] model an Openstack cloud architecture, with a focus on the scaling of VMs in response to load. This approach involves the modelling of multiple nodes, each of which encapsulates different aspects of system behaviour. Research presented within this thesis follows a hybridised modelling approach for evaluation of performance and verification of behaviours. Our automated classification and generation processes are distinct from extant PEPA works.

3.3.3 Verification

Medel et al [59, 58] approach performance modelling of a Kubernetes architecture using extended Petri Nets to capture aspects of system performance. Internal component behaviours are are modelled as Petri Nets to capture resource management and system capacity. Khebbeb et al [51] evaluate the performance of a controller to decide elastic scaling events within a cloud environment. Behaviours are depicted as bigraphs, while FSAs following a Kripke structure employ supervisory control. Other works model autoscaling of cloud systems to verify the adherence to SLOs [32, 83]. Evangelidis et al [32] present work toward predicting the autoscaling performance of VM based cloud environments, while van Zijl et al [83] model an horizontal autoscaler component within a Kubernetes architecture.

Van Zijl [83] models Horizontal Pod auto-scaling within Kubernetes using EFSMs in WATERS. The focus of the presented research in this case was the capturing of auto-scaling behaviours. Sha et al. [77] and Ding et al. [26, 25]model cloud performance using PEPA, but focus on component utilisation of tradition VM architectures rather than the adaptation of a Kubernetes-style architecture under load conditions.

3.3.4 Other Modelled approaches

We find several literature reviews that evaluate cloud modelling frameworks [4, 76, 34]. provides a robust overview of existing approaches for modelling QoS within cloud systems. The presented work evaluates different approaches of workload vs. system modelling and the different applications of each for evaluating SLO adherence. Ardagna et al. also present a high level overview of how Queuing theory can be applied to represent such systems, along with discussion around existing modelling paradigms such as stochastic Petri Nets. Sakellari et al [76] present and evaluate formal approaches to mathematical modelling of cloud systems within a research context. The authors review existing approaches for simulated and physically deployed cloud systems, with a focus on larger systems of scale. Jindal et al [46] present an approach to performance modelling of microservices using the *Terminus* modelling tool. This approach benchmarks performance based on CPU utilisation for given request loads to predict behavioural aspects of a Kubernetes cluster. Johnson et al [47] present the *INcremental VErification STrategy* framework as a mechanism for re-verification of component-based models. This approach makes use of Discrete Finite Automata and high level algebras to establish probabilistic models of computing systems.

3.4 Research Areas

The above outlined work establishes a strong basis from which to explore performance modelling of cloud systems. The intention of this thesis is to achieve this same end, while avoiding repetition of prior work. As such, We focus on three key areas. Architecture, modelling context, and self-adaptation.

3.4.1 Architecture

Much prior work relating to cloud performance is based on a traditional VM architectures. Research into the performance of containerized applications within the context of a Kubernetes cluster is currently scarce. With the increased prevalence of microservice architectures, we identify this as an area in need of further research. We model microservice applications hosted within a Kubernetes cluster. We model pods as units of container encapsulation to evaluate performance.

3.4.2 Modelling Context

Previous work on cloud performance modelling predominantly makes use of simulation techniques [14, 12] and modelling paradigms such as extended Petri Nets [59]. Extant modelling frameworks predominantly on either performance or behaviour modelling. We combine these two aspects in a hybridised approach that makes use of parallel modelling paradigms. We model performance as DES models using EFSMs, while performance is captured through probabilistic modelling using the PEPA syntax. Furthermore, we present work toward automatic generation of models at runtime.

3.4.3 Self-Adaptation

We identify the automation of classification and modelling processes as an area in need of further research. Much existing research on microservice performance focuses on classification or modelling at design time. A key contribution of our work is a framework for automating the classification and subsequent modelling of microservice applications. We present a self-adaptive architecture that utilises our proposed framework to model performance at runtime. The content presented through the rest of this thesis documents the initial development and evaluation of our self-adaptive architecture.

Chapter 4

Proposed Self-Adaptive Design

Self-adaptive systems alter aspects of their operational context and behaviour to meet performance goals under changing conditions. The self-adaptive architecture presented within this chapter improves upon the performance of a Kubernetes cluster through predictive performance modelling. The goal of our architecture is to achieve a greater level of consistency for observed performance of microservice applications managed by the Kubernetes platform. Our approach achieves this goal through scaling actions that adjust the processing capacity of microservices based on expected future performance. Our architecture combines the 6 waves of self-adaptation [84]. Within this chapter, we present the underlying architecture of our self-adaptive approach

4.1 Self-adaptation Requirements in Kubernetes

As the reader may recall from previous chapters, primary benefits of the Kubernetes container management platform are the self-healing and self-adaptive behaviour that is established through controlling components. This form of self-adaptation fits the definition of reactive control. Kubernetes controllers monitor the performance of pod replicas across deployments. Corrective actions are taken when variation is encountered between configured expectations and observed behaviours. A drawback of this approach is that decreased performance is observed during the time required for corrective actions to be implemented [61]. In the case of pod failures, the delay in corrective actions results in the remaining active pod replicas experiencing an increase in load. This leads to compounding failures. In cases where CPU throttling is engaged, backlogs of requests develop which impact the future observed service rate of pods. Consequently, the efficiency with which corrective actions can be engaged is limited. An observed outcome is that pod replicas within a deployment oscillate between a failed and active status. Active pods experience a spike in load and fail during the time in which failed pods are restarted. The microservice application thus operates at reduced capacity.

While Kubernetes engages in self-adaptive behaviours, the associated performance is sub-optimal. The incorporation of additional tools to achieve greater efficiency is required to satisfy performance requirements associated with microservice applications. Such a tool requires knowledge of anticipated performance to avoid the issues associated with delayed controlling actions.

4.2 Proposed architecture

We propose a self-adaptive architecture to improve upon the efficiency of corrective actions using predictive performance evaluation. Our approach incorporates a new controlling component that takes corrective action based on anticipated performance. This mitigates the compounding impact of request handling in under-provisioned deployment contexts. The research presented within this thesis contributes to our proposed architecture which executes controlling actions based on a combination of observed performance and predictive modelling. Our approach follows that of a Monitor-Analyse-Plan-Execute-Knowledge (MAPE-K) architecture for self-adaptive systems [50]. The aim of our approach is to facilitate adaptation of a Kubernetes cluster to meet performance goals under degrees of uncertainty. We classify the different phases of our approach within the MAPE-K context.

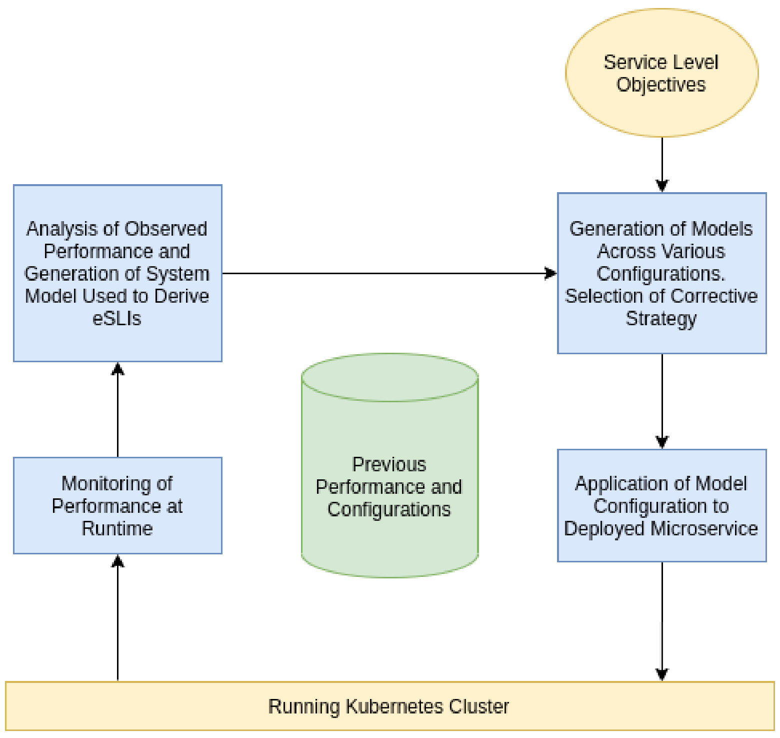

Service Level Objectives
Analysis of Observed Performance and Generation of System Model Used to Derive eSLIs
Generation of Models Across Various Configurations. Selection of Corrective Strategy
Previous Performance and Configurations
Monitoring of Performance at Runtime
Application of Model Configuration to Deployed Microservice
Running Kubernetes Cluster

Figure 4.1: Proposed Architecture

4.2.1 Monitoring

We monitor observed behaviour of deployed microservices at runtime. We track resource utilisation and pod failures. Monitoring occurs at defined time intervals, which provides average metrics across the given time period.

4.2.2 Analysis

Resource utilisation metrics are analysed to find an average cost associated with request handling. Pod failures are analysed to establish limits associated with concurrent handling of requests. Resource utilisation and pod failures are our SLIs. These are used in conjunction with known deployment configurations to generate a model that reflects the capacity and maximum throughput of pod replicas within the deployment. The expected performance of our model is referred to as our expected SLIs (eSLIs). There are two possible outcomes of model analysis.

- Where average load patterns are observed to be static, our approach models eSLIs for load patterns where the rate of incoming requests are within a tolerance range above that of our observed system. This establishes predictive performance that accounts for variation of incoming rate around the observed average.

- Where load patterns are observed to follow an increasing trend, our approach models eSLIs for predicted load based on the observed trend. This establishes predictive performance that accounts for changing load conditions.

We only model scenarios where the incoming request rate is equivalent to or greater than that of our observed system. Under-utilisation of resources does not have a compounding impact on future performance. Possible over-utilisation is therefore the limiting aspect of performance. Through this approach, we establish eSLIs based on the observed performance of our microservice at runtime. These eSLIs are compared against target SLOs to determine whether corrective actions must be taken.

4.2.3 Planning

In situations where our analysis phase dictates that corrective actions must be taken, the planning strategy within our approach is as follows:

- The direction of corrective action is determined. Within the context of horizontal scaling discussed within this thesis, the required change in an increase or decrease in the number of pod replicas associated with a deployment. Where autoscaling is enabled, corrective action is taken to adjust the maximum and minimum number of pod replicas associated with the deployment.

- Probabilistic models are automatically generated using information attained through the monitoring stage. Models are generated using incoming request rates within a range established in the analysis stage. Multiple models are generated using different pod replica configurations.

A model is selected based on the adherence of eSLIs to SLOs and the resource cost of the modelled configuration. The planning phase of our self-adaptive strategy models potential configurations of our deployed microservice against possible load patterns. It evaluates the eSLIs and resource cost associated with each configuration. A modelled configuration is selected that produces eSLIs that adhere to SLOs. Modelled configurations that meet this criteria are prioritised by their associated resource cost.

4.2.4 Execution

Our self-adaptive architecture applies the modelled configuration selected in our planning phase to the physically deployed microservice.

Our proposed architecture shares knowledge of previous deployment performance and configurations across a native Kubernetes cluster and an additional controlling component. This controlling component automatically generates probabilistic models of a deployed microservice to establish a predicted level of performance. Controlling actions are engaged to minimise the error between predicted performance and that defined within our SLOs.

4.3 Implementation

There are multiple implementation stages of our proposed self-adaptive architecture. These stages are presented in Figure 4.2. The focus of research presented within this thesis is establishing the potential for usage of our approach in predictive scaling of microservice applications within a Kubernetes Cluster.

We establish methods for classifying system behaviours and analysing performance within the context of resource utilisation. We develop an initial modelling approach to represent a Kubernetes Cluster as a Discrete Event

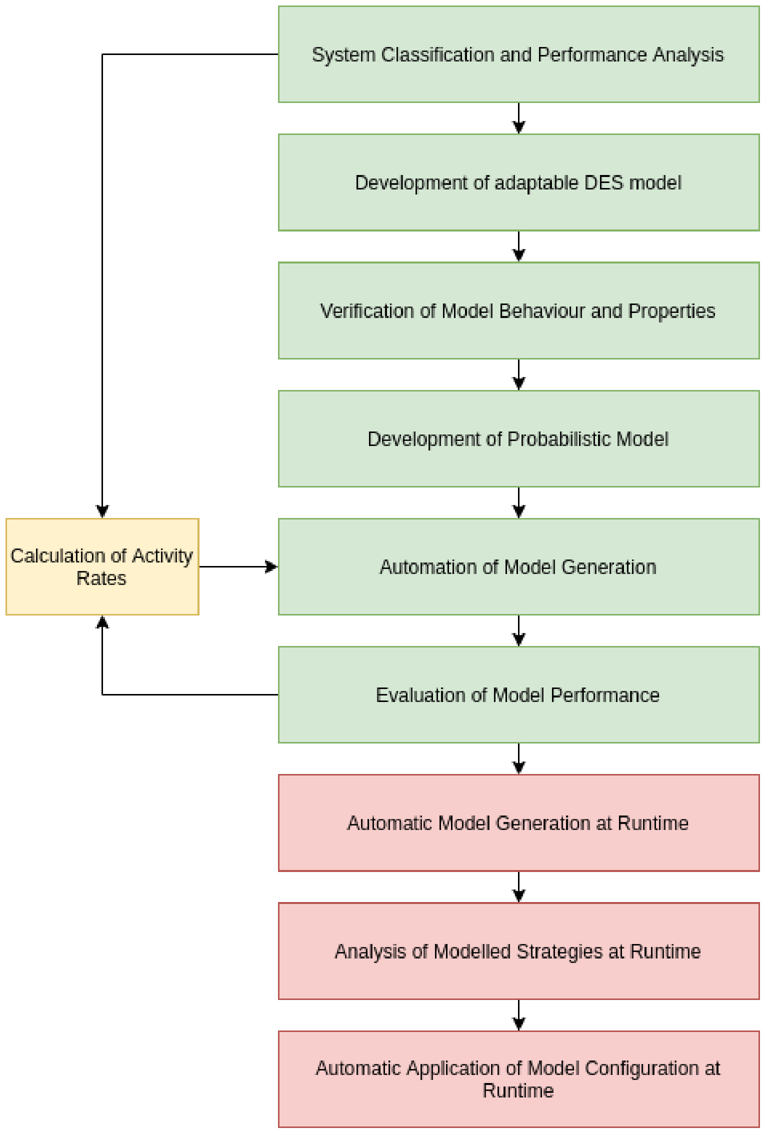

System Classification and Performance Analysis
Development of adaptable DES model
Verification of Model Behaviour and Properties
Development of Probabilistic Model
Calculation of Activity Rates
Automation of Model Generation
Evaluation of Model Performance
Automatic Model Generation at Runtime
Analysis of Modelled Strategies at Runtime
Automatic Application of Model Configuration at Runtime

Figure 4.2: Stages of Proposed Self-Adaptive Architecture

System in WATERS. We check that our DES model matches expected behaviours of our system and verify properties of our model using the WATERS framework. We transpose the logic of our DES model into a probabilistic context using the PEPA syntax. We create an automated process for generating PEPA models based on our prior observations of system performance. We evaluate the suitability of our performance model against observed behaviours of physically deployed microservices.

Due to limiting time constraints, the presented research seeks primarily to examine the potential for evaluating the performance of our modelling approach against that of deployed microservices. Future research aims to improve upon the accuracy with which performance is modelled by capturing wider behaviours that contribute to the observed SLIs of deployed applications. Furthermore, in future we aim to evaluate the efficacy of our proposed self-adaptive architecture in preventing SLO violations associated with dynamic load patterns, through the implementation and consequent evaluation of our proposed controlling component.

Chapter 5

Classifying Aspects of Kubernetes

This chapter presents characteristics of the Kubernetes container management platform. We explore components within the platform and establish behaviours. We use this analysis to produce our system models. The aim of this approach is to produce some prediction of expected performance at design time.

We model cluster behaviours with regard to request handling and underlying performance. We capture this as a client-server communication relationship. We model cluster elements within the context of incoming and handled requests across pod replicas of a deployed microservice architecture.

5.1 Cluster Elements

From the earlier background section, the reader may recall the core components of Kubernetes. Within this thesis, we view the scaling pod components of a Kubernetes cluster as a series of parallel queues. Each queue then represents the resource pool that is currently available to a deployed microservice. As the number of pod replicas scales, the resource pool scales proportionally. Resource competition between tenants can be captured by a queue that represents the shared available resource pool of a node.

5.1.1 Pods and Deployments

Pods are the unit of scale within Kubernetes. Each pod can be viewed as a self-sufficient clone of a microservice. The service can function if there is a single pod available, but increasing the number of pods will scale the processing power made available to it. A deployment describes the configuration of pods. It describes which container image a pod should pull, how much memory and CPU should be allocated to the pod, and how many replicas of the pod should be created. Deployments can be scaled by Horizontal Pod Autoscalers, discussed later in this chapter.

5.1.2 Nodes

Nodes are the physical hardware across which a cluster is hosted. Each node has a finite resource pool that pods can be deployed to. Each node has an IP address within the cluster. Pods are scheduled onto nodes with enough available resources. Each pod is assigned an IP address from a pool of addresses available for internal routing on each node. Endpoints map pods to a deployment and node.

5.1.3 Controllers

The major functionality that Kubernetes provides is its capacity for self-management. The Kubernetes architecture relies on system controllers to ensure that the observed configuration and behaviours of the cluster match those defined within the applied configuration files. In this way, a Kubernetes cluster can be viewed as a controlled system [85], seeking to match user defined configuration setpoints. This is achieved by controlling components housed within the master node of the cluster.

- **Replica Controller** checks the number of currently deployed pods against the expected number of replicas described by the relevant deployment. If there are too few pod replicas, the controller creates a pod replica. This pod remains pending until it is deployed to a node by the Scheduler. If there are too many pods, the controller terminates a pod replica.

- **Node Controller** updates the information stored within etcd to match observed configuration and resource availability of nodes within the cluster.
- **Scheduler** determines which physical nodes within the cluster have the resource capacity to accommodate the requirements of a pending pod. If and when a suitable node is found, the pod is scheduled to be deployed onto the node.
- **Endpoint Controller** keeps the endpoints up to date by creating and removing endpoints as pods are deployed and terminated. It checks endpoints within etcd against the observed configuration of the cluster. Stale endpoints are removed and updated

5.1.4 Deployed Applications

Applications are defined through deployments within Kubernetes. Deployments are created by applying configuration files to the cluster.

5.1.5 Horizontal Autoscaling

A Horizontal pod Autoscaler (HPA) allows for efficient management of resource utilisation. An HPA can be configured to work with resource values such as CPU or memory utilisation, or custom metrics such as average response time. A setpoint is established for the relevant metric. The HPA component checks the observed behaviour of pods within the associated deployment against this setpoint. Observed metrics are averaged over the number of pod replicas. A controlling action is taken that adjusts the number of pod replicas to minimise observed error. The HPA has a default buffer window of 10% to avoid rapid creation and termination of pod replicas. If a target utilisation of 70% is set, the HPA will seek to increase the number of expected pod replicas once utilisation exceeds 80%, and decrease it if it reaches 60%. This is achieved through the following equation:

$$desiredReplicas =$$

$$ceil[currentReplicas * (currentMetricValue/desiredMetricValue)]$$

Where *currentMetricValue* represents the measured value for CPU or memory utilisation and *desiredMetricValue* reflects the target utilisation goal. Autoscaling allows for achieving of SLOs under dynamic load, while reducing overhead from underutilised resources. Our modelling approach can be easily adapted to fit desired resource metrics.

5.2 Request Handling

As pods are created for a deployment, they are scheduled to nodes that have the resources necessary to run them. Exposed services are assigned an external IP that makes them accessible from outside of the cluster. As requests enter the cluster, they are handled by a load balancer. Requests are routed to the pod experiencing the least utilisation. Thus, load is balanced between pod replicas.

5.2.1 Resource Constraints

Pod Resource limits are passed to the control groups (cgroups) of the container host. The underlying linux kernel then manages resource allocation and utilisation between containers. Memory limits are treated as a hard cap. If the utilized memory of a container exceeds that of the set pod Limit, it will be killed with an Out of Memory error, and restarted based on the Kubernetes restart policy. CPU limits ensure that containers do not steal CPU time from one another as their load increases. This is achieved through throttling of running processes [1]. Resource Requests are the minimum amount of allocatable resources required for a pod to be scheduled on a given node. Resource Limits are the maximum utilization of a given resource that a pod can engage.

The way that Kuberenetes deals with resource management differs with regard to Memory and CPU.

- Memory is allocated in units of MiB or Mebibytes. A pod will be deployed and reserve its requested memory block. It can utilise anywhere up to the defined limit, at which point it is killed by the scheduler due to an Out of Memory error.

- CPU allocation is handled differently. CPU requests are dealt with in terms of Shares. A single core CPU is split into 1024 shares. A request of $CPU = "0.5"$ means that a node must have at least 512 shares available for the pod to be deployed.

- CPU limits are not dealt with in terms of shares but rather a quota over a given time period (default 100ms). If a CPU limit is set to 0.5, this means that the pod is able to use up to 500ms of CPU across concurrent requests within a 100ms time period. This means that if we have a system wherein we have 10 parallel threads, each of which engages the CPU for 100ms, we will reach the Quota in 50ms, ie 10*50ms = 500ms. The pod will experience throttling for the remainder of the CPU period. If we do not set a CPU limit, a Pod utilises as much available CPU as it requires.

Our modelling approach requires that we identify the limits of our deployed application with regard to concurrent request capacity. CPU limits are defined in terms of millicores of CPU within our deployment configurations. As containers are created, the Docker engine inherits the defined CPU limit, and contextualises it as a measure of CPU shares. Each running container on a host OS will have a measure of CPU shares assigned to it. To achieve this, Docker determines CPU allocation at runtime using two metrics. CPU quota and period. The CPU period is set as $1/10^{th}$ of a second. Quota is the amount of CPU time that the running container can utilise within a single period, and is inherited from the deployment configuration. If a container exceeds its quota within a period, then the running processes will be throttled until the next period. Increasing the number of concurrent of CPU intensive tasks therefore leads to increased delays for the completion of all tasks, as they all encounter throttling proportional to the number of tasks being handled.

