CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Research Commons@Waikato

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Model Checking for Cloud Autoscaling using WATERS

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Master of Science (Research) in Computer Science

at

The University of Waikato

by

Martin van Zijl

2020

Abstract

This thesis investigates the use of formal methods to verify cloud system designs against Service Level Agreements (SLAs), towards providing guarantees under uncertainty. We used *WATERS* (the *Waikato Analysis Toolkit for Events in Reactive Systems*), which is a model-checking tool based on discrete event systems. We created models for one aspect of cloud computing, horizontal autoscaling, and used this to verify cloud system designs against an SLA that specifies the maximum request response time.

To evaluate the accuracy of the WATERS models, the cloud system designs are simulated on a private *Kubernetes* cluster, using *JMeter* to drive the workload. The results from Kubernetes are compared to the verification results from WATERS. A key research goal was to have these match as closely as possible, and to explain the discrepancies between the two. This process is followed for two applications: a default installation of NGINX, a web server with a fast but variable response time, and a hand-written Node.js program enforcing a fixed response time.

The results suggest that WATERS can be used to predict potential SLA violations. Lessons learned include that the state space must be constrained to avoid excessive checking times, and we provide a method for doing so. An advantage of our model checking-based technique is that it verifies against all possible patterns of arriving requests (up to a given maximum), which would be impractical to test with a load testing tool such as JMeter.

A key difference from existing work is our use non-probabilistic finite state machines, as opposed to probabilistic models which are prevalent in existing research. In addition, we have attempted to model the detail of the autoscaling process (a “white-box” approach), whereas much existing research attempts to find patterns between autoscaling parameters and SLA violation, effectively viewing autoscaling as a black-box process.

Future work includes refining the WATERS models to more closely match Kubernetes, and modelling other SLO types. Other methods may also be used to limit the compilation and verification time for the models. This includes attempting different algorithms and perhaps editing the models to reduce the state space.

Acknowledgements

I would like to thank Dr. Robi Malik and Dr. Panos Patros for guidance in producing this thesis, and Vladimir Podolskiy and Stephen Burroughs for their advice on cloud computing and Kubernetes. I would also like to thank the creators of the WATERS, JMeter and Kubernetes software.

Contents

1 Introduction **1**

2 Background **4**
2.1 Cloud Computing . . . 4
2.2 Service Level Agreements . . . 9
2.3 Formal Methods . . . 11
2.4 Discrete Event Systems . . . 13
2.5 WATERS . . . 14
2.6 Cloud Management Software . . . 16

3 Related Work **18**
3.1 Existing Frameworks . . . 18
3.2 Overview Papers . . . 25
3.3 Investigative Papers . . . 26

4 Theoretical Model **29**
4.1 Workload . . . 30
4.2 Cloud . . . 30
4.3 SLA . . . 31
4.4 Master . . . 32

5 System Model **34**
5.1 Master . . . 35
5.2 Workload . . . 37
5.3 Cloud . . . 40
5.4 Horizontal Pod Autoscaler . . . 41
5.5 SLA . . . 44
5.6 Differences in Node.js Model . . . 48

6 Experimental Methodology **52**
6.1 Applications . . . 54
6.2 Kubernetes . . . 54
6.3 JMeter . . . 55

6.4 WATERS 57

7 Results and Analysis **60**

7.1 Application 1: NGINX 60

7.2 Application 2: Node.js Program 69

7.3 Discussion 70

8 Conclusions and Future Work **73**

8.1 Future Work 74

Appendices **76**

A Raw Results **76**

A.1 NGINX 76

A.2 Node.js 76

Chapter 1

Introduction

Cloud computing abstracts computing resources, such as memory, disk and CPU. Clients, such as businesses, can use these resources from a *Cloud Services Provider* (CSP) on a pay-as-you-go basis [7]. This is an alternative to a traditional in-house IT infrastructure.