5.2.2 Internal processes

Internal container processes contribute to resource utilisation. A recognised contributing factor to CPU utilisation in Java applications is Garbage Collection. As memory is utilised, GC processes are engaged to remove state memory objects. CPU resources are engaged during this process. Internal memory management therefore leads to an increase in observed CPU utilisation, thus impacted performance of deployed applications. [71]

Our modelling approach captures the interaction between components within a Kubernetes cluster. We use our knowledge of microservices and the Kubernetes platform to establish parallel queues that represent the capacity of deployed pod replicas and node resources. Synchronous composition and logical guards are employed to capture the behaviour of controlling components. Classification of deployed microservice applications is performed to capture the tailored performance of each microservice application. We use this knowledge to produce performance models of the associated system.

Chapter 6

Experimental Setup

Our automated modelling required two sets of experiments. The first set, concurrency testing experiments, were for classification of observed behaviours. The second set, load testing experiments, were for evaluation of our produced models. These experiments used a Kubernetes cluster on which an opensource stress-testing application was deployed as a microservice architecture. Experiments were run across three homogeneous local clusters. Clusters were each run on identical machines with Ubuntu 18.04.4. Each machine had 16GB of RAM and an Intel i7-8700CPU with 6 cores. Clusters were hosted within VMs and managed by Minikube. Each VM was assigned 8GB of RAM and 3 CPU cores. Each identical cluster was an experimental environment for our deployed stressing application. Apache JMeter was used to drive load to local clusters, while the status and resource utilisation of cluster components were recorded through *kubectl* commands within a bash script.

6.1 Minikube setup

Each cluster was hosted using a VirtualBox wrapper. The clusters were thus established within identical VM environments with specific resource allocations. Each cluster then contained a stressing namespace to which applications were deployed. The Docker image that our deployment configuration file pulled was installed into each VM's local docker environment. This removed any network delays when restarting pod replicas. Our Minikube clusters had the *metrics-server* and *metallb* addons enabled for access to pod metrics and load balancing respectively.

6.1.1 Deployed Application

The application deployed for testing within this thesis was an open source stress-testing Docker image created by Flavio Stutz[1]. This application was chosen due to its customizable request load and known behaviours. The reader may recall from previous sections that our modelling approach is tailored toward microservices with low individual pod capacity. Our chosen application image simulated request load based on user defined parameters. When sent a request with a specified number n of bytes and duration t. The application created an array of size n and then slept for the specified duration before releasing the reserved memory. We configured pods within our deployments to pull this image. We assigned different resource allocations to different deployments. Load testing of our deployed application was performed using Apache JMeter. We observed the behaviour of pods within our deployments under different request load patterns. Testing was performed across multiple deployment configurations.

6.1.2 Load Balancing

Kubernetes supports the inclusion of load balancing, but relies on an external load balancing service to manage routing decisions within the hosted environment. When a deployment is exposed as a load balancing service, the external load balancer assigns an IP address from an available pool. This IP serves as the access point for external requests that are made to the deployed application. This IP address is mapped to the internal addresses of pods within the deployment through endpoint objects. As requests enter the cluster, the load balancer will evaluate the load across pods within the cluster, and priori-

[1]https://github.com/flaviostutz/web-stress-simulator

tise pods for routing based on observed utilisation. Minikube is designed as a localised testing environment for evaluating a Kubernetes cluster. Default Minikube configuration does not provide a load balancing service, as such, we used the Minikube *metallb*[2] addon as our load balancer. This addon has three main components:

- Configuration determines the IP address ranges that you want the load balancer to be able to assign to services as they are exposed.
- A *Controller* deployment assigns IP addresses to services as they are exposed.
- A *Speaker* daemonset that ensures that services are reachable.

We configured metallb to assign a consistent external IP address across all of our experimental environments. When we exposed our deployments as load balancing services, this IP was then mapped to the pod endpoints that the service was linked to. As requests entered the cluster, they were routed to an appropriate pod based on the average observed load.

6.1.3 Scaling

Horizontal autoscaling of our application used CPU utilisation for scaling decisions. Our chosen application was Java-based. JVM based applications make use of a memory heap. As memory objects are created, they are added to the heap. When heap utilisation exceeds default thresholds, Garbage Collection processes are engaged to free objects[70]. Memory-based scaling was redundant for our deployed application as measured heapsize was not reflective of active utilisation. Experimental tuning of parameters to increase the rate at which GC processes were engaged resulted in greatly increased CPU utilisation and consequent throttling.

[2]https://metallb.universe.tf/

6.2 JMeter Scripts

JMeter Scripts were used to fire requests to the deployed System. Requests consisted of a specified number of bytes and a time. The application handled these requests by generating an array of characters to use the number of bytes specified, and then slept for the time duration. This established a memory cost of servicing requests. We made use of two testing scenarios. Concurrent request testing was used to establish our CPU coefficient and limits of our system. Load testing under different incoming request rates was used to evaluate the accuracy of our modelled predictions. Concurrent testing scenarios involved JMeter maintaining a constant number of active requests for a given duration. The desired number of active requests wass created, and a new one was sent only when a response was received. We thus measured the resource cost of request handling, and the limits of our deployed Pods. Scripts were fired sequentially, with a concurrent request target increasing from 1-15. Deployments were restarted between each test and given time for the containers to fully spin up to avoid the behaviours of one test round impacting results of another. Concurrent testing was used to obtain information required for model generation.

Once we acquired the information required to model our system, we compared the predicted performance of our models with the observed performance of our system under further testing scenarios. This further testing involved firing the same requests as concurrent testing, but with a focus on increasing the rate at which requests enter the system, rather than the number of requests being handled concurrently. These scripts were also executed sequentially, with the incoming request rate increasing from 1-10 requests per second.

For concurrency testing, we repeated our experiments across deployment configurations of 1, 2, and 3 pod replicas. We repeated our further experiments across these same configurations, along with configurations that scale the number of pod replicas from 1-2, 1-3, and 2-3.

6.3 PodMetrics

As Jmeter scripts were run to stress the system, information was obtained through the *Kubernetes Command-line Tool* (kubectl). As tests were run, a bash script was utilised to make kubectl requests at regular intervals. Information was thus recorded regarding the behaviour and resource utilisation of pods during testing. The kubectl commands were:

- *kubectl top pods -n stressing*, which returned the resource usage of pod replicas deployed within the stressing namespace. These metrics were obtained through the metrics-server addon, which scrapes resource usage of pods with a default resolution of 60 seconds.
- *kubectl get pods -n stressing* returned the number of pods within the stressing namespace and the associated state of each pod. These states were *RUNNING*, *CREATING*, or *CrashLoopBackoff*.
- When scaling was engaged, we also queried the associated HPA for information regarding resource utilisation and expected Pod replicas.

6.4 Postprocessing

Initially, results were not formatted in a way that was suited to any form of analysis. Hence, techniques were utilised to extract relevant information and present it in a format that was more convenient for performance evaluation. The process through which we attained and analysed our data is as follows: Pod level data generated by our bash scripts was saved into csv files. This provided us with a record of status and resource utilisation for each pod as our system was stressed within each experimental run. We utilised several Python scripts to analyse our observed results. These scripts stripped the data down to relevant information, which was then analysed to evaluate average performance across different load patterns. This data was then graphed using the pyplot package from the matplotlib library. For Cluster data, we graphed pod metric usage along with observed pod replica counts. During this postprocessing, classification of other system aspects was also performed. This classification produced metrics that were used to calculate transition rates within the generation of performance models. The next implementation stage of our self-adaptive architecture is the automation of this classification process at runtime. Produced graphs can be seen within the later discussion chapters. Results of testing that involved concurrent requests was the basis of our model generation, while results of testing that involved increasing load were compared against our modelled results. The data from each round of testing was further examined to provide insight into the performance results that are produced.

Chapter 7

Tuning Model Parameters

Within this chapter we present our process for classification of parameters used for the generation of our system models. We outline the various request and deployment configurations across which we test our modelling approach, and discuss the classification of our deployed microservice architectures.

7.1 Variations to Experimental Context

Verifying the suitability of our modelling approach involved testing across multiple environmental contexts. To create such models, we observed the reaction of our deployed applications across various request loads. We evaluated the performance of our modelled systems against the observed behaviour of deployment scenarios under various load conditions. We compared observed behaviours of deployment scenarios under differing request loads. We also compared the performance of our application using different deployment scenarios with the same request load.

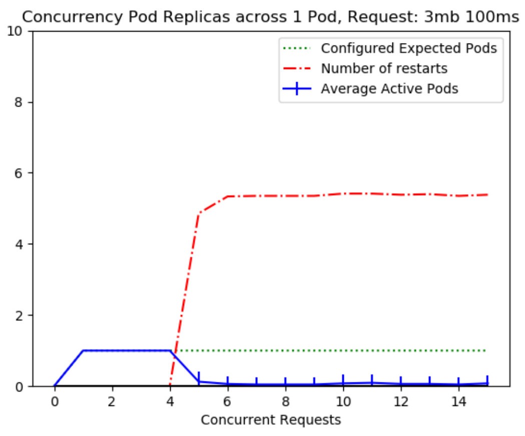
7.1.1 Deployment Configuration

We evaluated the suitability of our modelled approach across scenarios with varying resource allocations associated with our deployed application. Each deployment configuration was stressed with the same baseline request load of

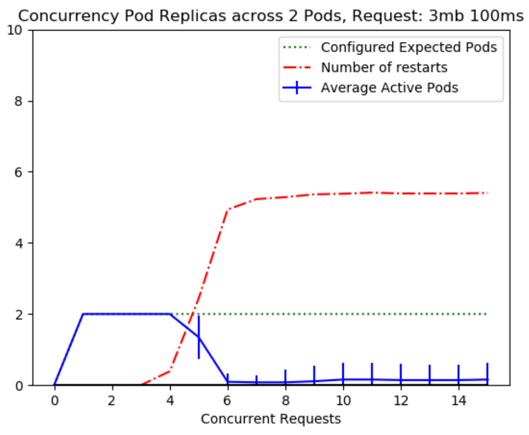
3MB with duration of 100ms. We established different CPU and memory allocations across our deployment scenarios. CPU limits were not enforced on deployment scenarios within concurrency testing. CPU utilisation was able to increase up to the maximum available in the namespace, being 3000 millicores. As such, analysis of concurrency testing established an estimated CPU utilisation cost that was not impacted by throttling behaviours. The difference between observed performance across scenarios were attributed to memory allocation, with all other variables controlled.

- **Scenario A** - 150MiB Memory
- **Scenario B** - 250MiB Memory
- **Scenario C** - 350MiB Memory

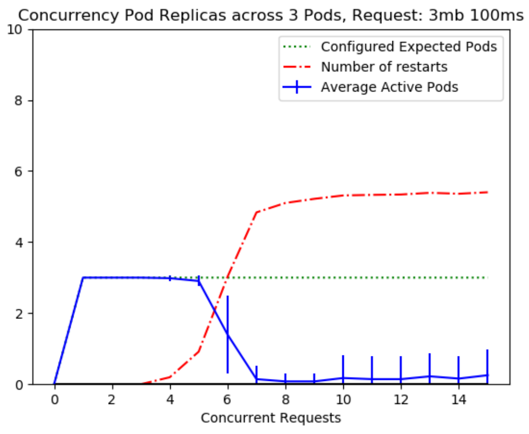
Comparison between observed behaviours of each deployment scenario provided insight to the impact of memory availability on concurrency handling. Varying the memory limit assigned to pods within our scenarios directly impacted the maximum concurrent request capacity.



(a) Replica Status across 1 Pod, Deployment Scenario A

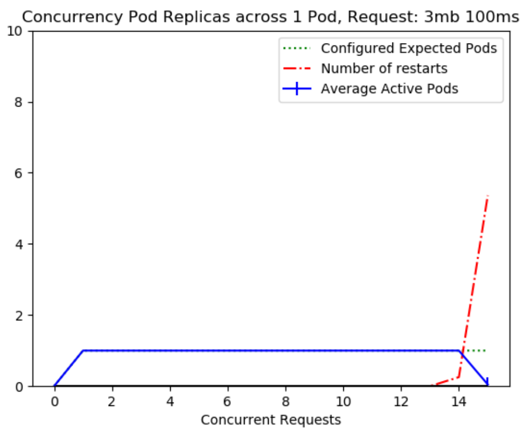


(b) Replica Status across 2 Pods, Deployment Scenario A

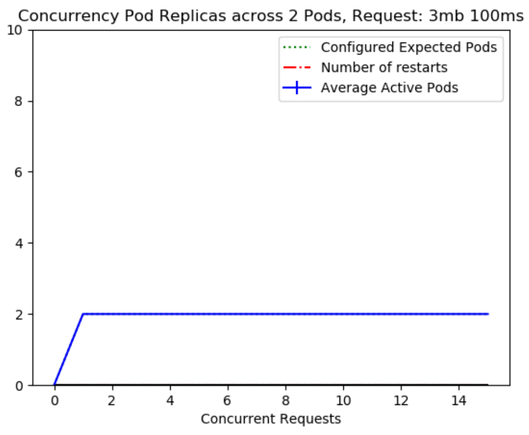


(c) Replica Status across 3 Pods, Deployment Scenario A

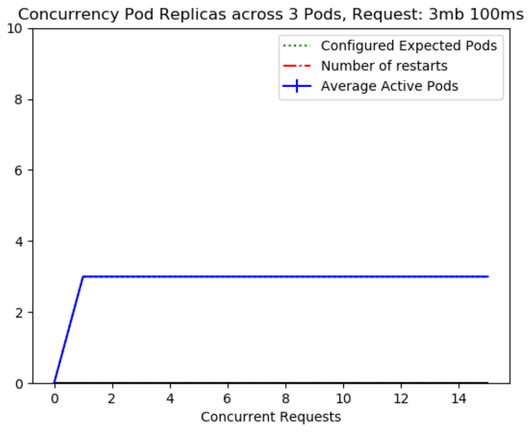
Figure 7.1: Observed Replica Status for Deployment Scenario A



(a) Replica Status across 1 Pod, Deployment Scenario C



(b) Replica Status across 2 Pods, Deployment Scenario C



(c) Replica Status across 3 Pod, Deployment Scenario C

Figure 7.2: Observed Replica Status for Deployment Scenario C

Comparison between the graphs presented within Figures 7.1, 7.2 and 7.7 illustrates the impact of available memory on concurrency limits of our system. It is worth noting that the difference in concurrent capacity did not match expectations from our knowledge of request load and allocated resources. Requests used to generate the presented Figures use the same load of 3MB with duration of 100ms. Given that the deployed application was constant across deployment scenarios, and that memory usage was driven by our request pattern, we anticipated that the concurrent capacity of pod replicas would increase linearly with the increased resource allocation.

$$\Delta Lim_{Concurrent} = \frac{Pod_{mem}}{request_{mem}} + C \tag{7.1}$$

Where C was the memory utilisation of the running container image. In the case of Figures 7.7c and 7.1a, we anticipated an increase in capacity that reflected the cost of each request against the increased capacity, i.e.,

$$\frac{250 - 150}{3} = 16\frac{2}{3}$$

We observed an increase 3. We attributed this discrepancy to the nature of JVM heap management.

Heapsize does not directly correlate to the measured concurrency level of an application. Instead, memory is allocated within the heap as processes are engaged. Once processing completes, this memory remains allocated until Garbage Collection (GC) processes are engaged[70]. As the heap approaches its maximum size, GC engages CPU resources to check for and remove stale objects in memory.

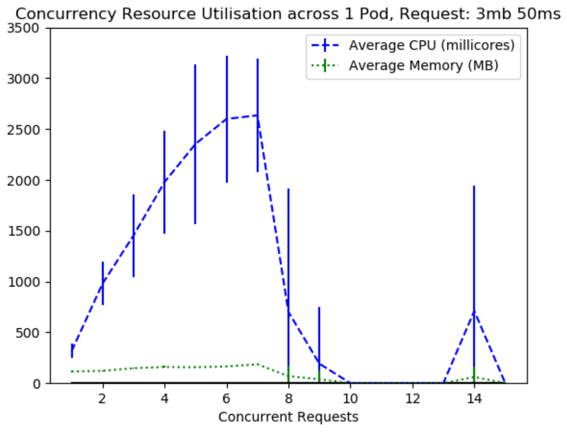
GC behaviours contributed to the observed capacity of pods across our deployments. Stale objects were not freed from memory until GC was engaged. Memory utilisation from previous requests reduced the memory available for incoming requests. Consequently, the observed failure point of an individual pod did not correlate to the relative cost of requests against available resources. As memory allocation increased, the efficiency of GC also increased. GC processes are triggered when the used space in a heap exceeds a threshold. As memory allocations were increased across our deployment scenarios, GC processes were encountered less often, and each successful GC run freed up a larger amount of memory proportional to the increased allocation. As such, we observed a dramatic increase in pod replica concurrency limits as the configured memory allocation was increased.

7.1.2 Request Configuration

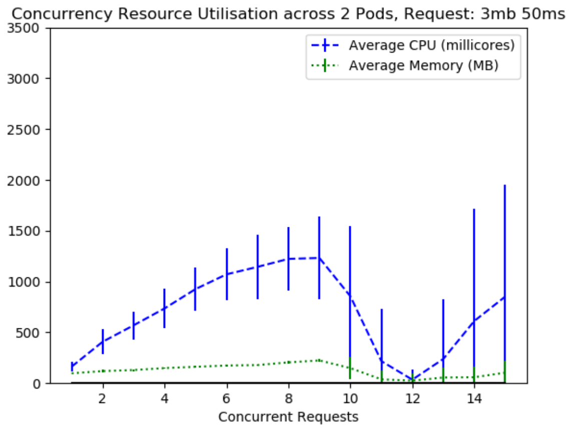
We used configuration of our application in scenario B to classify behaviours of our system under different load conditions. Previous Figures 7.9a and 7.7 depict the behaviour of pod replicas within this scenario when faced with requests of size 3MB and duration of 100ms. We utilised 5 different request loads across this same system.

- 3MB with duration of 50ms
- 3MB with duration of 100ms
- 3MB with duration of 200ms
- 1MB with duration of 100ms
- 5MB with duration of 100ms

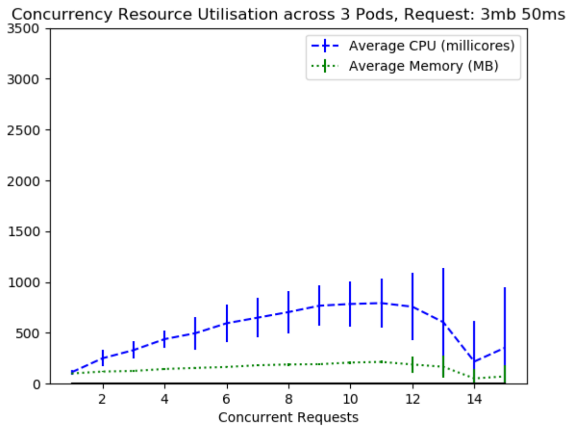
Results returned by these experiments served two purposes. The first was to establish the impact of varying request size and duration on the observed behaviour of our system. The second was to verify the accuracy with which our modelling approach is fitted to a system under changing conditions. Both of these aspects contributed to the answering of our research questions, specifically the validity of our approach for modelling the behaviours of a Kubernetes cluster. Figures 7.3 and 7.5 depict the resource utilisation and status of pods within our system for Scenario B under requests with a duration of 50ms. Figures 7.9, 7.7, 7.4, and 7.6 depict the same scenario and request size, with durations of 100ms and 200ms.