The cloud services provider and client usually agree upon a *Service Level Agreement* (SLA) that defines the expected level of service required from the cloud. The provider aims to ensure that their cloud system meets the SLA, to avoid financial penalties and ensure good reputation [5]. The SLA specifies the *Service Level Objectives* (SLO) that the cloud system must meet.

The task of the cloud administrator is to configure the cloud system to meet the requirements of the SLA. Due to the complexity of cloud systems and uncertainties about the environment (such as frequency of incoming requests), this is an inexact process and prone to error [16]. A method of formally verifying a cloud system design against an SLA would certainly be useful in this process. In this work, we present a prototype for formal verification of one aspect of cloud computing, namely horizontal autoscaling. For this we use WATERS [2] in order to provide formal guarantees of SLA satisfaction under these uncertainties.

The research questions we aimed to address are as follows:

- How accurately can model-checking using WATERS verify that a cloud

design meets its SLA with regard to availability?

- How closely can a WATERS model match Kubernetes?

Our hypothesis was that model checking using WATERS can verify performance requirements to a limited extent, and thereby help ensure more robust cloud system design. We also hypothesise that WATERS can closely match Kubernetes within a limited range of parameters, and under certain assumptions.

The key contributions of this thesis are:

- Prototype models in WATERS to represent horizontal autoscaling in Kubernetes, and a discussion of its accuracy.
- A summary of lessons learned and design principles for formal modelling of cloud systems.

The accuracy of our models is evaluated by comparing the verification results from WATERS to experimental results obtained using JMeter[1] and Kubernetes, in which the cloud system simulated and checked against the SLA.

A key discovery was that model checking allowed exploration of all possible traffic patterns up to a given maximum, which would take a very long time to test using a load-testing tool. Another conclusion was that the state space sometimes has to be deliberately reduced in order to ensure fast compilation and verification times in WATERS.

In terms of the *MAPE-K loop* [17], this work fits within the *Analysis* and *Planning* elements. That is, it analyses if the planned cloud design will meet its objectives. In terms of the Waves of Self-adaptation [69], this work belongs to Waves III (Performance Models) and V (Guarantees Under Uncertainties). This is because our work checks the possible (uncertain) scenarios a cloud system may encounter, and aims to provide formal guarantees as to whether the

[1] `https://jmeter.apache.org/`

system will meet its requirements (SLA) or not. This is done via a performance model.

We envision that this type of model checking can be used by a CSP's self-management module to verify new configurations before deciding to apply them. The planned cloud configuration parameters and the SLA could be run through a WATERS model. If WATERS reports a possibility of an SLA violation, different parameters could be attempted until an acceptable set is found. This type of verification would be especially useful for mission-critical applications, where formal guarantees are required. It could also be used by CSPs to provide stricter SLAs, thus attracting more customers. This research has also been submitted as a conference paper, and is pending review.

The rest of this thesis is structured as follows: Chapter 2 discusses the background including SLAs, formal methods, WATERS and cloud control software. Chapter 3 presents existing work related to this thesis. Chapter 4 describes our theoretical model. Chapter 5 presents the system model. Chapter 6 presents the testing method and experimental setup, and Chapter 7 discusses the execution and results. Finally, Chapter 8 states the conclusions and suggestions for future work.

Chapter 2

Background

This chapter presents the relevant background information for this thesis. Firstly, an overview of cloud computing is given. Secondly, SLAs for cloud systems are discussed. This is followed by an overview of formal methods, including model checking (which is the focus of this thesis). After this discrete event systems are introduced, which are the basis of the models created in WATERS. Next, an overview of the WATERS model checking tool is given. Finally, we present a brief introduction to cloud management software including Kubernetes, which is used as a basis for our WATERS models.