(a) Metrics across 1 Pod, 50ms Request Duration

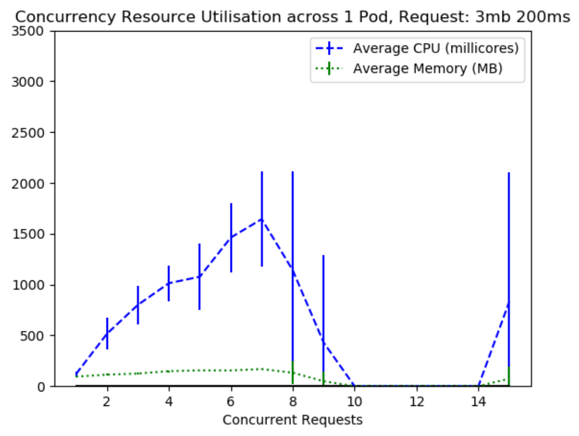


(b) Metrics across 2 Pods, 50ms Request Duration

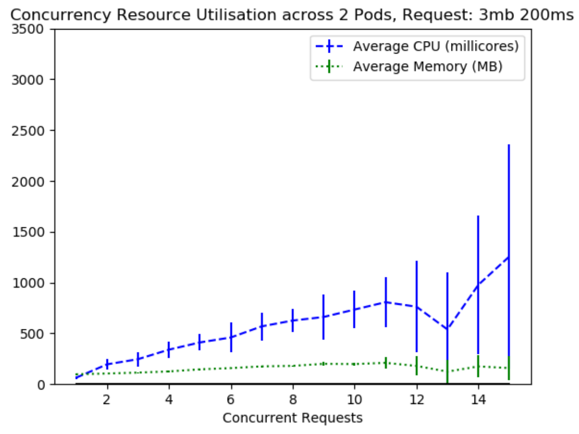


(c) Metrics across 3 Pods, 50ms Request Duration

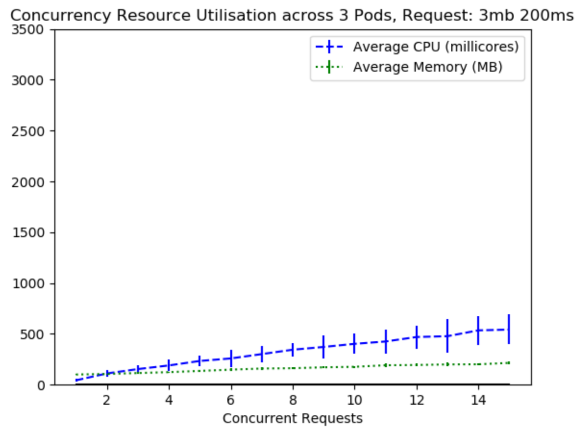
Figure 7.3: Observed Resource Utilisation for Deployment Scenario B, Request duration 50ms



(a) Metrics across 1 Pod, 200ms Request Duration



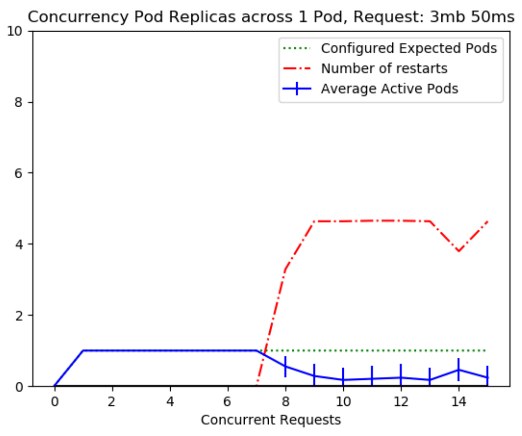
(b) Metrics across 2 Pods, 200ms Request Duration



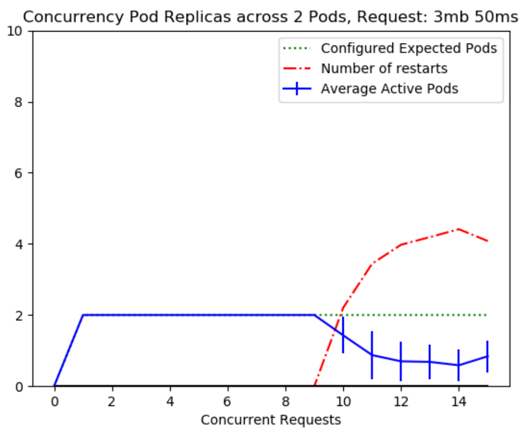
(c) Metrics across 3 Pods, 200ms Request Duration

Figure 7.4: Observed Resource Utilisation for Deployment Scenario B, Request duration 200ms

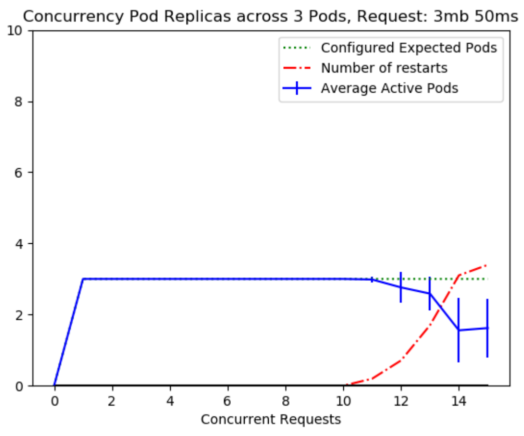
Comparison between utilisation across the different durations showed what we can classify as an inverse relationship between request duration and CPU utilisation within concurrency testing. We found an explanation for this within our stressing methodology. The reader may recall from previous discussions that our application handled memory stressing requests by creating an array of size n bytes, after which it slept for t ms before releasing the assigned memory. As such, the CPU load of our application was almost entirely due to creating and freeing these arrays in conjunction with receiving and replying to requests. As concurrency stressing required that requests were replaced as soon as they were handled, a decreased request duration resulted in an increased rate at which requests entered the system. Thus, the amount of CPU time required to maintain a level of concurrency also increased.



(a) Replica Status across 1 Pod, 50ms Request Duration

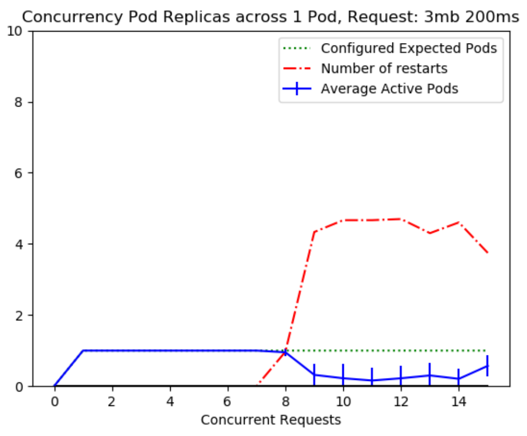


(b) Replica Status across 2 Pods, 50ms Request Duration

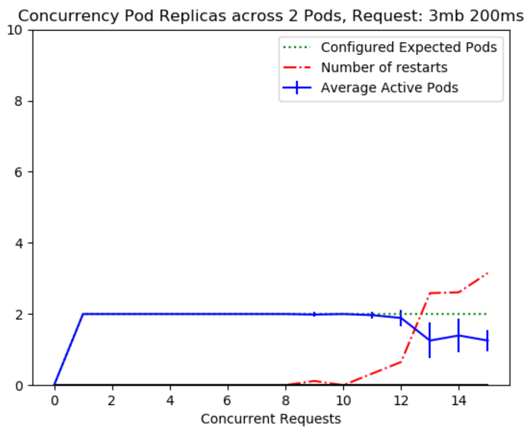


(c) Replica Status across 3 Pods, 50ms Request Duration

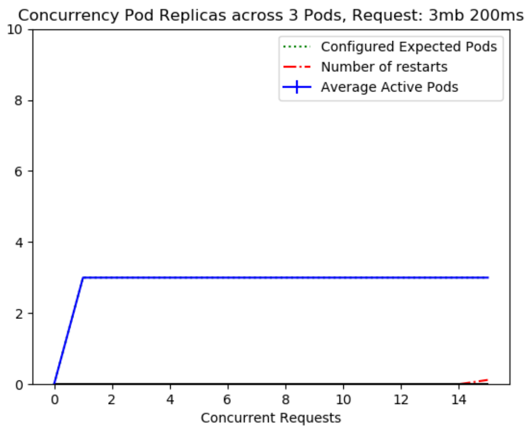
Figure 7.5: Observed Replica Status for Deployment Scenario B, Request Duration 50ms



(a) Replica Status across 1 Pod, 200ms Request Duration



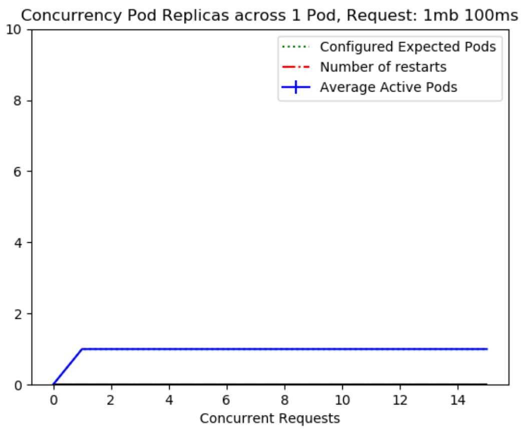
(b) Replica Status across 2 Pods, 200ms Request Duration



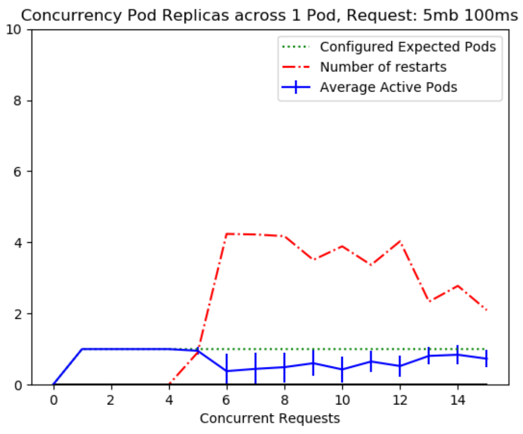
(c) Replica Status across 3 Pods, 200ms Request Duration

Figure 7.6: Observed Replica Status for Deployment Scenario B, Request Duration 50ms

Examination of pod status presented in Figures 7.5, 7.7 and 7.6 indicates that our shorter request duration also resulted in earlier observations of failure as the number of pod replicas in our deployment increases. This relationship can be explained by the inaccuracy displayed by our load balancer. As previously discussed, lower request durations resulted in higher incoming request rates when our stressing involved concurrency. The result of this is that our load balancer was engaged more often, which further exacerbated the impact of incorrectly routed requests.



(a) Replica Status across 1 Pod, 1MB Request Size



(b) Replica Status across 1 Pod, 5MB Request Size

We established the impact of request load via a comparison between observed failure points of pods within deployment scenario B, using request duration of 100ms and size of 1mb, 3mb, and 5MB respectively. From these produced results, we observed that the size of our request correlated to the concurrency level at which we encountered a pod failure. This was consistent with our expectations of behaviour based on the OOM condition that led to pod failure. As the memory utilisation increased due to request size, we expected a lower capacity for request concurrency.

7.2 System Classification

The following aspects of pod replicas within our deployed microservice architecture were known from deployment configuration files at design time:

- Memory requests and limits.
- CPU requests and limits.
- The minimum and maximum number of pod replicas.
- The utilisation threshold for horizontal autoscaling actions.

Classification of the following aspects was achieved through concurrency testing:

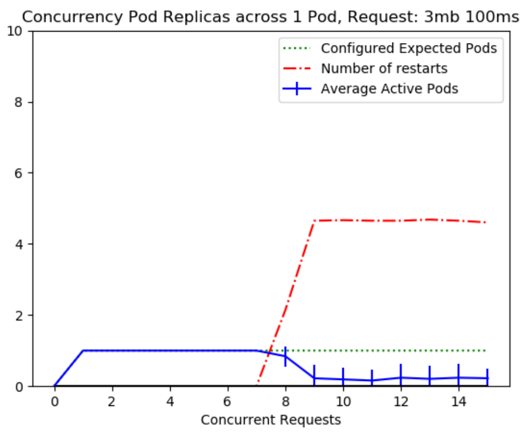
- The average CPU utilisation for a single pod replica across concurrent request levels.
- The concurrency limit of a single pod replica with regard to pod failure.

Concurrency testing utilised multiple experimental rounds with incrementally increasing levels of active request concurrency to benchmark our systems. Python scripts were used to automate the classification process. These scripts analysed test results and pod information across repeated rounds of testing. Such scripts return the average CPU utilisation and concurrency limit of pod replicas within a deployment scenario. We anticipate that future development of our architecture will incorporate this process at runtime.

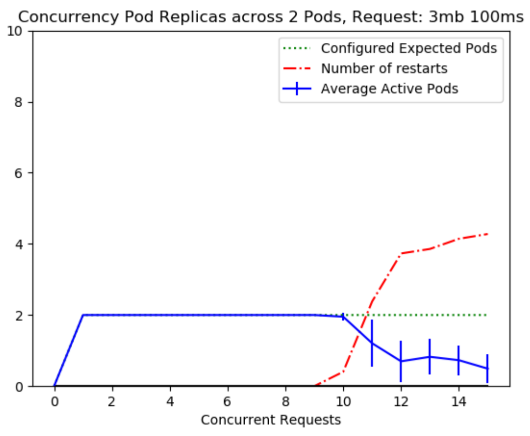
7.2.1 Limits

For the purpose of identifying processing limits, we analysed the state of pod replicas within our deployed application across different levels of concurrency. Figure 7.7 shows the expected number of active replicas against the observed count for our B scenario. The X axes within the captured graphs represent experimental rounds. Each experiment lasted 10 minutes, within which time a stable concurrent load was generated. Each integer along the x axes corresponds to the level of concurrency within that experimental round. We inferred three aspects of our system's behaviour from the graphs within this figure. Figure 7.7c indicates that our application began to fail at the point where a single pod reached a concurrency level of 8 active requests. Figures 7.7d and 7.7e show that this failure point did not scale linearly with relation to the number of deployed pod replicas. Instead, we saw a staggered increase in the limit for concurrent request handling. Figures 7.7c and 7.7d show recorded attempts to restart pods quickly increased and then plateaued.

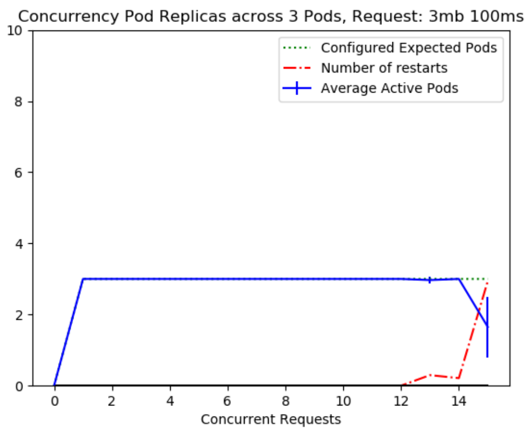
The cause of failure in our system can be deduced from Figure 7.9, where we see that the concurrency level associated with failure was also the point at which the average memory usage of each pod exceeded the configured limit of 250MiB. This was further supported by the observed status of deployed pods, found within the experimental results in our appendix. When we examined the status of pods within our cluster, we saw a failure with an *OOM* error, indicating that pods exceeded the resource cap defined within their internal CGroups.



(c) Replica Status across 1 Pod

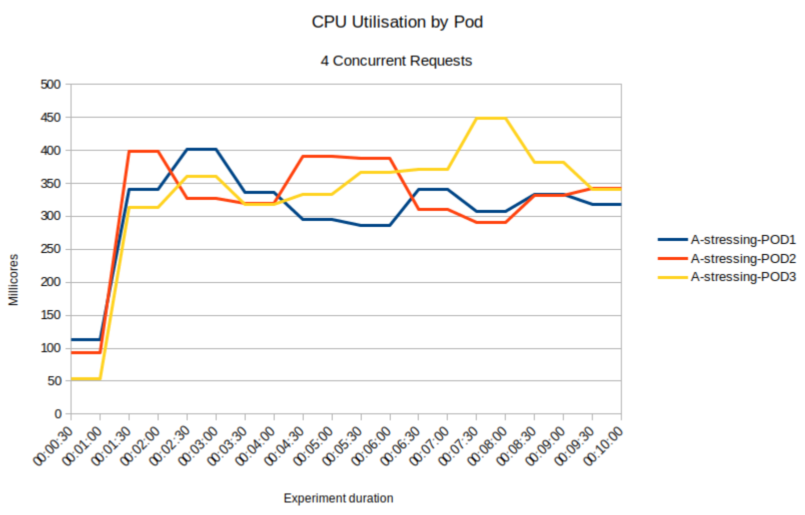


(d) Replica Status across 2 Pods

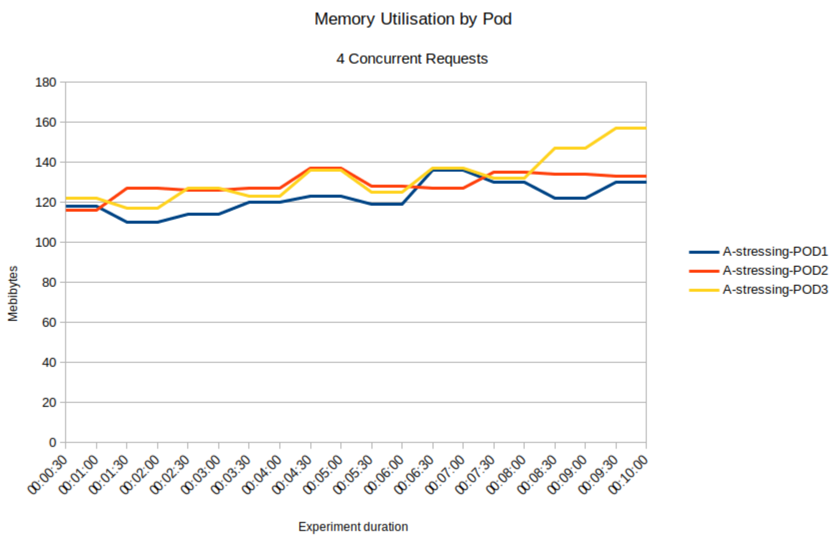


(e) Replica Status across 3 Pods

Figure 7.7: Observed Replica Status of Pod Replicas in Deployment Scenario B



(a) CPU Utilisation Across 3 Pods



(b) Memory Utilisation Across 3 Pods

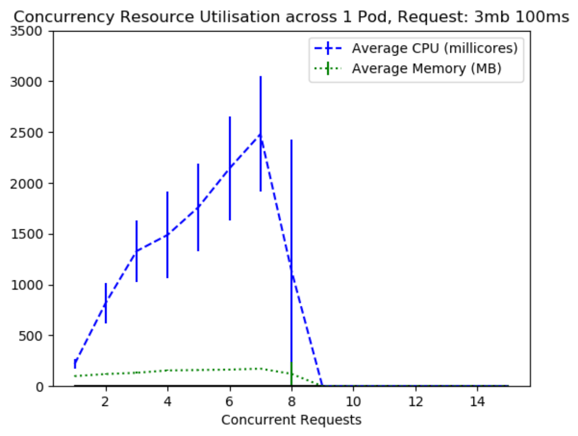
Figure 7.8: Balanced Resource Utilisation of Pod Replicas in Deployment Scenario B

The staggered scaling of our observed failure point as the number of deployed pod replicas increased can be partially explained by the imperfect nature of load balancing within our cluster. In Figure 7.8, we see the observed resource utilisation of each active pod in our deployment. These graphs were generated directly from our first round of concurrency testing, with three deployed pod replicas and a concurrent request level of 4. The subfigures show that while the load was similar across pod replicas, there were still differences in observed utilisation from one pod to the next. As a result, it was possible for pods to experience a spike in utilisation that exceeded their configured resource limits, and thus fail as the generated load approached the total system capacity. As one pod failed, the utilisation of other pods increased as they were routed extra requests. The result of this is that we experienced a cascading failure of all pod replicas within the deployment. The impact of load balancing efficacy was exacerbated within our system due to the low concurrent handling capacity of each single pod.

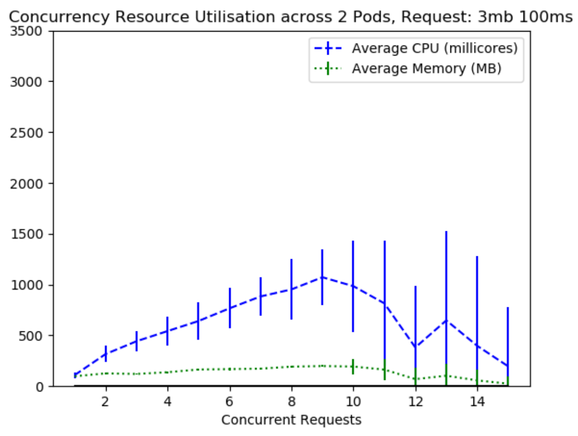
Kubernetes makes use of a staggered delay between each attempted pod restart. The result of this is that each failure within an experimental round resulted in significant downtime for the failed pod, while the number of attempted restarts was constrained by the duration of the experiment.

7.2.2 CPU Cost

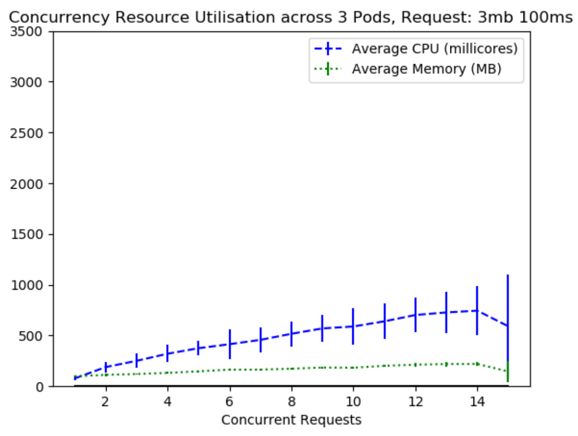
From Figure 7.7, we established the concurrency limits of a single pod replica. Further examination of experimental results established that request handling capacity was impacted by memory allocation. Having classified the maximum capacity of pod replicas, we then identified expected throughput. Our justification for this was that as the maximum throughput of a system is increased, the likelihood of it exceeding its concurrency limit for a given rate of incoming requests is decreased. The reader may recall that throughput in Kubernetes is limited by CPU throttling. As such, we experimentally measured a CPU coefficient for generation of our modelled service rate.



(a) Resource Utilisation of a Single Pod Replica in Deployment Scenario B



(b) Resource Utilisation Across Two Pod Replicas in Deployment Scenario B



(c) Resource Utilisation Across Three Pod Replicas in Deployment Scenario B

Figure 7.9: Resource Utilisation Pod Replicas in Deployment Scenario B

Figure 7.9 presents the averaged resource utilisation across different numbers of pod replicas for our deployment scenario B. The reasonably consistent standard deviation observed prior to the failure point across graphs within this figure reinforces previous observations regarding load balancing efficacy, showing that observed load across experimental repeats fell within a consistent range of average utilisation. Within the process of generating Figure 7.9a, an average of the CPU utilisation against concurrent request count was taken from each level of concurrency prior to failure. The average of these recorded values became our CPU coefficient. For scenario B, we found that maintaining a level concurrency for a single request utilised on average 375millicores of CPU. This value, along with the concurrent handling limit of a single pod, were used in conjunction with other known information about the deployed configuration of our microservice application to produce system models.
Identification of system capacity and utilisation was automated with the use of python scripts. Other information was taken from deployment configurations. We anticipate that future development will automate both of these processes by scraping runtime metrics of microservice applications.

From our concurrency testing, we established performance parameters for applications across our deployment scenarios. The key aspects identified were the concurrency limit of pod replicas within a given deployment, and the average CPU cost associated with maintaining a level of concurrency. Such processes fell into the system classification step of our self-adaptive strategy outlined in Figure 4.2. The classification information from each of our deployment scenarios was used in the next steps of our strategy, the development of our system models.