2.1 Cloud Computing

Cloud computing allows clients to host data and software externally in data centres maintained by a cloud services provider (CPS). This is a convenient and popular alternative to traditional IT infrastructure, where the client maintains their own servers, data, and software [7]. A key advantage of cloud computing for the client is that they are only required to pay for the resources actually used (a "pay-as-you-use" model) [7]. From the provider perspective, cloud computing allows profit through economies of scale. Common examples of cloud-based services include social networking and web hosting [12]. Businesses now commonly use cloud systems to host their services [25].

One disadvantage of traditional IT infrastructure is the difficulty in choos-

ing the right number of resources (servers, RAM, CPU, and so on) to purchase. If too few resources are available, requests to the system will be dropped (known as *underprovisioning*). If more resources are invested into than are required to handle maximum load, this represents unnecessary cost (known as *overprovisioning*). Cloud computing overcomes this problem by providing seemingly infinite resources available on demand. That is, the cloud will ensure sufficient resources are allocated to the client to handle all requests, without overprovisioning and thereby incurring unnecessary costs [7].

The term *cloud system* refers to both the infrastructure of the provider and the software running on it. A cloud system typically consists datacenters containing a large amount of commodity servers. Cloud systems have traditionally been based on Virtual Machines (VMs) [29]; the physical servers host VMs, each of which is referred to as a *node*. When a request is made to the cloud, it is delegated to one of the nodes [7] [29]. The nodes are provisioned by the provider to clients as required; this is largely done automatically via cloud management software.

Many modern cloud management systems, such as *Kubernetes*, use *containers* instead of or in addition to VMs [32]. Containers include libraries, software and data, but unlike VMs do not include an operating system, making them more lightweight than VMs and easier to migrate. Multiple containers typically run inside a physical or virtual machine, which acts as their host [29].

Cloud services can be classified into the following types [7]: ***Infrastructure as a Service*** (IaaS): The hardware, such as network, data centre, Virtual Machines (VMs) and so on is provided, but the client must install or choose the operating systems and applications. ***Platform as a Service*** (PaaS): The hardware and operating systems (containers, load balancing, and so on) are provided, but the client must install or choose the applications. Microsoft Azure is one example of a provider of PaaS. ***Software as a Service*** (SaaS): The hardware, operating systems and software are provided. The client simply adds their data. ***Function as a Service*** (FaaS) could also be

added to this list; it executes a single function on demand.

Clouds can be categorised as public, private or hybrid [7]. A *public cloud* makes services available on a pay-as-you-go basis to the general public. *A private cloud*, such as an internal data centre of a business, is not available to the general public. A *hybrid* cloud contains public and private elements.

Theoretically, any type of application may be deployed to a cloud system. For modelling purposes, it is useful to categorise application types into stand-alone, multitier and microservice applications. *Standalone* applications are fully contained in a single node. *Multitier applications* usually provision separate nodes for each layer of the application. For example, a 3-tier application may consist of a database, logic, and presentation layer, each with its own node. Many web applications are structured in a similar manner [16]. *Microservice* applications consist of many individual services, each performing a specific function [32].

Many cloud applications follow a client/server model: clients make requests to the server which runs in a cloud system [48].

Clouds serving more than one client are referred to as *multitenant clouds*, and each client is referred to as a *tenant*. In a multi-tenant cloud two or more tenants may share the same application code on the same node. It is the responsibility of the provider to ensure that each tenant has sufficient resources, and that there is no interference between tenant (this is an important security aspect of cloud computing).

A cloud system is an example of a *self-adaptive system*: that is, a system which adapts itself to changes in the environment and itself [71]. A self-adaptive system consists of a *managed system* and a *managing (controlling) system* [70]. The managing system ensures that the managed system meets its quality objectives. This is performed via perform *self-healing* and *self-adaptation* [41]. Self-healing refers to the automatic resolution of issues to ensure the reliability of the system. In the context of cloud computing, a node which fails may be replaced with a new one automatically. Self-adaptation

refers to modification in structure or behaviour in response to external conditions which are difficult to anticipate at design time.

Self-adaptation in a cloud system is typically performed by cloud management software, such as *Kubernetes*[1] or *Docker Swarm*[2]. The orchestration software usually runs on a special node called the *master*, and is responsible for making ensuring the system runs smoothly. One self-adaptive strategy used by cloud management is automatic scaling (autoscaling), referring to the automatic provisioning of resources to meet demand, which is the focus of this thesis. For example, when the master detects that 80% of the CPU is being used, it could scale by starting another node to lessen the CPU load of the existing nodes.

Three scaling strategies are commonly used in practice [29]: ***Threshold*** scaling performs scaling when certain threshold is reached (such as 80% average CPU consumption across existing servers); in ***Predictive*** scaling, the system predicts beforehand when to apply scaling, typically using algorithms or historic data; ***Seasonal*** scaling applies scaling according to known “busy periods” or seasons (such as end of tax year for financial applications). This thesis focuses on threshold scaling.

A distinction is also drawn between *horizontal scaling* and *vertical scaling*. In *vertical scaling*, more resources are added to existing nodes; this is also referred to as scaling *up* or *down*. In *horizontal scaling*, entire nodes are added or removed; this is also referred to as scaling *in* or *out* [74]. In practice, normally a combination of both is used, but we shall focus on horizontal scaling in this thesis.

Currently auto-scaling policies tend to lack correctness guarantees [21], which is a large motivator for this work.

Cloud management software is also able to manage the number of instances of each microservice in a cloud-based application [32]. For example, if the

[1] https://kubernetes.io/

[2] https://docs.docker.com/engine/swarm/

database nodes have a high load, it can create more database node instances.

Another self-adaptive strategy used within cloud systems is *load balancing*, or delegating requests evenly to servers in a cloud so that all requests can be handled within an acceptable time limit [74, 11]. Common load balancing strategies for cloud systems include *Round Robin*, *Weighted Round Robin*, *Sticky Session*, *Least Connections* and *IP Hash* [30]. In a *Round Robin* strategy, the first requests is sent the first node, the second request to the second node, and so on. This is useful for homogeneous clouds, where all nodes can handle roughly the same amount of work. *Weighted Round Robin* assigns a weight to each node so that more requests are sent to those with higher weight. This is useful for heterogeneous clouds, where different nodes have different resource limits. *Sticky Session* involves "pinning" a user to a node so that all requests from that particular user are sent to the same node. This is useful in stateful applications. Using *Least Connections*, the node with the least connections open gets the next request. This is useful for stateful applications. Using *IP Hash*, each request has a hash computed for it based on the IP address. This hash maps to a particular node, and the request is then sent to that node. In the models presented in this thesis, we shall assume Round Robin load balancing is used.

There are concerns which hinder the adoption of cloud computing. One example is *reputation sharing*: In a multitenant cloud, if a datacenter is compromised due to one misbehaving client, other clients may also be affected also [7]. For example, if data from a cloud system must be confiscated for one client, it will also be confiscated for other clients sharing the same resources [29]. Another concern is whether cloud computing provides adequate availability, since systems are often expected to be "always available" [7]. Another notable concern is whether cloud computing provides adequate security adaptation [7, 10] including confidentiality of data in a multi-tenant cloud, and in the context of attacks such Distributed Denial of Service (DDoS) attacks and Man-in the-Middle attacks during VM cloning [29]. The large datacenters

used in cloud systems also tend to consume a vast amount of power, which has a potentially adverse effect on the environment [29].

2.2 Service Level Agreements

A cloud system is usually created by a *provider* for a *client.* Examples of a cloud service providers include AWS (*Amazon Web Services*)[3], *IBM Bluemix*[4] and *Microsoft Azure*[5].

In order to ensure that the cloud system is acceptable, the client and provider typically create a *service level agreement (SLA)* [11] to define the responsibilities of the client and the provider. In particular, the SLA specifies the *non-functional requirements* of the cloud system [38], called *Service Level Objectives* (SLOs). The SLA also defines escalation procedures and financial penalties (costs) for the provider if the SLOs are not met [16].