Chapter 8

Modelling Approaches

This chapter explores our modelling approaches using both WATERS and PEPA. We capture expected behaviour of components as the cluster evolves over time. We first establish a visual representation of system behaviour that can be adjusted to fit different cluster configurations. This is achieved using automata and variable tracking within WATERS. We verify that our model adheres to expected behavioural properties, such as controllability and non-blocking nature. We transpose the controlling logic of our model into a generator script that produces a process algebra representation of the same model within a PEPA syntax. Figure 8.1 depicts our hybridised modelling approach. This approach takes advantage of the ease of use and inference provided by the WATERS Graphical User Interface, while also allowing us to evaluate expected performance using the more convoluted PEPA syntax. The logic of control plane components such as autoscaling and a deployment controller is captured within guard statements in WATERS automata. PEPA does not afford us with this functionality, as such it is instead implemented within the logic of the model generator found within the accompanying appendices to capture this same behaviour. Each PEPA component is generated with defined behaviours. Control of the overall cluster is achieved by synchronising components on relevent actions.

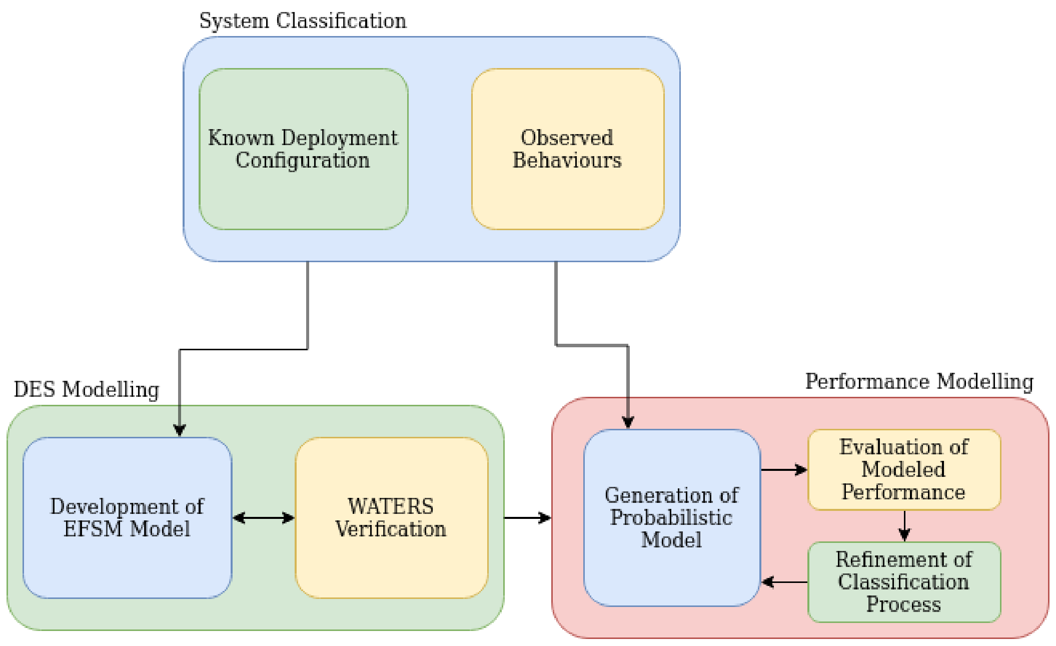

System Classification
Known Deployment Configuration
Observed Behaviours
DES Modelling
Development of EFSM Model
WATERS Verification
Performance Modelling
Generation of Probabilistic Model
Evaluation of Modeled Performance
Refinement of Classification Process

Figure 8.1: Our Hybridised Modelling Approach

8.1 Workload

The workload associated with a given service comprises requests that are handled by multiple pods within the relevant deployment. Each request will have an assigned number of bytes of memory that it will engage during its completion, and an assigned duration for which it should engage these resources. The resource cost to memory is constant during the requests' completion. We identify limits to our system capacity through concurrency testing outlined in our experimental methodology. These limits are the point at which our system is pushed over its memory resource allocation, and we encounter a pod crash with an *Out of Memory* (OOM) error. The maximum capacity is implemented as an integer value within our model. WATERS tracks this against the observed request count and enables failure transitions when the limit is passed. Our PEPA generator generates states representing a request queue up to the maximum capacity, after which failure actions are enabled. WATERS does not allow for tracking of request rates, but is used for verification purposes instead. We evaluate expected performance within PEPA, where we are able to vary the rate at which requests enter our system.

8.2 Component Classification

We model the internal components of the Kubernetes platform as $M|M|k$ queues that synchronise over common events. We denote a queue comprising request handling times using standard notation [38]. The first M indicates that the inter-arrival time of requests are *memoryless*. The second M indicates that the service times are *memoryless*. This means that neither of these behavioural aspects of our system are impacted by past behaviour. These characteristics are then shared across a pool of k servers. Within the context of this research, k can be viewed as the number of parallel pod replicas. Each Deployment can be viewed as parallel $M|M|k$ queues of length L_{pod}. Deployment queues with different k values can exist in parallel, linked via a deployment, termination, or failure action. Incoming and handled requests synchronise with transitions within the deployment. We represent node resources as two queues, for memory and CPU respectively. By synchronising deployment and termination actions with those of the deployment queues, we are able to model total node resource availability as our system evolves. The queue length L_{node} represents total resource capacity, and is determined by the Greatest Common Divisor, GCD, found between the node's assigned resources and the resource cost associated with each kind of pod. Each pod creation will cause the queue position to increase by a scale factor that is proportional to the cost of the created pod.

$$L_{node} = \frac{Assigned_{node}}{GCD(Assigned_{node}, Assigned_{podi}, Assigned_{podj}, ..., Assigned_{podn})} \tag{8.1}$$

Where *Assigned* refers to assigned CPU or memory resource respectively. By synchronizing on pod creation and termination, the node queue can then block scaling actions of deployments on the node where there are not adequate resources available. Thus, we can build a basic system model from our constituent components through the formalisation of a system equation to represent the synchronous composition of system components across shared events.

We model pods of different deployments in parallel with each other, while they all synchronise with the resource queues over their relative transitions.

$$(Pods_j[n] \bowtie Pods_k[m]) \underset{activate_{jk}, terminate_{jk}}{\bowtie} Node_{resources} \tag{8.2}$$

Where $Pods_j[n] \bowtie Pods_k[m]$ represents the parallel relationships between n pods of deployment j and m pods of deployment k. *activate* and *terminate* refer to the actions taken to establish or destroy a running pod from a given deployment. $Node_{resources}$ is the synchronous product of our node CPU and memory resource queues.

$$Node_{resources} = Node_{mem} \underset{activate_{jk}, terminate_{jk}}{\bowtie} Node_{cpu} \tag{8.3}$$

The pod queue position of a given deployment will change based on internal request arrival and completion actions. The number of pods within the deployment may increase or decrease as load varies. Pod creation, termination, and failure events are synchronised with the node queues which ensure that resource constraints are obeyed. Such an approach allows for the capturing of behaviour of controlling components within the master node without explicit modelling, as the behaviours are controlled via synchronisation across components, where actions must be possible across all relevant components before they can be fired.

8.2.1 Pod Queues

The queue limit for our Pods for is determined by running initial concurrency stress tests on deployed microservice applications. Analysis of produced results reveals the maximum number of concurrent active requests that a single pod replica can service before it exceeds its memory limit and is killed. CPU utilisation is averaged over the all of the experimental rounds prior to pod failure to attain the *CPU coefficient* associated with concurrent levels of request handling.

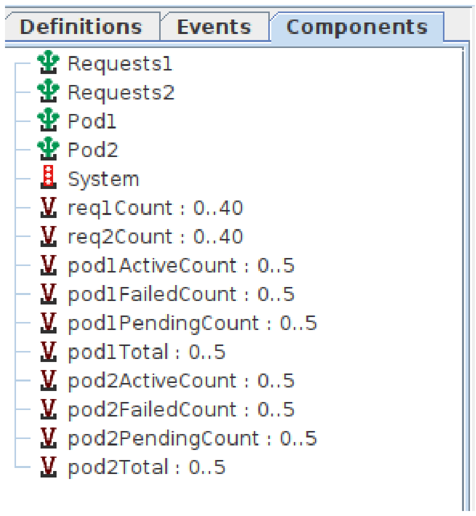
8.3 WATERS representation

This thesis makes use of modelling within both WATERS and PEPA contexts to make the most of the analysis offered by both. Through this approach, visual representations of system components can be presented through WATERS along with functional verification of model properties. The logic of this model can then be applied to generate the language of our PEPA model. This PEPA model is then used to evaluate expected performance.

8.3.1 Automata

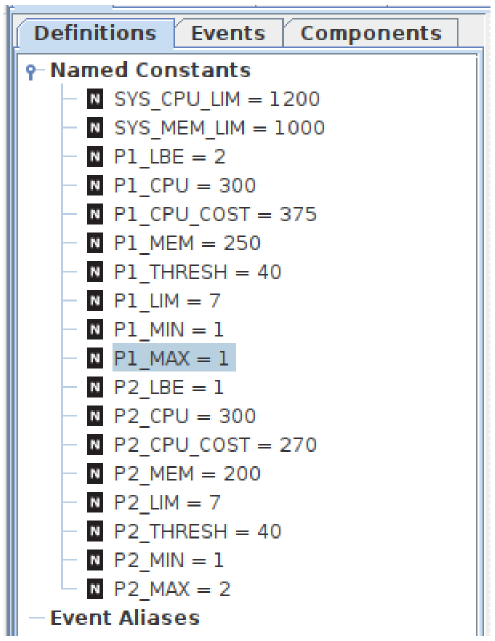
WATERS modelling of cluster behaviour is achieved through a base model that can be quickly adjusted to meet changing deployment configurations and system constraints. Each deployment is modelled by a scaling pod component, which is synchronised with a plant component defining request behaviour and a system controller that ensures that node resource constraints are adhered to.

8.3.1.1 Components and Constants



- Requests1
- Requests2
- Pod1
- Pod2
- System
- req1Count : 0..40
- req2Count : 0..40
- pod1ActiveCount : 0..5
- pod1FailedCount : 0..5
- pod1PendingCount : 0..5
- pod1Total : 0..5
- pod2ActiveCount : 0..5
- pod2FailedCount : 0..5
- pod2PendingCount : 0..5
- pod2Total : 0..5

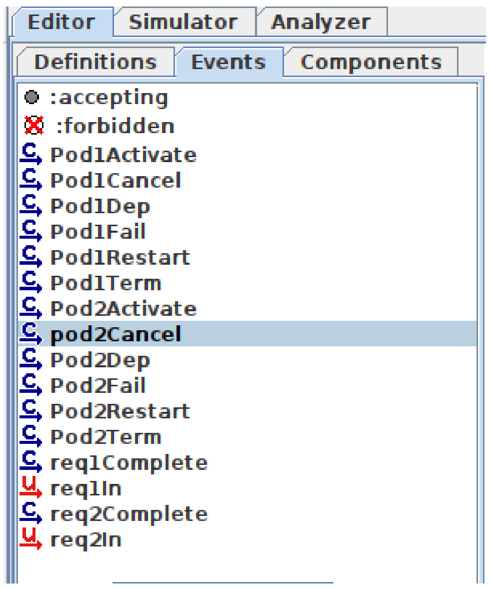
(a) WATERS Components and variables



Named Constants
- SYS_CPU_LIM = 1200
- SYS_MEM_LIM = 1000
- P1_LBE = 2
- P1_CPU = 300
- P1_CPU_COST = 375
- P1_MEM = 250
- P1_THRESH = 40
- P1_LIM = 7
- P1_MIN = 1
- P1_MAX = 1
- P2_LBE = 1
- P2_CPU = 300
- P2_CPU_COST = 270
- P2_MEM = 200
- P2_LIM = 7
- P2_THRESH = 40
- P2_MIN = 1
- P2_MAX = 2

Event Aliases

(b) WATERS Constants



- :accepting
- :forbidden
- Pod1Activate
- Pod1Cancel
- Pod1Dep
- Pod1Fail
- Pod1Restart
- Pod1Term
- Pod2Activate
- pod2Cancel
- Pod2Dep
- Pod2Fail
- Pod2Restart
- Pod2Term
- req1Complete
- req1In
- req2Complete
- req2In

(c) WATERS Events

Figure 8.2: WATERS Elements

The Figures 8.2a and 8.2b show the components and constants that are utilised within WATERS to model cluster architecture. In the shown Figures, a system is presented with two plants, Pod1 and Pod2. Each represents a different deployment *tenant* within the cluster. Requests entering each pod are handled by Request plants, while the System supervisor establishes control within our model. We keep track of how our system evolves over time through the defined variables in 8.2a, which track modelled resource utilisation. Within 8.2b, we are presented with the system constants that determine our modelled deployments' possible behaviour. The role of each defined constant is as follows:

- *SYS_CPU_LIM* is the maximum value associated with CPU allocation for the node. It is given in units of millicores.
- *SYS_MEM_LIM* is the maximum value associated with memory allocation for the node, given in units of mebibytes

For each deployment, the constants associated with allocation and performance are provided given a prefix that matches them to the pod component that engages in deployment related actions, shown in 8.2b as P1 and P2. The constants are then given as follows:

- *CPU* is the cost in millicores associated with a single pod of the given type being deployed.
- *CPU_COST* is the measured CPU utilisation when handling a single concurrent request.
- *MEM* is the cost in mebibytes associated with a single pod of the given type being deployed.
- *THRESH* is the threshold for resource utilisation as a percentage of the total resources assigned.
- *LIM* is the maximum queue length that a given pod can handle before it encounters a failure point due to an Out Of Memory (OOM) exception
- *LBE* is the error associated with load balancing, representing the number of potentially incorrectly routed requests per pod replica beyond 1.

- *MIN* is the minimum number of active pods that the deployment can scale down to.
- *MAX* is the maximum number of active pods that the deployment can scale up to.

The first thing of note within the definitions presented in Figure 8.2b are the *MIN* and *MAX* values given for P1 and P2. For P1, both of these values are set as 1. This represents a deployment that seeks to maintain only one active pod with no dynamic scaling. P2 has values of 1 and 3 respectively, indicating that the deployment is can scale between 1 and 3 pod replicas. It is also worth noting that *THRESH* values are given as integers rather than floating point decimals as might be expected for percentage based scaling. This is due to the fact that WATERS only supports integer values.

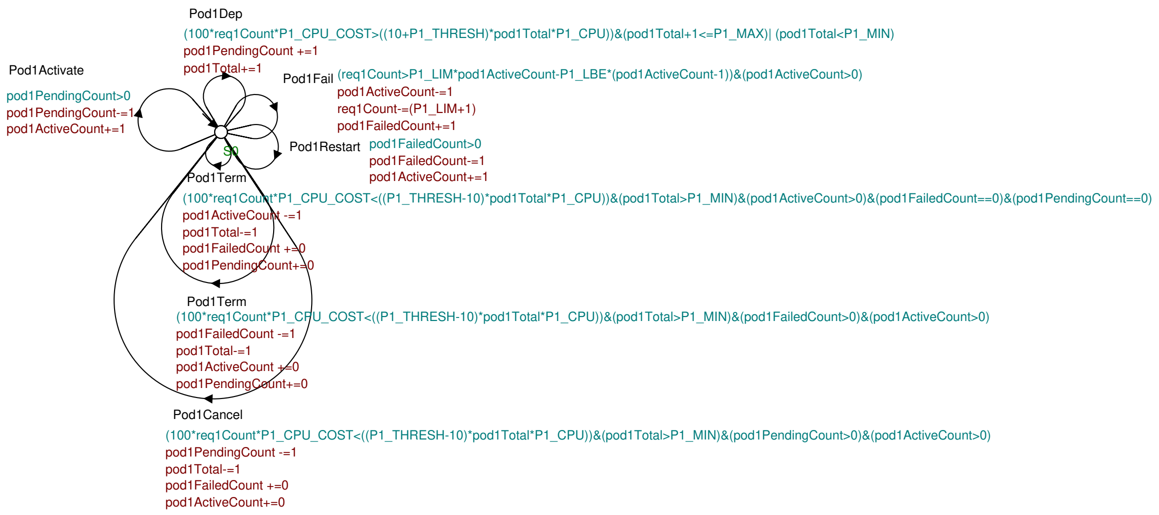
We track the state of our cluster through variable tracking, as indicated within Figure 8.2a. We keep track of variables for all tenants, distinguishing between them numerically.

- *reqCount* is a counter for how many active requests there are being handled by the relevant pods.
- *podActiveCount* is the number of deployed pods that are currently in an active state and able to handle requests.
- *podFailedCount* is the number of deployed pods that are currently in a failed state due to exceeding their capacity.
- *podPendingcount* is the number of pending pods. These are pods that have been created due to some scaling action, but are not yet active.
- *podTotal* is the total number of pods across all three states.

Figure 8.2a also shows the plant and supervisory components of our system. These are our pods, requests, and system. These components capture the behaviour of our system through transitions over defined events. These transitions then impact the variable values. The events can be seen in Figure 8.2c and are again distinguished numerically.

- *PodActivate* is the event that transitions a pod from a pending state into an active state.
- *PodCancel* destroys a pending pod if it is no longer required.
- *PodDep* creates a pending pod.
- *PodFail* transitions a pod from an active state to a failed state.
- *PodRestart* transitions a pod from a failed state to an active state.
- *PodTerm* removes an active pod.
- *reqIn* indicates an incoming request. This is an uncontrollable action and must be possible whenever it is enabled.
- *reqComplete* is the completion of a request.

8.3.1.2 Pods



Pod1Dep
(100*req1Count*P1_CPU_COST>((10+P1_THRESH)*pod1Total*P1_CPU))&(pod1Total+1<=P1_MAX)| (pod1Total<P1_MIN)
pod1PendingCount +=1
pod1Total+=1
Pod1Activate
pod1PendingCount>0
pod1PendingCount-=1
pod1ActiveCount+=1
Pod1Fail
(req1Count>P1_LIM*pod1ActiveCount-P1_LBE*(pod1ActiveCount-1))&(pod1ActiveCount>0)
pod1ActiveCount-=1
req1Count-=(P1_LIM+1)
pod1FailedCount+=1
S0
Pod1Restart
pod1FailedCount>0
pod1FailedCount-=1
pod1ActiveCount+=1
Pod1Term
(100*req1Count*P1_CPU_COST<((P1_THRESH-10)*pod1Total*P1_CPU))&(pod1Total>P1_MIN)&(pod1ActiveCount>0)&(pod1FailedCount==0)&(pod1PendingCount==0)
pod1ActiveCount -=1
pod1Total-=1
pod1FailedCount +=0
pod1PendingCount+=0
Pod1Term
(100*req1Count*P1_CPU_COST<((P1_THRESH-10)*pod1Total*P1_CPU))&(pod1Total>P1_MIN)&(pod1FailedCount>0)&(pod1ActiveCount>0)
pod1FailedCount -=1
pod1Total-=1
pod1ActiveCount +=0
pod1PendingCount+=0
Pod1Cancel
(100*req1Count*P1_CPU_COST<((P1_THRESH-10)*pod1Total*P1_CPU))&(pod1Total>P1_MIN)&(pod1PendingCount>0)&(pod1ActiveCount>0)
pod1PendingCount -=1
pod1Total-=1
pod1FailedCount +=0
pod1ActiveCount+=0

Figure 8.3: WATERS Pod component

In Figure 8.3, we see the possible actions for pods within a given deployment, shown by the black text transition labels within the WATERS model. The ability of any given action to fire within a model is determined by a guard, the logic of which is shown in the blue text within the Figure. As actions are fired, they impact the associated variables following the logic of the red text within the Figure. The variables that are used within these checks can be found in Figures 8.2a and 8.2b. The functionality provided by WATERS for storing variables that may be subject to change as a system evolves or is altered allows us to track the impact of deployment actions on system resources. The defined guards for each transition are as follows:

PodDep models the deployment of a new pod. This action is able to fire if the conditions of one of two logical guards is met:

- The current request queue associated with the deployment must be such that CPU utilisation is above the set threshold required for horizontal pod scaling. Scaling actions must not exceed the maximum number of pod replicas associated with the relevant deployment: $100 * req1Count * P1\_CPU\_COST >= ((10 + P1\_THRESH) * pod1Total * P1\_CPU)$ **AND** $pod1Total + 1 <= P1\_MAX$. Note that we engage scaling at 10% to either side of the threshold value to avoid thrashing behaviour where pod replicas are rapidly created and terminated to meet a set-point.

- The current number of pods for the relevant deployment is below the minimum acceptable number due to a pod failure: $pod1Count < P1\_MIN$.

PodTerm is the modelled action that represents the termination of a Pod within a deployment. There are two possible outcomes from this transition, each action is able to fire if both of the following guard conditions are met:

- The current request queue associated with the deployment is such that the CPU utilisation across pods in the deployment is less than the given threshold value.: $100*req1Count*P1\_CPU\_COST <= ((P1\_THRESH - 10) * pod1Total * P1\_CPU)$.

- Scaling the deployment down must not result in a pod count below the minimum accepted number of replicas: $pod1Total - 1 >= P1\_MIN$.

The choice of transition is determined by the state of pod replicas associated with the given deployment. If there are failed pods within the deployment, then they will be prioritised for termination. If there are no pods in a failed state, then an active pod is terminated.

If there are pending pods present and the requirements for pod termination are met, then it is possible to cancel a pending pod instead. PodDep, PodCancel, and PodTerm actions result in a change (either an increase or decrease) to the relevant podTotal value. Enabling and disabling of these actions is achieved via a comparison between the CPU utilisation associated with the current request count and the threshold set-point associated with scaling actions. The scale factor of 100 is to allow for the use of integers within this formula, as previously discussed.

PodFail is the action modelling pod failures. This action is enabled when the number of requests in the queue are greater than the maximum total memory capacity of the active pods for the relevant deployment, after accounting for load balancing errors: $req1Count > P1\_LIM * pod1ActiveCount - P1LBE * (pod1ActiveCount - 1)$. This is to simulate the handling of *Out of Memory* (OOM) errors where pods are forcibly killed. When a fail action is undertaken, the podActiveCount is decreased and the podFailedCount increased. The number of active requests in the queue is decreased by the pod request limit plus the extra request: $req1Count -= P1\_LIM + 1$. This simulates dropping of requests that were being handled by the pod in question.

PodRestart is the action taken to restart failed pods. This action is enabled when there are pods within the deployment that are in a failed state. The outcome of this action is to decrease the podFailCount and increase the podActiveCount.

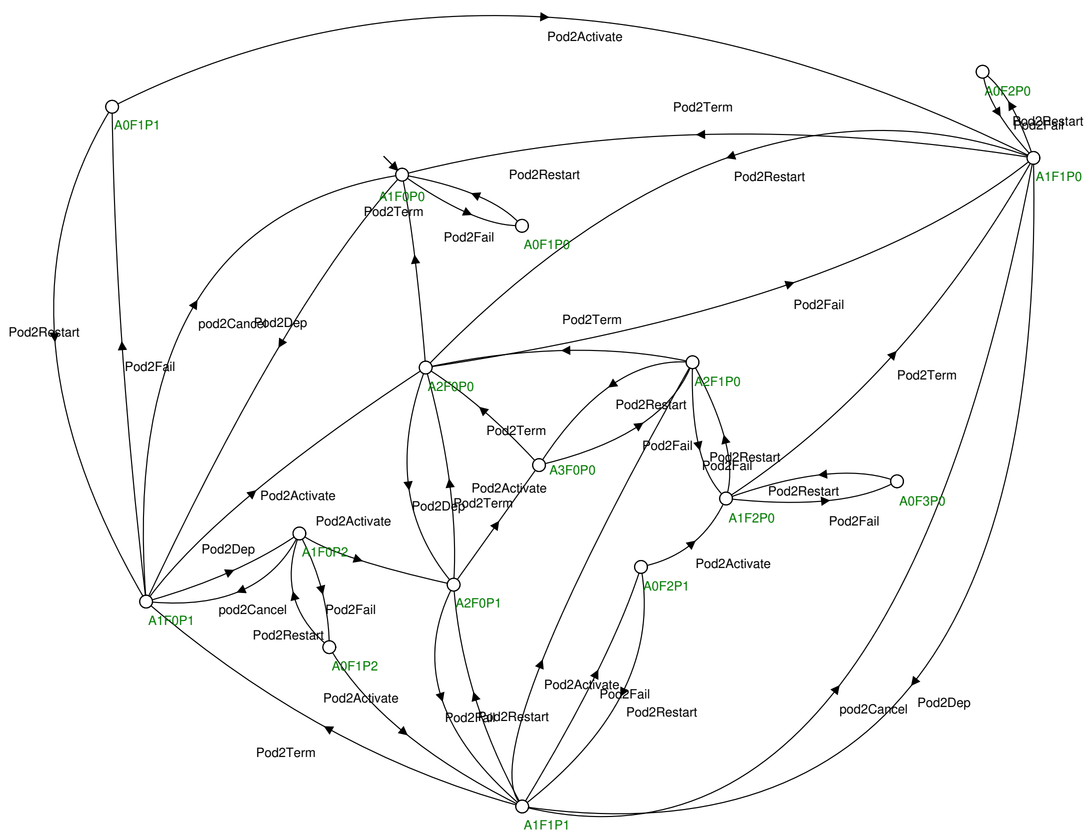


Figure 8.4: WATERS Unpacked Deployment

The logic dictated within the guards of each transition within the automaton controls the possible cluster statespace through limiting the behaviours of pods within the deployment to those allowed by the defined guards. This means that we can capture all of the states that we encounter within a deployment within a single state automaton with supporting variables. Without the use of variable tracking, this can be unpacked to give the deployment statespace shown within 8.4. Each state in the automaton is named with regard to the status of deployed pods, in the format *AiFjPk* where *i* is the number of active pods, *j* is the number of failed pods, and *k* is the number of pending pods within the deployment. The number of states is determined by the maximum number of running pod replicas defined within the constants of Figure 8.2b. Our approach to modelling deployment behaviour using guard logic with an emphasis on defined transitory behaviour rather than representative state-space, we are able to remove the need for creation of new automata when we wish to explore different cluster configurations, as altering constant values will automatically adjust our model to fit changing deployment and cluster resource configurations. We are thus able to capture the core behaviours of a generalised cluster, which we can adjust to suit our context. We can show that our refined model captures the same behaviour as the unpacked automaton, by producing a synchronous product of each option and comparing the number of states in the resulting space. Equivalent models will produce the same number of states.

8.3.1.3 Requests

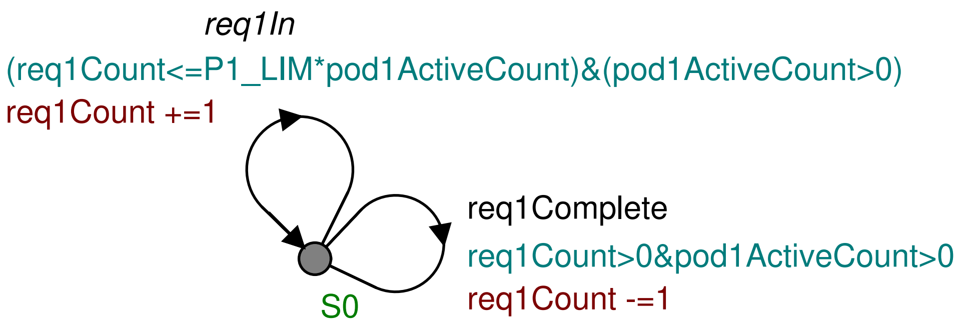

req1In
(req1Count<=P1_LIM*pod1ActiveCount)&(pod1ActiveCount>0)
req1Count +=1
req1Complete
req1Count>0&pod1ActiveCount>0
req1Count -=1
S0

Figure 8.5: WATERS Request Component

Figure 8.5 represents requests entering or being completed within the system.

reqIn is an uncontrollable action, meaning that no other specification components within the system are able to stop it from occurring. The guards $req1Count <= P1\_LIM * pod1Count$ and $pod1Count > 0$ stop the infinite increase of requests in a queue that cannot be handled, as our modelling approach is not concerned with modelling a backlog of requests, as going beyond the limit for pod request handling results in an OOM failure, which is already modelled within our Pod component.

reqComplete is enabled when there are requests present in the queue, and an active pod that is able to handle them. This action decreases reqCount.

8.3.1.4 System

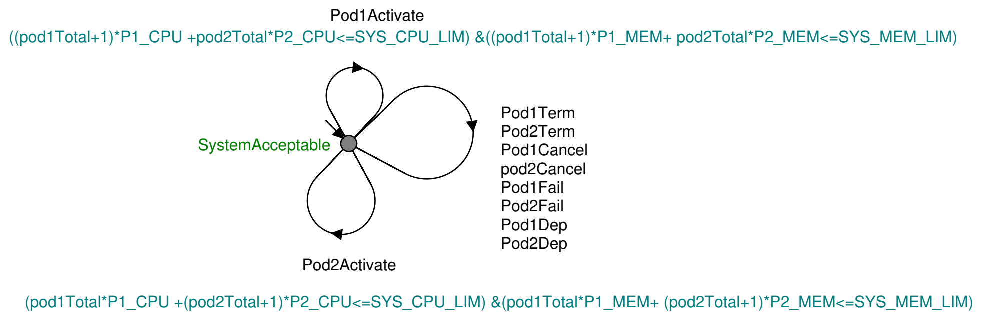

Pod1Activate
((pod1Total+1)*P1_CPU +pod2Total*P2_CPU<=SYS_CPU_LIM) &((pod1Total+1)*P1_MEM+ pod2Total*P2_MEM<=SYS_MEM_LIM)
SystemAcceptable
Pod1Term
Pod2Term
Pod1Cancel
pod2Cancel
Pod1Fail
Pod2Fail
Pod1Dep
Pod2Dep
Pod2Activate
(pod1Total*P1_CPU +(pod2Total+1)*P2_CPU<=SYS_CPU_LIM) &(pod1Total*P1_MEM+ (pod2Total+1)*P2_MEM<=SYS_MEM_LIM)

Figure 8.6: WATERS System Controller

Figure 8.6 then represents the System component, which is the controller for our cluster model. This component is not of much consequence for systems that utilise only a single deployment, but becomes increasingly relevant as we seek to model interactions associated with multitenancy. The component synchronises with all deployment, termination, and failure actions across all pod components within the cluster. To this end, the only guards are associated with deploying new pods into a running system, as this is the only action that is impacted by node resource constraints. A deployment action is only allowed to fire if the additional resource costs incurred by creating another pod instance do not cause the total system CPU and Memory allocation to exceed the max limit as defined within our defined Constants.

8.3.2 Verification

We model components of a Kubernetes Cluster as automata within WATERS. We engage in sanity checking for our model properties. We ensure that our non-deterministic representation of our cluster architecture reflects expected behavioural properties. To this end, we use WATERS inbuilt framework to verify that our model is controllable by our system specification. We also verify that the synchronous composition of components within our model does not ever result in deadlocked states and that uncontrollable actions such as incoming requests are never blocked. We observe the impact of scaling components on the length of request queues and the scaling ability of other tenants within the cluster. Our WATERS model captures functionality of components within a Kubernetes cluster. This establishes possible behaviours and configurations of our modelled system, but is not indicative of expected performance. We apply the logic of our guards within WATERS to the generation stage of our PEPA model. We thus create a digital twin that captures the same logic, while allowing for further evaluation of performance. We establish consistency between models through comparison between the generated synchronous statespaces.

8.4 PEPA

While modelling in WATERS provides us with a mechanism through which we are able to perform qualitative analysis of our cluster, it does not allow for the evaluation of expected performance levels. Thus a second modelling approach using PEPA is also utilised to explore the potential for performance evaluation once we are confident in the behaviours of our WATERS model.

8.4.1 Automatic Model Generation

While WATERS offers an intuitive GUI, PEPA models are created using a process algebraic syntax. As such, it is difficult to conceptualise the behaviour of components within the context of transitions inside of a statespace. Our approach is therefore, to create an automatic model generator that is capable of language within a PEPA syntax to reflect the logic of our WATERS model, based on a system information textfile. This approach provides a framework for quickly producing and evaluating performance models based on known information about a cluster. This framework fits within the third wave of self-adaptive systems, providing a mechanism for producing models at runtime.

8.4.1.1 System Metrics

Model generation is based on information from a textfile that describes system metrics. The following sections discuss the behaviour of a model identical to that of the given WATERS example, generated based on the following parameters:

```
sysmem: 1000
syscpu: 1200
Label: a
memReq: 250
cpuReq: 300
cpuLim: 300
cpuCost: 375
reqLim: 8
reqIn: 1
minReps: 1
maxReps: 2
thresh: 0.4

Label: b
memReq: 200
cpuReq: 300
cpuLim: 300
cpuCost: 270
reqLim: 7
reqIn: 1
minReps: 1
maxReps: 1
thresh: 0.4
```

The given parameters describes a cluster containing two deployments, *a* and b. The cluster node has a memory capacity of 1000mb and 1200millicores of CPU that are allocatable. Upon creation, each pod is allocated their respective cpu and memory. Each pod also has an associated CPU limit of 300. The cpuCost is how much cpu is utilised at each level of concurrency, while reqLim shows the maximum concurrent requests before the pod is killed with an Out of Memory exception. reqIn is the initial rate of incoming requests given as requests per second. We vary the incoming request rate to evaluate performance. The Reps values give us the range of actively deployed pod replicas that the deployment may scale between. Thresh is the average CPU utilization value for which scaling is enabled.

8.4.1.2 Rates

```
deploy = 0.06;

restart = 0.01;

terminate = 5;

activate = 0.3;

a_RI = 1.0;

b_RI = 1.0;
```

The maximum throughput capacity of our modelled system is established by finding the number of CPU periods required to service each level of request concurrency.

This can be written as:

$$CPU\_Periods = \frac{ReqCost_{CPU}}{CPU\_Limit}$$

Where $ReqCost_{CPU}$ is the measured CPU utilisation across a single concurrent request, and CPU_Limit is the limit assigned within our deployment configuration file.

$$ServiceRate = \frac{10}{CPU\_Periods}$$

Default Kubernetes behaviour defines a single CPU period as 100ms, or 0.1 seconds. As such, $\frac{10}{CPU\_Periods}$ gives us the expected service rate of a request. Within PEPA, transitions are made with a probability that is determined by the rate of the current activity as a ratio to all other enabled activity rates at that point in time. Rates within our physical cluster are consistently measured in units/second. We transpose these rates directly into our generated model. The incoming request rate can be set by varying the value associated with aRI. The heavy weighting given to termination ensures that this action will be favoured when available, which is also reflective of a practical environment. Rates associated with *deploy*, *restart*, and *activate* actions are taken from known behaviour of Kubernetes. Kubernetes HPA components scale pod replica counts when the average utilisation is above the set-point over a 15 second period. Restarts occur 10 seconds after pod failure is determined. We establish our activation rate based on the observed time taken for scheduled pods to be capable of handling requests.

8.4.1.3 Node Resource Constraints

The reader may recall from previous sections that PEPA does not encapsulate components in the same manner as WATERS. Instead, the PEPA syntax makes use of constants with defined behaviour. In the below example, we see queues that represent node memory (*SM*) and CPU (*SC*) capacities. Each queue represents the possible states of the relevant node resource pool. Each state in the queue component is given as a named constant. This constant has an associated number of choices between action-rate pairs, after which it will behave as the next constant. In this instance, **SM1** is the initial position in our memory queue, from which there is a choice of transitions between that of action *a_act* and *b_act*, after which the component will act as the constant **SM13** or **SM14** respectively.

```
SM9 = (a_act, activate).SM14 + (b_act, activate).SM13;

SM10 = (a_act, activate).SM15 + (b_act, activate).SM14;
```

```
SM11 = (a_act, activate).SM16 + (b_act, activate).SM15;
SM12 = (a_act, activate).SM17 + (b_act, activate).SM16;
SM13 = (a_act, activate).SM18 + (b_act, activate).SM17 +
    (b_term, terminate).SM9;
SM14 = (a_act, activate).SM19 + (a_term, terminate).SM9 +
    (b_act, activate).SM18 + (b_term, terminate).SM10;
    .
    .
    .
SM20 = (a_term, terminate).SM15 + (b_term, terminate).SM16;


SC2 = (a_act, activate).SC3 + (b_act, activate).SC3;
SC3 = (a_act, activate).SC4 + (a_term, terminate).SC2 +
    (b_act, activate).SC4 + (b_term, terminate).SC2;
SC4 = (a_term, terminate).SC3 + (b_term, terminate).SC3;
```

Pods within a cluster are constrained by limited physical resources that are available across physical nodes. As such, we must consider maximum node capacity within our model. While WATERS allows for the tracking of variables, this is not a feature that is supported within PEPA. As such, we model our capacity using queues to represent the total capacity of the node with regard to CPU and memory. Queue capacity is determined by the highest common factor between the resource costs of each modelled pod and the total capacity of the system. Each state within the queue represents a movement with this granularity. States within the queue are linked using activate and terminate actions across modelled deployments. As deployments scale, the resource queue position will change based on the freed/reserved resources of the relevant deployment.

8.4.1.4 Deployment Queues

```
Pa_A_1_F_0_P_0_0 = (req, a_RI).Pa_A_1_F_0_P_0_1;
```

```
Pa_A_1_F_0_P_0_1 = (req, a_RI).Pa_A_1_F_0_P_0_2 +
    (serve, 8.0).Pa_A_1_F_0_P_0_0 +
    (a_dep, deploy).Pa_A_1_F_0_P_1_1;
    .
    .
    .
Pa_A_1_F_0_P_0_8 = (serve, 8.0).Pa_A_1_F_0_P_0_7 +
    (a_fail, a_RI).Pa_A_0_F_1_P_0_0 +
    (a_dep, deploy).Pa_A_1_F_0_P_1_8;
```

Deployments are represented as parallel queues that represent the total capacity for concurrent request handling. The above queue represents our described cluster at minimum expected operating capacity. The prefix of each state in a queue identifies it as a Pod (**P**) from deployment **a**, taken from the system description file. We identify the current queue by the number of active, failed, and pending pod replicas, **A1F0P0**, while the integer following the underscore represents the current number of concurrent requests within the system. **a_RI** is the rate at which requests associated with the deployment, *a*, are coming into the system, while the service rate is determined by a formula to reflect CPU throttling behaviours within the architecture. Deployment and Termination activities are added to the model by the generator based on the scaling threshold given in the descriptor file. Likewise, the ability for a pod to fail is determined at the generation stage of the model based on expected load balancing of requests across pod replicas within the cluster. As deployments scale, they enter into a parallel queue at the same position, but with capacity reflecting the new number of active pods.

```
Pa_A_1_F_0_P_1_0 = (req, a_RI).Pa_A_1_F_0_P_1_1 +
    (a_cancel, terminate).Pa_A_1_F_0_P_0_0 +
    (a_act, activate).Pa_A_2_F_0_P_0_0;
Pa_A_1_F_0_P_1_1 = (req, a_RI).Pa_A_1_F_0_P_1_2 +
    (serve, 8.0).Pa_A_1_F_0_P_1_0 + (a_act, activate).Pa_A_2_F_0_P_0_1;
    .

    .

    .

Pa_A_1_F_0_P_1_7 = (req, a_RI).Pa_A_1_F_0_P_1_8 +
    (serve, 8.0).Pa_A_1_F_0_P_1_6 + (a_act, activate).Pa_A_2_F_0_P_0_7;
Pa_A_1_F_0_P_1_8 = (serve, 8.0).Pa_A_1_F_0_P_1_7 +
    (a_fail, a_RI).Pa_A_0_F_1_P_1_0;
```

This is achieved firstly through a deploy action that transitions to a parallel queue with an added pending pod, followed by an activate action that transitions into a queue with an added active pod.

```
Pa_A_2_F_0_P_0_0 = (req, a_RI).Pa_A_2_F_0_P_0_1 +
    (a_term, terminate).Pa_A_1_F_0_P_0_0;
Pa_A_2_F_0_P_0_1 = (req, a_RI).Pa_A_2_F_0_P_0_2 +
    (serve, 16.0).Pa_A_2_F_0_P_0_0;
Pa_A_2_F_0_P_0_2 = (req, a_RI).Pa_A_2_F_0_P_0_3 +
    (serve, 16.0).Pa_A_2_F_0_P_0_1;
    .

    .

    .

Pa_A_2_F_0_P_0_13 = (req, a_RI).Pa_A_2_F_0_P_0_14 +
    (serve, 16.0).Pa_A_2_F_0_P_0_12 + (a_fail, a_RI).Pa_A_1_F_1_P_0_6;
Pa_A_2_F_0_P_0_14 = (req, a_RI).Pa_A_2_F_0_P_0_15 +
    (serve, 16.0).Pa_A_2_F_0_P_0_13 + (a_fail, a_RI).Pa_A_1_F_1_P_0_7;
Pa_A_2_F_0_P_0_15 = (serve, 16.0).Pa_A_2_F_0_P_0_14 +
    (a_fail, a_RI).Pa_A_1_F_1_P_0_7;
```

Pod failure thresholds are determined by our load balancing error algorithm outlined later in this section. This returns a threshold value that reflects the point at which a pod within the deployment queue may exceed its memory limit and be killed, allowing for a margin of error due to incorrect routing of requests by the load balancer. As this failure is triggered by a request count above the capactiy of our pod, it has the same rate when enabled as incoming requests. Failure actions transition into another parallel queue. The index is reduced by the maximum capacity of the a single pod to reflect dropped requests, while the state label changes to indicate the updated pod status.

```
Pa_A_0_F_1_P_0_0 = (req, a_RI).Pa_A_0_F_1_P_0_0 +
    (a_restart, restart).Pa_A_1_F_0_P_0_0;


Pa_A_0_F_1_P_1_0 = (req, a_RI).Pa_A_0_F_1_P_1_0 +
    (a_restart, restart).Pa_A_1_F_0_P_1_0 +
    (a_act, activate).Pa_A_1_F_1_P_0_0;


Pa_A_0_F_2_P_0_0 = (req, a_RI).Pa_A_0_F_2_P_0_0 +
    (a_restart, restart).Pa_A_1_F_1_P_0_0;


Pa_A_1_F_1_P_0_0 = (req, a_RI).Pa_A_1_F_1_P_0_1 +
    (a_term, terminate).Pa_A_1_F_0_P_0_0;
    .
    .
    .
Pa_A_1_F_1_P_0_8 = (serve, 8.0).Pa_A_1_F_1_P_0_7 +
    (a_fail, a_RI).Pa_A_0_F_2_P_0_0;
```

8.4.1.5 System Equation

```
    ((SM9<a_act, a_term, b_act, b_term>SC2)
    <a_act, a_term, b_act, b_term>
    (Pa_A_1_F_0_P_0_0<>Pb_A_1_F_0_P_0_0))
```

This system equation reflects the same behaviour of our controlling component within the WATERS model. The system itself is comprised of two resource queues, SM and SC, that reflect the CPU and Memory capacity for the modelled node. (SM1<a_act, a_term, b_act, b_term>SC1) shows that both resource queues synchronise with each other on scaling and deployment actions, while the synchronous composition of the two resource queues synchronises on such actions of a deployment. In this way, our PEPA model is able to reflect the capacity of the overall system without having to make use of dynamic variables.

8.4.2 Assumptions

It is not feasible to model the behaviour of a load balancer with regard to directing requests without increasing the complexity of created models and thus increasing the likelihood of state space explosions. As such, it is difficult to establish scaling based on load across a deployment where pods are modelled as individual components, as invariably one pod will reach a failure state while others are under-utilised. The approach of this work is therefore to abstract away the individual pods within a system and instead make use of parallel queues of differing length with regard to request handling, which can be switched between through the use pod scaling actions. The effect of a load balancer is captured by an error margin within our generator that represents the percentage of requests that may be incorrectly routed throughout system. We thus calculate the threshold at which failure actions are enabled as follows,

$$n_{reqs} = \frac{n_{max}}{n_{pods}^{-1} + e} \tag{8.4}$$

where e is the percentage of mis-routed requests, given as a decimal. The number of concurrent requests that must be in the system for Pod failure to be enabled (n_{reqs}) can be found by dividing the maximum number of requests for a single Pod (n_{max}) by the sum of the inverse of the number of Pod replicas (n_{pods}^{-1}) and the load-balancer margin of error. This formula is naive in that it assumes that an increase in active pod replicas will have a proportional effect on total system capacity.

Our combined modelling approach captures the advantages offered by WATERS in verification of model properties and visual representation. Generation of our PEPA model is automated through a generator script that captures logic of our verified model. Steady-state analysis of this PEPA model reveals the probability of a total deployment queue across all pods being at a certain length at any given time. From this, we are able to evaluate the likelihood of our deployment being in any given state with regard to scaling and failure under a given load. This provides us with a framework through which we predict performance of a defined deployment under various load conditions. We evaluate the performance of our generated performance models against observed behaviours of deployed microservices.

Chapter 9

Model Evaluation

We used the information gathered during the classification process as the basis for our models. Initially, we modelled interactive cluster components as EFSMs within WATERS. Transition behaviours were defined through logical guards and actions. Synchronisation between components was captured by WATERS' supervisory control logic. Within WATERS, models were representative of possible system state-space without knowledge of rates or probabilities. WATERS verification was performed to ensure that models adhered to expected behaviours. Once we were satisfied with how our WATERS modelling approach captured cluster behaviour, we developed a tool to automatically generate PEPA models that followed the same logic. Modelled performance was evaluated against observed behaviour for the following deployment scenarios:

Scenario	Pod Memory Limit	Pod CPU Limit
A	150MiB	300millicores
B	250MiB	300millicores
C	350MiB	300millicores
D	250MiB	600millicores

These scenarios were identical to those used within the previously discussed classification process, other than the defined CPU limit. A new Scenario, D, was introduced to evaluate the impact of CPU allocation.

9.1 Analysis of Deterministic Model

In developing our EFSM-based modelling approach, we sought to capture two primary aspects of behaviour. The first was the failure points of pod replicas within a given deployment. The second was possible configurations of pod replicas across deployments where resource competition was encountered. The produced EFSMs within our deterministic model were adapted to fit different system configurations by altering the values of our defined constants.

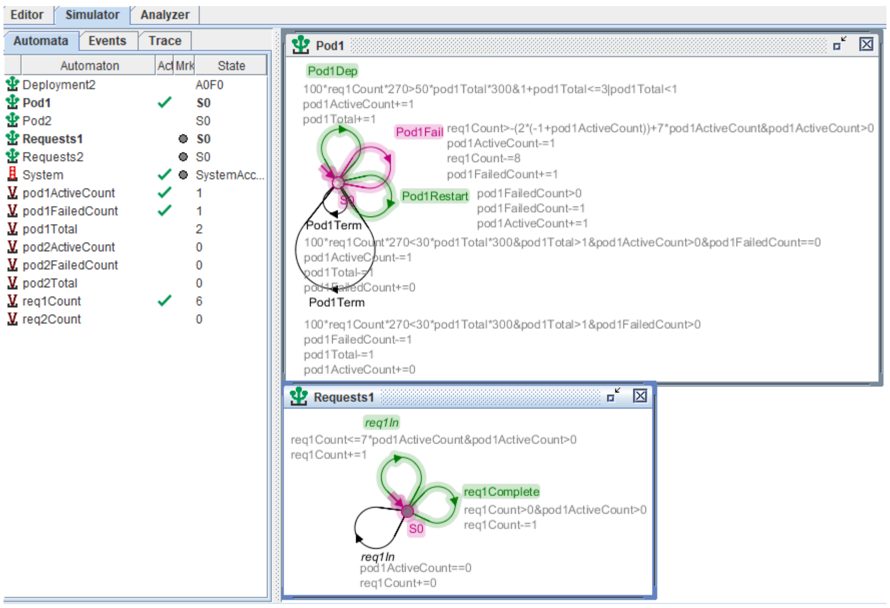
9.1.1 Model Development

Our deterministic modelling approach used variable abstraction to manage transitional logic without explicit mapping of the entire state space. Plant and supervisor components were comprised of a single state, while transition guards provided the controlling logic. Analysis of our model was thus based on the changing of variables as transitions were fired. Simulation was used within the WATERS model checker to evaluate the fit of modelled behaviours. The WATERS verification framework was used to verify that the synchronous product of components adhered to expected model properties

Our EFSMs were adapted to suit our deployment scenarios through the alteration of defined constants. Values were determined during the previously outlined classification process.

9.1.2 Evaluation

The adherence of models to the behaviour of our observed system was evaluated through the WATERS simulation framework. The effects of incoming requests on our modelled systems were evaluated by firing discrete transitions. The WATERS simulator framework enabled and disabled potential transitions within modelled components based on adapting variables. Simulation was performed across the different pod replica configurations of our deployment scenarios. Figure 9.1 illustrates the simulation process.



	Automaton	Act	Mrk	State
	Deployment2			A0F0
	Pod1	✓		**S0**
	Pod2			S0
	Requests1		●	**S0**
	Requests2		●	S0
	System	✓	●	SystemAcc...
	pod1ActiveCount	✓		1
	pod1FailedCount	✓		1
	pod1Total			2
	pod2ActiveCount			0
	pod2FailedCount			0
	pod2Total			0
	req1Count	✓		6
	req2Count			0

```
Pod1Dep
100*req1Count*270>50*pod1Total*300&1+pod1Total<=3|pod1Total<1
pod1ActiveCount+=1
pod1Total+=1

Pod1Fail req1Count>-(2*(-1+pod1ActiveCount))+7*pod1ActiveCount&pod1ActiveCount>0
         pod1ActiveCount-=1
         req1Count-=8
         pod1FailedCount+=1

Pod1Restart pod1FailedCount>0
            pod1FailedCount-=1
            pod1ActiveCount+=1

Pod1Term
100*req1Count*270<30*pod1Total*300&pod1Total>1&pod1ActiveCount>0&pod1FailedCount==0
pod1ActiveCount-=1
pod1Total-=1
pod1FailedCount+=0

Pod1Term
100*req1Count*270<30*pod1Total*300&pod1Total>1&pod1FailedCount>0
pod1FailedCount-=1
pod1Total-=1
pod1ActiveCount+=0
```

```
req1In
req1Count<=7*pod1ActiveCount&pod1ActiveCount>0
req1Count+=1

req1Complete
req1Count>0&pod1ActiveCount>0
req1Count-=1

req1In
pod1ActiveCount==0
req1Count+=0
```

Figure 9.1: WATERS Simulator, Deployment Scenario B

We used the verification framework within WATERS to ensure that our models adhered to expected properties with regard to controlling and blocking behaviours. We generated synchronous products to show the combined statespace of modelled components. This product captures all possible combinations of variables based on the controlling logic of our model. In practice, this presents us with more detail than we require. Such a state-space is immense and cannot be easily represented visually. We compared the size of the statespace produced by our EFSM models with those of our later generated PEPA models. This was done across all pod replica configurations of our deployment scenarios. As both modelling approaches sought to capture the same logic, both should produce the same sized synchronous product under equivalent configurations. We observed consistently equivalent statespaces across our modelling approaches. The following table depicts the comparative statespaces for produced WATERS and PEPA models across different deployment and request configurations.

Deployment Scenario	Request	Pod Replicas	EFSM	PEPA

A	3mb, 100ms	1	6	6
A	3mb, 100ms	2	15	15
A	3mb, 100ms	3	28	28
B	3mb, 100ms	1	10	10
B	3mb, 100ms	2	27	27
B	3mb, 100ms	3	52	52
B	3mb, 100ms	1 Scale 2	47	47
B	3mb, 100ms	1 Scale 3	136	136
B	3mb, 100ms	2 Scale 3	106	106
B	3mb, 50ms	1	10	10
B	3mb, 50ms	2	27	27
B	3mb, 50ms	3	52	52
B	3mb, 200ms	1	10	10
B	3mb, 200ms	2	27	27
B	3mb, 200ms	3	52	52
B	1mb, 100ms	1	23	23
B	1mb, 100ms	2	66	66
B	1mb, 100ms	3	130	130
B	5mb, 100ms	1	7	7
B	5mb, 100ms	2	18	18
B	5mb, 100ms	3	34	34
C	3mb, 100ms	1	7	7
C	3mb, 100ms	2	18	18
C	3mb, 100ms	3	34	34
D	3mb, 100ms	1	10	10
D	3mb, 100ms	2	27	27
D	3mb, 100ms	3	52	52

Table 9.2: Comparative Statespaces across Modelling Approaches

9.2 Probabilistic Modelling

We created a tool to automatically produce probabilistic models to represent a Kubernetes cluster. We used this tool to produce PEPA models that captured the same behaviour as logical guards and actions within our EFSA models. Transitions relating to cluster management were assigned rates that reflected observed behaviours across our Kubernetes environments. Pod replica capacities and system resource allocations were modelled from information gathered during the previously discussed classification process. Rates assigned to transitions relating to microservice performance were calculated from classification information.

9.2.1 Model Evaluation

We used the PEPA workbench to derive the statespace for our modelled systems. Steadystate analysis was performed to associate probabilities with states in the resulting synchronous products. We analysed these statespaces to find the different possible combinations of pod replicas within the model, each being in an *active*, *pending*, or *failed* state. This analysis process was automated to find the probability of each pod state combination. We performed this evaluation process over a range of modelled incoming request rates. The rate of request servicing was calculated at generation time based on the observed CPU coefficient found within the previously discussed classification process. The *Pod Req Limit* was the observed concurrency limit, also established during classification. This information for our experimental deployment scenarios is given in Table 9.3.

Scenario	Request	CPU Coefficient	Pod Req Limit	Service Rate
A	3mb 100ms	366	4	8.2
B	3mb 100ms	352	8	8.52

B	3mb 50ms	441	8	6.80
B	3mb 200ms	219	8	13.7
B	1mb 100ms	186	21	16.1
B	5mb 100ms	410	5	7.32
C	3mb 100ms	310	15	9.68
D	3mb 100ms	366	8	16.4

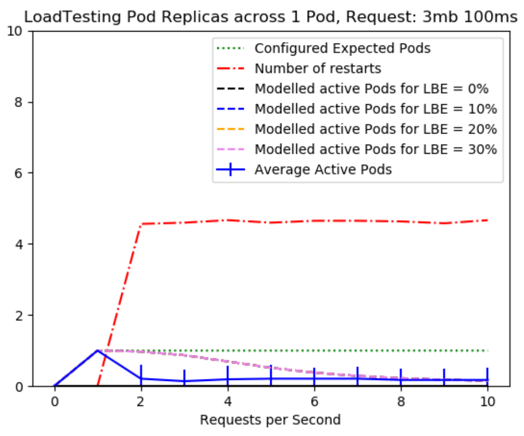
Table 9.3: Performance Information For Deployment Scenarios

Within the following sections, we present figures that plot the number of active and expected pod replicas for our modelled and deployed applications. We present results across different deployment scenarios and request configurations. Figures are presented alongside brief evaluation of model suitability. We examine experimental results to gain insights into observed performance. We suggest improvements to our initial modelling approach. Finally, we contextualise development of our automated modelling strategy within our proposed self-adaptive architecture.

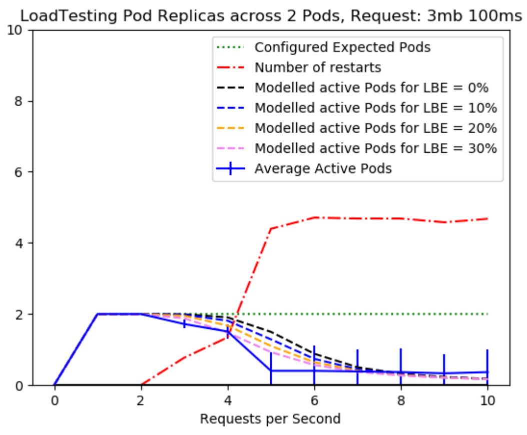
We evaluated the performance of our modelled system against the results attained from our load testing experiments. We repeated this process across different request patterns and deployment scenarios. The configured expected and the average active pods refer to pods deployed within our physical cluster, while the modelled active pods refer to the predicted number of active pods produced by our models under different degrees of load balancing error (LBE). For our deployed application, each plotted point was found by averaging the observed number of active pods across 3 repeated experimental rounds. Our modelled results were produced evaluating the performance of our modelled system under the same load conditions associated with our observed experimental results. The number of restarts refers to the average observed number of times that deployed individual pod replicas crashed and were restarted during each set of experimental rounds.

9.2.1.1 Deployment Scenarios

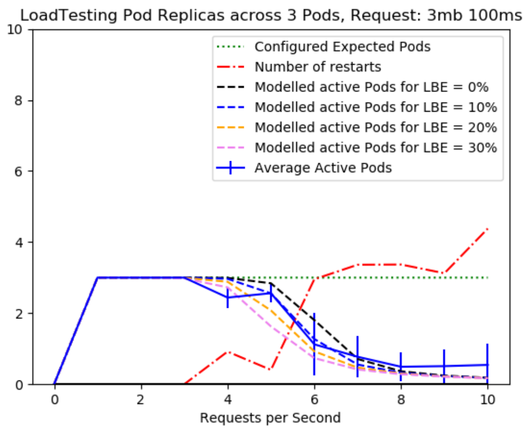
We evaluated the performance of our probabilistic models against the results attained through our load testing experiments across our 4 deployment scenarios.



(a) 1 Pod Replica, Scenario A: Each Pod Assigned 150MiB and 300millicores

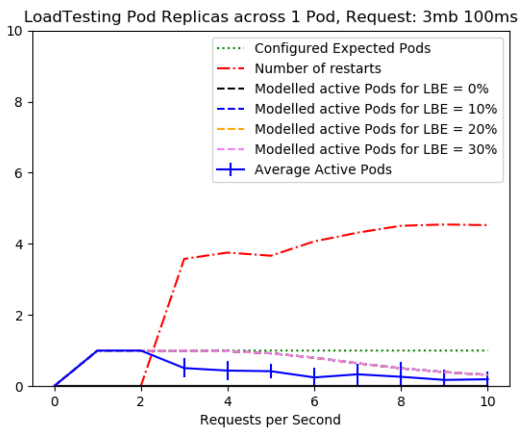


(b) 2 Pod Replicas, Scenario A: Each Pod Assigned 150MiB and 300millicores

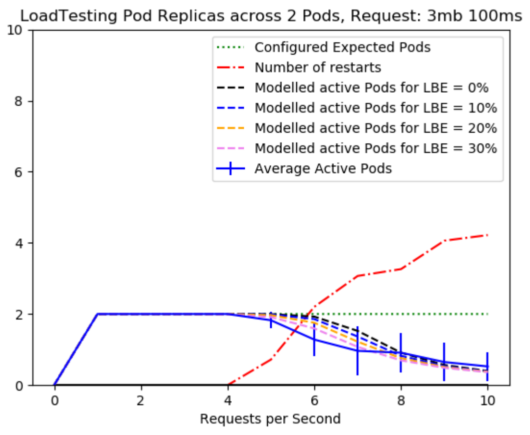


(c) 3 Pod Replicas, Scenario A: Each Pod Assigned 150MiB and 300millicores

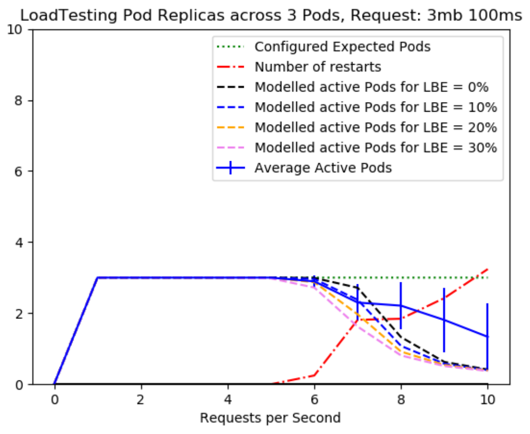
Figure 9.2: Modelled and Observed Replica Status For Scenario A, Requests of 3mb and 100ms



(a) 1 Pod Replica, Scenario B: Each Pod Assigned 250MiB and 300millicores

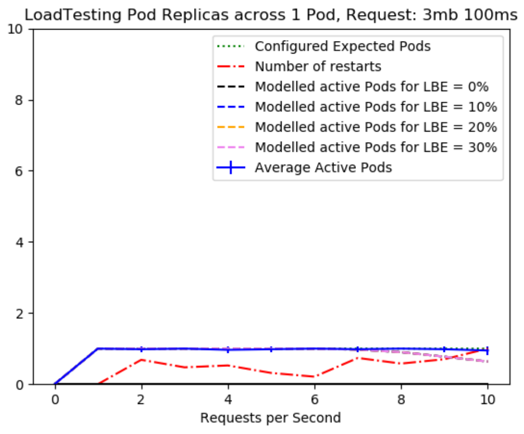


(b) 2 Pod Replicas, Scenario B: Each Pod Assigned 250MiB and 300millicores

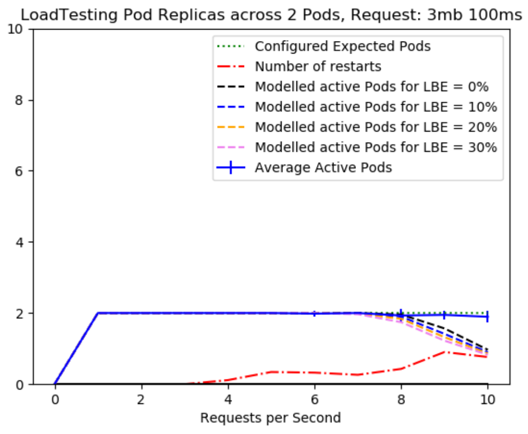


(c) 3 Pod Replicas, Scenario B: Each Pod Assigned 250MiB and 300millicores

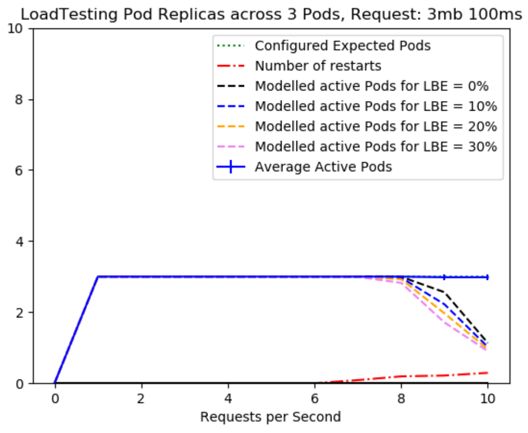
Figure 9.3: Modelled and Observed Replica Status For Scenario B, Requests of 3mb and 100ms



(a) 1 Pod Replica, Scenario C: Each Pod Assigned 350MiB and 300millicores

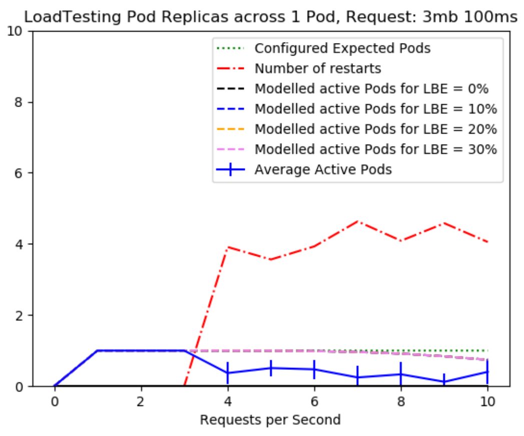


(b) 2 Pod Replicas, Scenario C: Each Pod Assigned 350MiB and 300millicores

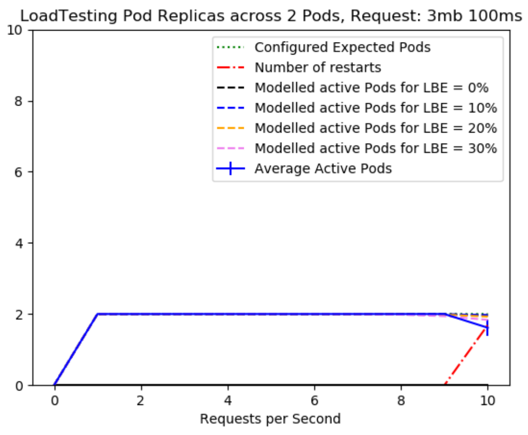


(c) 3 Pod Replicas, Scenario C: Each Pod Assigned 350MiB and 300millicores

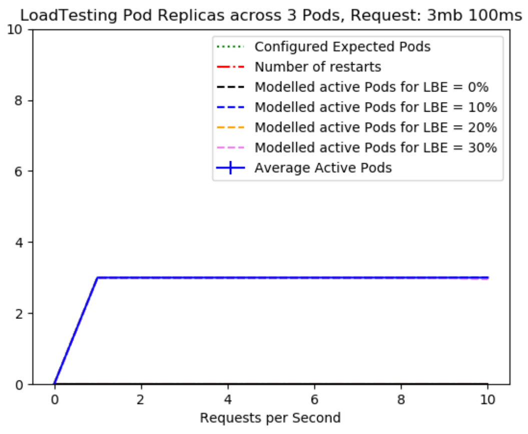
Figure 9.4: Modelled and Observed Replica Status For Scenario C, Requests of 3mb and 100ms



(a) 1 Pod Replica, Scenario D: Each Pod Assigned 250MiB and 600millicores



(b) 2 Pod Replicas, Scenario D: Each Pod Assigned 250MiB and 600millicores



(c) 3 Pod Replicas, Scenario D: Each Pod Assigned 250MiB and 600millicores

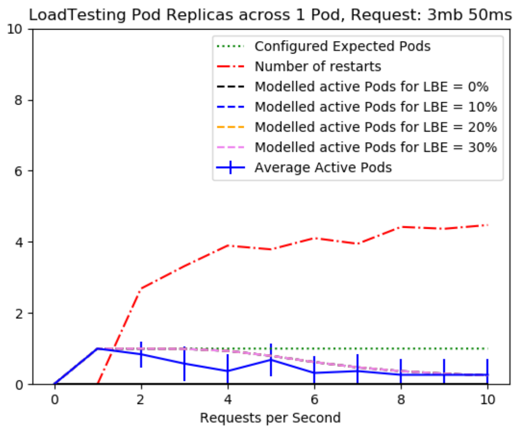
Figure 9.5: Modelled and Observed Replica Status For Scenario D, Requests of 3mb and 100ms

Figures 9.2, 9.3 9.4, and 9.5 present comparative performance of our probabilistic models against observed performance of the relevant deployment configurations. We see that the performance of our modelled systems was reasonably consistent with observed performance of the associated scenario. We identify two trends from the presented graphs.

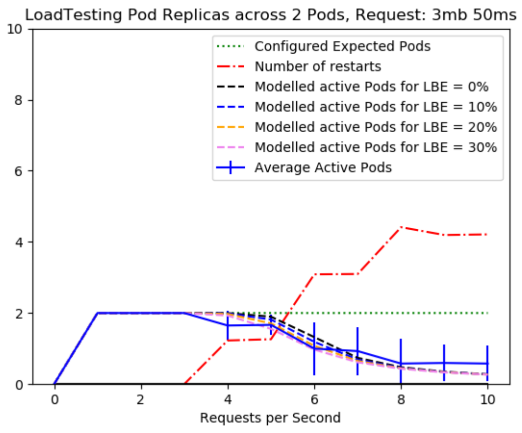
1. Modelled performance exceeded observed performance for configurations consisting of a single pod replica, with the exception of Scenario C.
2. Modelled systems underperformed as resource allocations were increased

9.2.1.2 Request configuration

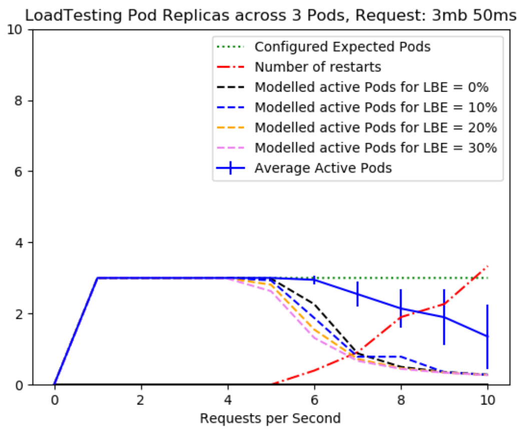
We evaluated the performance of our probabilistic models against the results attained through load testing experiments across multiple request configurations. The size and duration of requests were varied as respective independent variables. All of these experiments were conducted on deployment Scenario B as a controlled environment.



(a) 1 Pod Replica, Duration 50ms

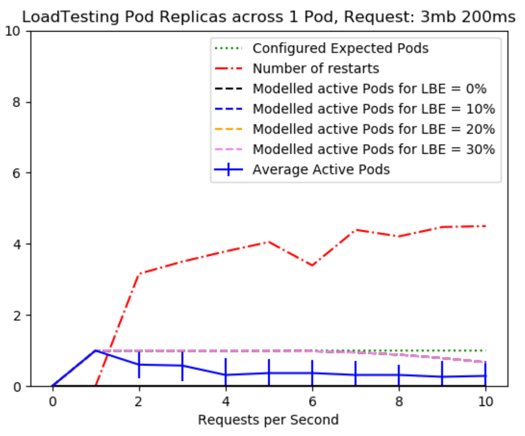


(b) 2 Pod Replicas, Duration 50ms

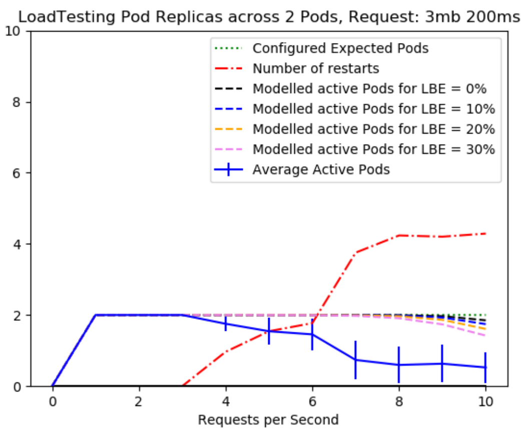


(c) 3 Pod Replicas, Duration 50ms

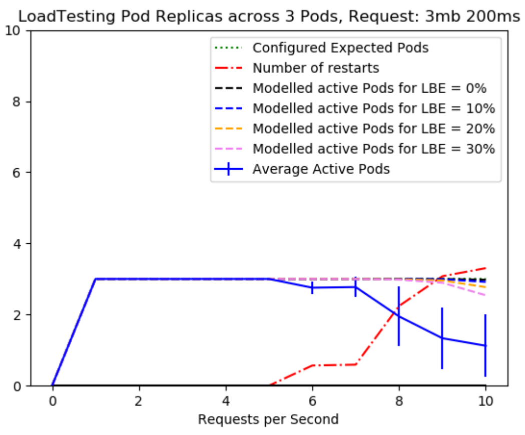
Figure 9.6: Modelled and Observed Replica Status For Scenario B, Request Duration 50ms



(a) 1 Pod Replica, Duration 200ms



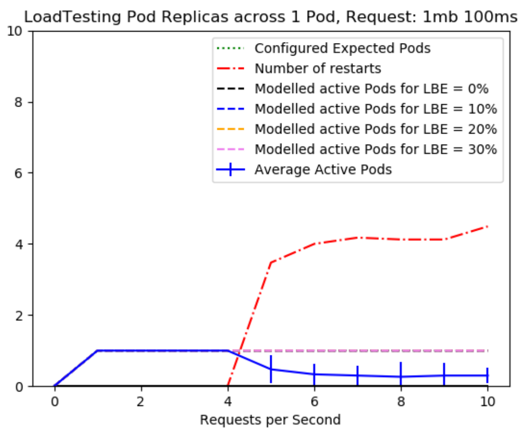
(b) 2 Pod Replicas, Duration 200ms



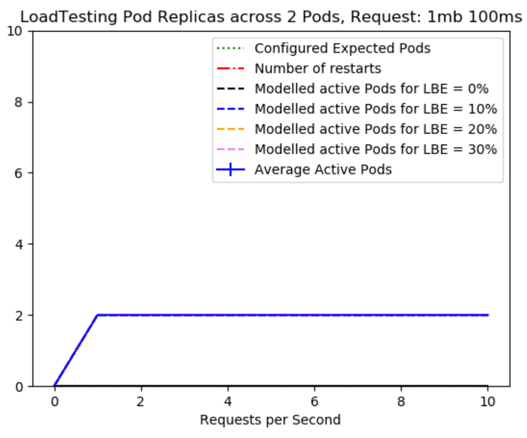
(c) 3 Pod Replicas, Duration 200ms

Figure 9.7: Modelled and Observed Replica Status For Scenario B, Request Duration 200ms

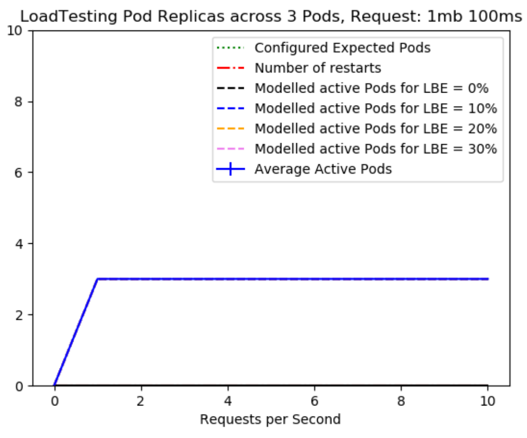
Figures 9.6, 9.3, and 9.7 present comparative performance across requests of 50, 100, and 200ms respective durations. The reader will observe that our generated system models began overperforming as the duration of requests was increased. We see from results presented in Figure 9.9b that overshoot of modelled performance was lower than the previous single pod configuration of 9.3a. Models produced for deployment configurations of 2 and 3 pods undershot measured performance for requests of 50ms duration. Conversely, Figure 9.7 shows that modelled performance for requests of 200ms duration overshot measured performance.



(a) Request Size 1mb, Duration 100ms

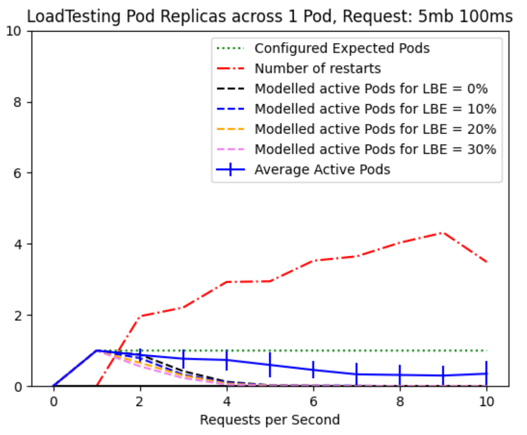


(b) Request Size 1mb, Duration 100ms

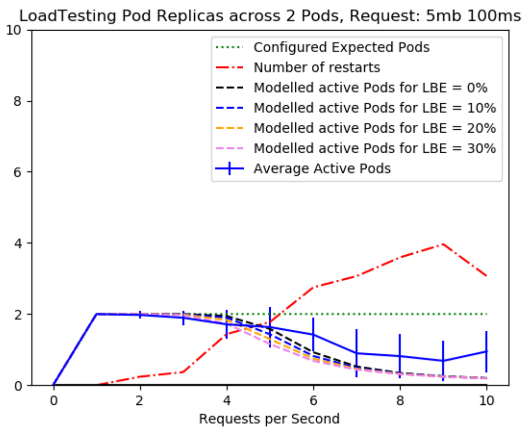


(c) Request Size 1mb, Duration 100ms

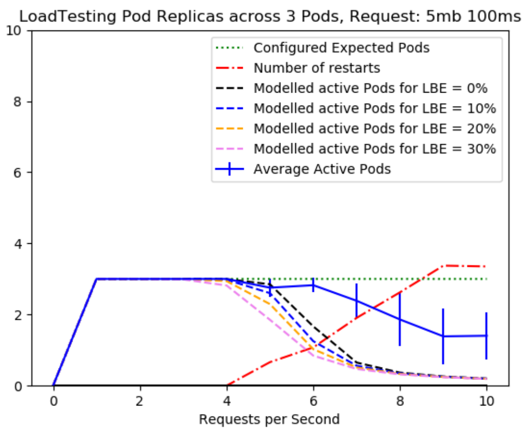
Figure 9.8: Modelled and Observed Replica Status For Scenario B, Request Size 1mb



(a) Request Size 5mb, Duration 100ms



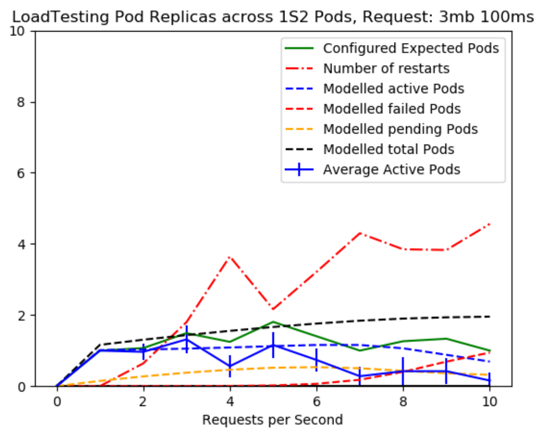
(b) Request Size 5mb, Duration 100ms



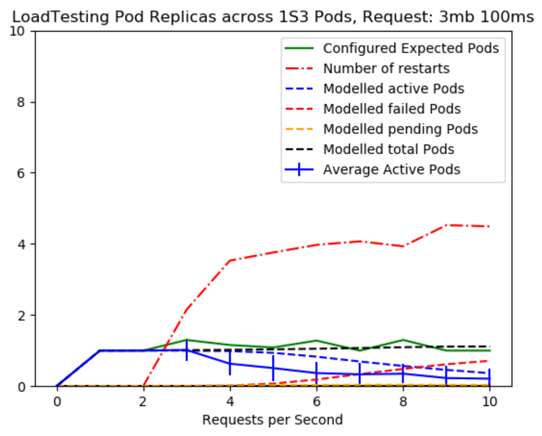
(c) Request Size 5mb, Duration 100ms

Figure 9.9: Modelled and Observed Replica Status For Scenario B, Request Size 5mb

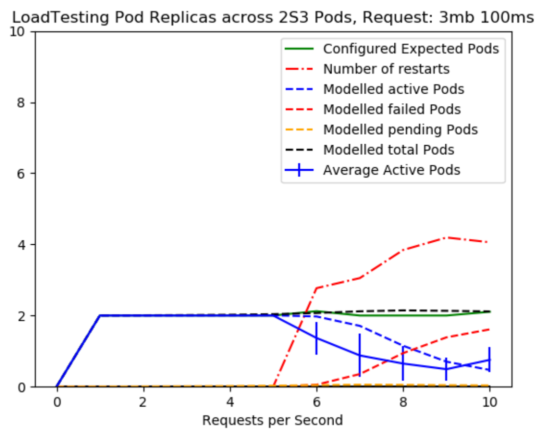
Figures 9.8, 9.3, and 9.9 present comparative performance across requests with sizes of 1, 3, and 5mb respectively. We observe that overshoot of modelled performance was reduced for single pod configurations as request size increased. Modelled performance was reasonably consistent for deployment configurations of 2 pod replicas across request sizes. Modelled performance for configurations with 3 pod replicas undershot that of the deployed application for request sizes of 3 and 5mb.



(a) Scaling 1 - 2 Pods, Deployment Scenario B



(b) Scaling 1 - 3 Pods, Deployment Scenario B



(c) Scaling 2 - 3 Pods, Deployment Scenario B

Figure 9.10: Modelled and Observed Replica Status For Scenario B, Autoscaling

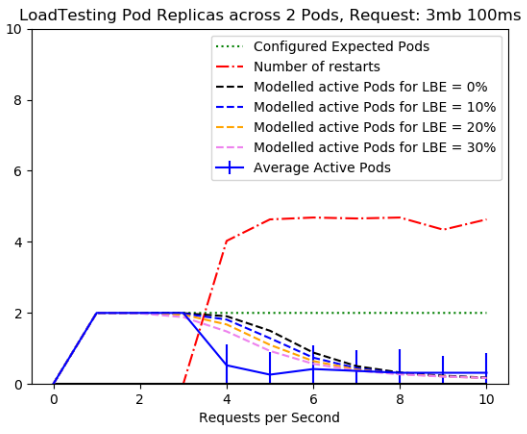
Figure 9.10 depicts the observed status of pod replicas within Scenario B using a Horizontal Pod Autoscaler. A scaling threshold of 40% CPU utilisation was used for scaling actions. This threshold was low to account for the CPU cost of handling requests, which was low in relation to the limits that we set for CPU within our load testing experiments. A low threshold thus increased the chance of engaging in a scaling action prior to encountering OOM failures.

9.2.2 Understanding Results

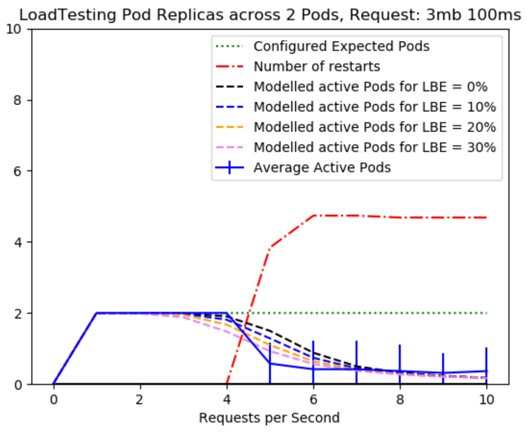
Further examination of observed performance of deployment scenarios was undertaken. This, in conjunction with our knowledge of Kubernetes behaviours, provided insights into further model development areas.

9.2.2.1 Deployment Scenarios

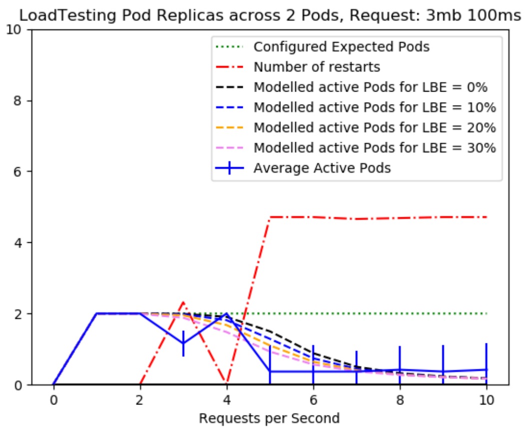
We saw a consistent overshoot across all deployment scenarios configured with a single pod replica, with the exception of Scenario C. In figures from scenarios A, B, and D, we see that observed failures had a tendency to quickly plateau as the rate of incoming requests increased. Kubernetes used an exponential back-off delay when restarting failed pods. There was also a small delay between a pod being marked as active, and a pod handling requests. Consequently, in single pod configurations, we saw that as pods were restarted, requests formed a backlog prior to full pod activation. As pods began handling requests, the processing of these backlogs quickly resulted in pods exceeding their memory allocations. As such, there was little time between pods being restarted and subsequently failing again. The combination of these factors resulted in a maximum cap for number of restarts observed by a single pod within the duration of a single experimental round. We anticipate that performance of models that are adjusted to account for the discussed exponential delay would produce figures with much steeper points of inflection for active pods.



(a) Replica Status for Round 1



(b) Replica Status for Round 2



(c) Replica Status for Round 3

Figure 9.11: Observed Replica Status Across Experimental Rounds for Scenario A with 2 Pod Replicas, Requests of 3mb 100ms Duration

Examination of results produced by individual rounds of experiments shown in Figure 9.11 reveals that the incoming request rate associated with pod replica failures was not consistent across all experimental rounds. We attribute this in part to inconsistent efficacy of load balancing. As previously discussed, modelled performance did not capture the effect of exponential backoff delays. The reader may recall from previous discussion around Figure 7.8 that we observed errors with balancing of resource utilisation across pod replicas. As pod replicas approached their memory capacities in Figure 9.11, poorly balanced load contributed to the failure of individual pod replicas. We observe a spike in failure that reflects the exponentially delayed restart of effected pod replicas.

Scenario	Replicas	LBE 0	LBE 10%	LBE 20%	LBE 30%
A	1	0.2838	0.2838	0.2838	0.2838
A	2	0.2419	0.1591	0.09870	0.5605
A	3	0.0971	0.05368	0.07489	0.1369
B	1	0.1192	0.1192	0.1192	0.1192
B	2	0.07119	0.05108	0.03433	0.01982
B	3	0.2914	0.3350	0.3894	0.4595
C	1	0.009016	0.009016	0.009016	0.009016
C	2	0.09167	0.1163	0.1336	0.1549
C	3	0.3276	0.4031	0.4658	0.5406
D	1	0.2161	0.2161	0.2161	0.2161
D	2	0.01107	0.01144	0.009008	0.004461
D	3	0.0	0.0	0.0	0.0

Table 9.4: RMS Errors Across Modelled Deployment Scenarios. Requests of Size 3mb and Duration 100ms

The effect of resource allocation can be seen when we examine the observed performance of our respective deployment scenarios. As allocated resources were increased, we observed an increased maximum incoming request rate prior to pod replica failures. We also observed a gentler decline in the number of active replicas for configurations with more than a single pod replica. Analysis of RMS error between modelled and observed performance presented in Table 9.4 reveals that our generated models began to underperform for situations as resource allocations were increased for deployment configurations involving multiple pod replicas. This indicates that our initial classification of system behaviours did not fully capture the relationship between resource allocation and processing capacity. Further discussion regarding this relationship is presented within the following section on model refinement.

9.2.2.2 Request Configurations

Evaluation over different request durations revealed that our classification produced models that underperformed against requests of short duration, but overperformed against requests of long duration. Knowledge of our deployed application provides insight as to why this is the case. Analysis of graphs in figures 9.6, 9.3, and 9.7 shows that the failure point of pods within the same scenario was consistent for requests of the same size, regardless of duration. The CPU coefficient produced during the classification process related to concurrent levels of request handling. A decreased request duration necessitated a higher rate of incoming requests to maintain the same level of concurrency. We present this equivalency in the below table. The information presented in

Scenario	Size	Duration	CPU Coefficient	Equivalent Requests/Second
B	3mb	50ms	441	20
B	3mb	100ms	352	10
B	3mb	200ms	219	5

Table 9.5: Duration and Equivalent Incoming Rate of Requests at a Concurrency Level of 1.

Table 9.5 reveals that the CPU cost associated with handling requests of the same size is correlated to the incoming rate of said requests. Further examination of running containers revealed that actual utilisation varied based on a number of factors:

- There was a stable cost associated with allocating the amount of memory specified within the request. This cost is proportional to the amount of memory used by the request.
- There was a stable cost associated with receiving requests and sending responses.
- There was an unknown cost when GC processes were engaged.

An increased rate of incoming requests resulted in an increase in associated request handling actions. This resulted in greater heap utilisation, and consequently increased engagement of GC processes. As such, the average CPU utilisation of pod replicas was observed to increase in line with the rate of incoming requests. Figures 9.6, 9.3, and 9.7 therefore model performance of the same microservice application using CPU coefficients that reflect average utilisation across pod processes at incoming request rates of 5, 10, and 20 respectively. We note that the performance of modelled systems in the case of incoming rates of 5 and 10 requests per second is consistent with observed behaviours of experimental rounds using the same rates.

Analysis of results presented in figures 9.8, 9.3, and 9.9 reveals that there was a significant decrease in observed performance when request size was increased from 1mb to 3mb. Performance was observed to be reasonably consistent when request size was increased from 3mb to 5mb, although pod replica failures were observed at lower rates of incoming requests for requests of 5mb. Analysis of resource utilisation for individual pod replicas across request configurations revealed CPU throttling to be a contributing factor to this. Heapsize of individual pod replicas was seen to approach the configured memory limit. Individual pods experience a large spike in CPU utilisation, followed by failures. This utilisation spike was attributed to the engagement of GC processes for heap management. As request sizes increased, there was a correlating increased in the rate at which GC processes were engaged. Once CPU utilisation exceeded configured pod CPU limits, throttling of internal pod processes was engaged. The reader may recall from previous chapters that throttling of processes has a compounding effect on performance. CPU utilisation of heap management processes for pod replicas handling requests of size 1mb did not exceed configured limits when multiple pod replicas were deployed. When request size was increased to 3mb, we saw that CPU utilisation spikes associated with these GC processes exceeded configured CPU limits. Consequent throttling resulted in memory utilisation exceeding configured limits, at which point pods failed with an OOM error. When request size was increased to 5mb, we observed that pod replicas encountered failures at lower incoming request rates. Again, further examination revealed the same throttling behaviours as in the case of 3mb requests. We attributed the reasonably consistent behaviour of pod replica configurations across both request sizes to our knowledge of how throttling impacts performance. Utilisation spikes lead to a reduced ability to process incoming requests. We observed that when throttling was engaged, there was a compounding impact on the observed CPU utilisation as processing was deferred from one CPU period to the next. While throttling was engaged at lower incoming rates for requests of size 5mb when compared to those of 3mb, the compounding impact on subsequent processes along with previously discussed behaviour relating to exponential restart delays resulted in similar observed performance once throttling was encountered. Comparison between observed performance of scenarios B and D in figures 9.3 and 9.5 respectively illustrates that increased CPU allocation mitigated performance issues related to throttling behaviours.

Figure 9.10 depicts the performance of pod replicas in Scenario B under scaling conditions. The reader will note the divergence of the configured expected vs observed active pod counts as scaling actions were engaged. This occurred when scaling was engaged, but a pod replica encountered a failure prior to the new scaled pod being ready to service requests. Delays in attempted restarting of said pod resulted in a lower number of observed active replicas than that of expected. This resulted in a situation where scaling occurred, but we observed oscillations between pods being active or failed. Load balancing did not factor much into the predicted failure of our system when we scaled from a base pod count of 1. As the reader may recall, load balancing errors do not contribute to performance for a single pod. The failure of pods following scaling actions resulted in a situation where we often observed only a single active pod replica even when our deployment was scaled. Performance of modelled deployment configurations in Figure 9.10 reflects this behaviour. Our produced models reflect the expected combination of states across scaling pod replicas.

9.2.3 Further Model Refinement

Evaluation of the performance of our modelled systems against the behaviours observed during our load testing experiments indicates that our approach of approach of generating probabilistic models from our system classification process has potential for predicting performance. We propose two refinements that we anticipate will improve upon the suitability of produced models.

9.2.3.1 Restart Delays

We observed from our load testing experiments that modelled performance did not capture the impact of delays between restart attempts on the active status of pod replicas. As discussed, Kubernetes uses an exponential backoff delay before attempting to restart failed pods. The effect of this was particularly apparent from observed experimental performance in configurations consisting of a single pod replica, or in deployment scenarios that encountered throttling. Initially, there is a 10s delay between an observed failure and attempted restart.

If the pod replica fails within 10 minutes of the previous restart, this delay is doubled up to a maximum of 5 minutes. If a pod is active for more than 10 minutes without encountering a failure, this delay is reset to 10. We propose the future inclusion of a new component to accompany deployment queues within our probabilistic model along the following logic:

```
Pa_F0 = (a_fail, infty).Pa_F1;
Pa_F1 = (a_fail, infty).Pa_F2 + (a_restart, 0.1).Pa_F1
    + (reset, 0.0.16).PaF0;
Pa_F2 = (a_fail, infty).Pa_F3 + (a_restart, 0.05).Pa_F2
    + (reset, 0.0.16).PaF0;
.
.
.
Pa_F6 = (a_fail, infty).Pa_F6 + (a_restart, 0.0032).Pa_F1
    + (reset, 0.016).PaF0;
```

Synchronisation between the above outlined component and failure/restart actions of modelled deployments is expected to better capture the impact of such delays through variation in the rate with which restart actions are fired. The *infty* rate associated with failure means that synchronisation of components on these actions does not impact performance. Rates associated with other activities represent the number of times per second each activity is expected to be fired. Synchronisation over restart actions means transition firing is determined by the lowest rate across activities of synchronised components. We anticipate that the addition of such a component will result in a steeper gradient about the point of inflection for modelled active pods.

9.2.3.2 Classification

We identify two areas of improvement regarding the classification process of our systems.

We observed from the results of our load testing across requests of different durations that the CPU utilisation of our microservice application was impacted by the rate at which requests entered the system. We learned from further analysis that this was indicative of a more complex relationship between JVM memory management and CPU utilisation. We propose refining our classification process to better capture this relationship. Our initial approach calculated service rates based on average CPU utilisation identified through concurrency testing. Consequent modelled performance was reflective of incoming request rates required to meet concurrency levels. The next step in refining this approach is to identify CPU coefficients based on the rate of incoming requests rather than the cost of concurrent request handling.

9.2.4 Modelling at Runtime

Models were produced automatically to fit deployment scenarios and configurations. The statespace and derived and analysed for each modelled system. This was repeated for incoming request rates ranging from 1-10. Statespaces and steadystate probabilities were exported as csv files. A filtering spreadsheet was used to automate the evaluation of pod replica state combinations across each set of models. The next implementation stage of our architecture is the automation of statespace generation and analysis. Following this is the incorporation of automated classification and modelling processes as runtime components. Finally is the application of modelled strategies to a system at runtime.

Chapter 10

Conclusions

Within this research, we established the increasing complexity associated with the management of distributed cloud systems. We discussed the need for Cloud Service Providers to adhere to defined SLOs to maintain profitability, and how this often leads to the underprovisioning of cluster resources. We established the role of self-adaptive systems within the context of cloud management and our aim of providing guarantees of predicted performance under uncertainty. We presented an architecture for a predictive self-management component of a Kubernetes Cluster. We presented work toward runtime modelling of microservice performance within a Kubernetes Cluster. We established a mechanism to facilitate automated classification of microservice properties. We presented a DES model using EFSMs that can be easily adapted to represent different cluster and deployment configurations at design time. We verified properties of this model and presented a generator tool that automates the creation of a probabilistic model using the PEPA syntax. We compared the performance of our modelled system against the observed behaviours of deployed application scenarios. We determined that our presented results indicate the potential for such a modelling approach to provide guarantees of performance under uncertainty. Finally, we identified areas of future refinement for our classification and modelling processes, and discussed the next implementation phases of our self-adaptive architecture.

10.1 Appraisal of Research Questions

At the outset to this thesis, we established three questions that we sought to answer during our research process. They are reiterated as follows.

1. How accurately are we able to capture the behaviours of the Kubernetes management platform using EFSMs?
2. Can we classify aspects of microservice applications to generate equivalent performance models?
3. Is there potential for such models to be used for predictive management?

With regard to our first research question, we find that our approach to modelling components as EFSMs provides a useful mechanism through which to plan potential system configurations. It captures the possible scaling of tenant pods, and establishes potential configurations of pod replicas at the cluster level. It also serves to model the allocation of a resource pool across distributed applications. Model properties were checked using the WATERS verification framework. Our produced model can be quickly adapted to suit different application scenarios based on known information. It allows for the visualisation of possible failure conditions under differing degrees of load balancing error. We suggest that our EFSM model serves as a useful tool for predicting limits and the state of pods within a cluster at design time.

With regard to our second research question, we find that evaluation of probabilistic models using a PEPA syntax has potential for performance prediction of distributed cloud systems. We established the potential for automated approaches to microservice classification and presented a tool to automatically produce models across a variety of deployment configurations. Examination of experimental results revealed potential areas of refinement to improve the suitability of produced models. We suggest that our modelling approach demonstrates potential for modelling performance of Kubernetes clusters.

With regard to our third research question, we find that comparisons between modelled and observed performance illustrated the potential of our automatic modelling approach in predicting performance of microtenant applications. Our experimental evaluations provided new insights into future refinement of our classification and modelling approaches. We anticipate that further development of these techniques will improve upon the accuracy with which performance is predicted.

10.2 Threats to Validity

We recognise that there are threats to the validity of the presented research due to limitations of our approach. We seek to address these limitations within this section.

The first limitation comes as an issue of scale. As has been discussed within this work, we seek to model cluster environments that house applications following a microservice architecture. The reason for this is largely due to the issues relating to the generation of explosive state spaces. Due to limitations in available computational power, we have not tested our approach on larger systems, and are unable to provide assurances as to the accuracy of this approach with regard to modelling systems of scale. We would seek to explore this approach in future.

We establish experimental environments using Minikube as our cluster management software. Our experiments deal explicitly with local resources, which does not account for factors that contribute to performance within a distributed Kubernetes architecture such as communication latency between nodes, or issues relating to inconsistent infrastructure. As clusters grow in size, the impact of these issues is expected to grow in line with the complexity of the managed system. Capturing this level of complexity within a modelled scenario is an area for future development, but goes beyond the scope of what we present in this research.

We acknowledge that there is a limitation relating to the generation of load to test our deployed applications. PEPA utilises exponentially distributed rates for transitions, including rates relating to request generation and handling. Within our load testing scripts, we make use of constant request rates that increase incrementally from 1-10 per second. Such a consistent load pattern is not reflective of what one might expect within a real-world scenario. As such, the evaluation of the efficacy of our modelling approach is not indicative of accuracy when the deployed system encounters more dynamic load patterns. This is an area that has room for future development.

The calculation of service rate based on our CPU coefficient does not accurately capture the complex relationship between memory management and CPU utilisation. In future, we seek to evaluate this approach under application scenarios that are not memory intensive as well as develop a more robust algorithm for calculating CPU utilisation.

Time constraints were a limiting factor within this research. Each experimental round required a minimum of 12 minutes to run, allowing for pods to become active and requests to be sent. We ran a minimum of 30 such rounds per number of pod replicas for each application scenario. Use of logical formulae contributed to the automated filtering of generated statespaces. Further development of these techniques would reduce associated processing time.

10.3 Further Future work

We have identified the room for further development of our CPU coefficient calculation that forms the basis of our performance evaluation. We have proposed a potential approach that accounts for scaling of associated CPU utilisation based on the observed rate of incoming requests. We also recognise that performance interference of microservices due to GC processes is an area of increasing research interest [70]. We anticipate that performance modelling of GC interactions is a potential area for future expansion.

We see potential for future extensions toward performance evaluation of Kubernetes clusters at design time. A major factor in performance across large distributed systems is network latency. This leads to request backlogs and load spikes. Our modelling approach is suited to microservices distributed across local resources. As such, it does not make allowances for delays based on infrastructure. There is scope for future work to evaluate our modelling approach across clusters that span multiple nodes.

The next phase of our architectural implementation involves further development regarding the automation of system classification and subsequent model generation processes. We anticipate future research into the suitability of our modelling approach for self-adaptive management.

References

[1] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 16(3):33–38, 2019.

[2] Knut Åkesson, Martin Fabian, Hugo Flordal, and Robi Malik. Supremica-an integrated environment for verification, synthesis and simulation of discrete event systems. In *2006 8th International Workshop on Discrete Event Systems*, pages 384–385. IEEE, 2006.

[3] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015.

[4] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5(1):11, 2014.

[5] Francesco Maria Aymerich, Gianni Fenu, and Simone Surcis. An approach to a cloud computing network. In *2008 First International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*, pages 113–118. IEEE, 2008.

[6] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *2010 18th IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, 2010.

[7] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes. *CWI Monograph series*, 3:89–138, 1986.

[8] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[9] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[10] Michael Boniface, Bassem Nasser, Juri Papay, Stephen C. Phillips, Arturo Servin, Xiaoyu Yang, Zlatko Zlatev, Spyridon V. Gogouvitis, Gregory Katsaros, Kleopatra Konstanteli, et al. Platform-as-a-service architecture for real-time quality of service management in clouds. In *2010 Fifth International Conference on Internet and Web Applications and Services*, pages 155–160. IEEE, 2010.

[11] Mario Bravetti, Stephen Gilmore, Claudio Guidi, and Mirco Tribastone. Replicating web services for scalability. In *International Symposium on Trustworthy Global Computing*, pages 204–221. Springer, 2007.

[12] Paul C. Brebner. Is your cloud elastic enough? performance modelling the elasticity of infrastructure as a service (iaas) cloud applications. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 263–266, 2012.

[13] Stephen D. Brookes, Charles A. R. Hoare, and Andrew W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[14] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[15] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.

[16] Radu Calinescu, Kenneth Johnson, and Yasmin Rafiq. Using observation ageing to improve markovian model learning in qos engineering. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, pages 505–510, 2011.

[17] Radu Calinescu, Yasmin Rafiq, Kenneth Johnson, and Mehmet Emin Bakır. Adaptive model learning for continual verification of non-functional properties. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 87–98, 2014.

[18] Emiliano Casalicchio and Vanessa Perciballi. Auto-scaling of containers: the impact of relative and absolute metrics. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 207–214. IEEE, 2017.

[19] Priscila Cedillo, Javier Gonzalez-Huerta, Silvia Mara Abrahão, and Emilio Insfran. Towards monitoring cloud services using models@ run. time. In *MoDELS@ Run. time*, pages 31–40, 2014.

[20] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai. Sla decomposition: Translating service level objectives to system level thresholds. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 3–3. IEEE, 2007.

[21] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *International Conference on Model Driven Engineering Languages and Systems*, pages 468–483. Springer, 2009.

[22] S. B. Dash, H. Saini, T. C. Panda, and A. Mishra. Service level agreement assurance in cloud computing: a trust issue. 2014.

[23] Petra Dietrich, Robi Malik, W Murray Wonham, and Bertil A Brandin. Implementation considerations in supervisory control. In *Synthesis and control of discrete event systems*, pages 185–201. Springer, 2002.

[24] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. IEEE, 2010.

[25] Jie Ding, Jane Hillston, and David I. Laurenson. Evaluating the response time of large scale content adaptation systems using performance evaluation process algebra. In *2010 IEEE International Conference on Communications*, pages 1–5. IEEE, 2010.

[26] Jie Ding, Leijie Sha, and Xiao Chen. Modeling and evaluating iaas cloud using performance evaluation process algebra. In *2016 22nd Asia-Pacific Conference on Communications (APCC)*, pages 243–247. IEEE, 2016.

[27] John C. Doyle, Bruce .A Francis, and Allen R. Tannenbaum. *Feedback control theory*. Courier Corporation, 2013.

[28] E. Allen Emerson. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, pages 27–45. Springer, 2008.

[29] Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 234–244, 2011.

[30] Naeem Esfahani and Sam Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 214–238. Springer, 2013.

[31] Kousha Etessami, Marta Kwiatkowska, Moshe Y Vardi, and Mihalis Yannakakis. Multi-objective model checking of markov decision processes. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 50–65. Springer, 2007.

[32] Alexandros Evangelidis, David Parker, and Rami Bahsoon. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems*, 87:629–638, 2018.

[33] Martin Fabian. Lecture notes in discrete event systems, 2006.

[34] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *2013 IEEE Sixth International Conference on cloud computing*, pages 887–894. IEEE, 2013.

[35] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, pages 299–310, 2014.

[36] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas d'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, et al. Software engineering meets control theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 71–82. IEEE Press, 2015.

[37] Joel Gibson, Robin Rondeau, Darren Eveleigh, and Qing Tan. Benefits and challenges of three cloud computing service models. In *2012 Fourth International Conference on Computational Aspects of Social Networks (CASoN)*, pages 198–205. IEEE, 2012.

[38] Mor Harchol-Balter. Performance modeling and design of computer systems: Queuing theory in action. 2014.

[39] Robert Heinrich, Eric Schmieders, Reiner Jung, Kiana Rostami, Andreas Metzger, Wilhelm Hasselbring, Ralf Reussner, and Klaus Pohl. Integrating run-time observations and design component models for cloud system analysis. 2014.

[40] Jane Hillston. Compositional Markovian modelling using a process algebra. In *Computations with Markov chains*, pages 177–196. Springer, 1995.

[41] Jane Hillston. *A compositional approach to performance modelling*, volume 12. Cambridge University Press, 2005.

[42] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[43] Yexi Jiang, Chang-shing Perng, Tao Li, and Rong Chang. Asap: A self-adaptive prediction system for instant cloud resource demand provisioning. In *2011 IEEE 11th International Conference on Data Mining*, pages 1104–1109. IEEE, 2011.

[44] Yexi Jiang, Chang-shing Perng, Tao Li, and Rong Chang. Self-adaptive cloud capacity planning. In *2012 IEEE Ninth International Conference on Services Computing*, pages 73–80. IEEE, 2012.

[45] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Multilayered cloud applications autoscaling performance estimation. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 24–31. IEEE, 2017.

[46] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32, 2019.

[47] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. An incremental verification framework for component-based software systems. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 33–42, 2013.

[48] Kenneth Johnson, Simon Reed, and Radu Calinescu. Specification and quantitative analysis of probabilistic cloud deployment patterns. In *Haifa Verification Conference*, pages 145–159. Springer, 2011.

[49] Michael J. Kavis. *Architecting the Cloud.* 2014.

[50] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[51] Khaled Khebbeb, Nabil Hameurlain, Faiza Belala, and Hamza Sahli. Formal modelling and verifying elasticity strategies in cloud systems. *IET Software*, 13(1):25–35, 2018.

[52] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE'07)*, pages 259–268. IEEE, 2007.

[53] Ajay D. kshemkalyani and Mukesh Singhal. *Distributed Computing. Principles, Algorithms, and Systems.* Cambridge University Press, 2008.

[54] Marta Kwiatkowska and David Parker. Automated verification and strategy synthesis for probabilistic systems. In *Automated Technology for Verification and Analysis*, pages 5–22. Springer, 2013.

[55] Siva Theja Maguluri, Rayadurgam Srikant, and Lei Ying. Stochastic models of load balancing and scheduling in cloud computing clusters. In *2012 Proceedings IEEE Infocom*, pages 702–710. IEEE, 2012.

[56] Robi Malik, Knut Åkesson, Hugo Flordal, and Martin Fabian. Supremica–an efficient tool for large-scale discrete event systems. *IFAC-PapersOnLine*, 50(1):5794–5799, 2017.

[57] Robi Malik, Martin Fabian, and Knut Åkesson. Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata. *IFAC Proceedings Volumes*, 44(1):7000–7005, 2011.

[58] V. Medel, R. Tolosana-Calasanz, J. Á. Bañares, U. Arronategui, and O. F. Rana. Characterising resource management performance in kubernetes. In *Computers & Electrical Engineering*, volume 68, pages 286–297, 2018.

[59] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 257–262, 2016.

[60] Sahar Mohajerani, Robi Malik, and Martin Fabian. A framework for compositional nonblocking verification of extended finite-state machines. *Discrete Event Dynamic Systems*, 26(1):33–84, 2016.

[61] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 1–12, 2015.

[62] Lucien Ouedraogo, Ratnesh Kumar, Robi Malik, and Knut Akesson. Nonblocking and safe control of discrete-event systems modeled as extended finite automata. *IEEE Transactions on Automation Science and Engineering*, 8(3):560–569, 2011.

[63] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 2017.

[64] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures–a technology review. In *2015 3rd international conference on future internet of things and cloud*, pages 379–386. IEEE, 2015.

[65] Panagiotis Patros, Dayal Dilli, Kenneth B. Kent, and Michael Dawson. Dynamically compiled artifact sharing for clouds. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 290–300. IEEE, 2017.

[66] Panagiotis Patros, Dayal Dilli, Kenneth B. Kent, Michael Dawson, and Thomas Watson. Multitenancy benefits in application servers. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 111–118, 2015.

[67] Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Investigating the effect of garbage collection on service level objectives of clouds. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 633–634. IEEE, 2017.

[68] Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Slo request modeling, reordering and scaling. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, pages 180–191, 2017.

[69] Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Mitigating garbage collection interference on containerized clouds. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 168–173. IEEE, 2018.

[70] Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Why is garbage collection causing my service level objectives to fail? *International Journal of Cloud Computing*, 7(3-4):282–322, 2018.

[71] Panagiotis Patros, Stephen A. MacKay, Kenneth B. Kent, and Michael Dawson. Investigating resource interference and scaling on multitenant PaaS clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 166–177, 2016.

[72] Vladimir Podolskiy, Michael Mayo, Abigail Koey, Michael Gerndt, and Panos Patros. Maintaining SLOs of cloud-native applications via self-adaptive resource sharing. In *2019 IEEE 13th International Conference on*

Self-Adaptive and Self-Organizing Systems (SASO), pages 72–81. IEEE, 2019.

[73] Vladimir Podolskiy, Maria Patrou, Panos Patros, Michael Gerndt, and Kenneth B Kent. The weakest link: revealing and modeling the architectural patterns of microservice applications. 2020.

[74] Nigel Poulton. *The Kubernetes Book*. Independently published, 2017.

[75] Peter J. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

[76] Georgia Sakellari and George Loukas. A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing. *Simulation Modelling Practice and Theory*, 39:92–103, 2013.

[77] Leijie Sha, Jie Ding, Xiao Chen, Xiaobin Zhang, Yun Zhang, and Yishi Zhao. Performance modeling of openstack cloud computing platform using performance evaluation process algebra. In *2015 International Conference on Cloud Computing and Big Data (CCBD)*, pages 49–56. IEEE, 2015.

[78] Doaa M Shawky. Performance evaluation of dynamic resource allocation in cloud computing platforms using stochastic process algebra. In *2013 8th International Conference on Computer Engineering & Systems (ICCES)*, pages 39–44. IEEE, 2013.

[79] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[80] Alireza Souri, Nima Jafari Navimipour, and Amir Masoud Rahmani. Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review. *Computer Standards & Interfaces*, 58:1–22, 2018.

[81] Marcelo Teixeira, Robi Malik, José ER Cury, and Max H de Queiroz. Supervisory control of DES with extended finite-state machines and variable abstraction. *IEEE Transactions on Automatic Control*, 60(1):118–129, 2014.

[82] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

[83] Martin van Zijl. Model checking for cloud autoscaling using WATERS. Master's thesis, The University of Waikato, 2020.

[84] Danny Weyns. Software engineering of self-adaptive systems: an organised tour and future challenges. *Chapter in Handbook of Software Engineering*, 2017.

[85] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer, 2013.

[86] Jiapeng Zhu, Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Node. js scalability investigation in the cloud. In *CASCON 2018*, pages 201–212. ACM, 2018.