An SLO normally specifies a *Service Level Indicator* (SLI) and its required value [11] [5]. An SLI is a measurement of the performance of a cloud system, such as the *average response time* of the cloud to incoming requests. Other examples of SLIs include the *error rate* (number of failed requests divided by the total number of requests) and *monthly up-time percentage*, which are listed on the SLA for AWS[6].

SLOs are often defined in terms of *percentiles* [63] instead of averages or absolute maximum or minimum values. For example, the SLO "the 99% percentile or response time must be less than 1 second" implies that it is allowable for the top 1% of requests to have response times of 1 second or more. This is partly because it is difficult to avoid "outliers" in practice. Possible examples of SLOs are:

- The system must be available 99.999% of the time (the "5 nines" rule) [25].

[3]https://aws.amazon.com/

[4]https://www.ibm.com/cloud/

[5]https://azure.microsoft.com/en-us/

[6]https://aws.amazon.com/s3/sla/

(Availability)

- The failure rate at most 1 in 10. (Reliability)

- The 99th percentile of response time must be less than 1 second. (Performance)

- The average CPU utilisation must be less than 80%. (Performance)

In this thesis, we shall focus on the *maximum response time*.

SLA often specify an *error budget*, which represents the amount of SLO violations which may occur within a rolling time window [5].

It is useful to categorise SLO requirements into *Quality Attributes* (QAs):

- Functionality: The system should provide the behaviour specified by its *functional requirements* [59].

- Reliability: The system maintains an adequate level of performance while running in all specified conditions [41].

- Availability: The system performs its required functions at the required points in time. This requires *maturity* (the ability to avoid failure), *fault tolerance* and *recoverability* (the ability to re-establish a specified level of performance after failure) [71].

- Fault-tolerance: The system should recover from *internal* faults [71] [59].

- Survivability: The system should survive a disaster or outage (these may be viewed as *external* faults).

- Performance: The response time (how long the system takes to respond to a request) and throughput (how many requests are served per unit time) of the system should be acceptable.

- Security: Only authorised parties should have access to data (*confidentiality*) [23], data must not be tampered with (*integrity*), and must always be available to its intended users (*availability*) [60].

- Maintainability: The system should be easily updated to meet new specifications.
- Replaceability: Components in the system should be replaced easily [71, 59].
- Cost: The system should be cost-efficient. In the context of cloud computing, financial penalties due to SLA violations and losses due to overprovisioning should be kept low.
- Power Efficiency: Power consumption should be kept low both to save cost and to reduce environmental impact.

Availability is the QA that is the focus of this thesis.

One disadvantage of textual SLAs is that they are difficult to verify. In particular, it is difficult to use a program to determine whether a system meets a textual SLA. Therefore it is useful to translate a textual SLA into a set of formal requirements. To this end, formal SLA languages have been developed, such as *SLAng* [38] and *SLAC* [64].

2.3 Formal Methods

Formal methods are mathematical techniques to check the validity of a system [61]. They are used to verify a system to eliminate design mistakes [27]. Formal methods are normally used to verify critical software, where failure must be avoided. Formal methods may be applied at design time (offline) or at runtime [70].

Model checking [18] is one type of formal method. This consists of creating a model of the system and a specification of the requirements, then verifying if the system actually meets the requirements. A key advantage of model checking is that the verification can be done automatically using a software tool, and thereby saving manual effort.

To create the model of the system, a variety of model types which may be used, which we shall briefly describe. *Automata* (Finite State Machines) consist of a set of states, a starting state, a set of final states and a set of transitions. Each event triggers a transition to another state [13]. Finite state machines are form the basis of models in WATERS.

Timed Automata are automata which simulate time using real-valued clocks [61]. Transitions are enabled based on the value of these clocks. *UPPAAL* a modelling tool that uses timed automata [70].

I/O Automata are an extension of automata which distinguish between *input*, *output* and *internal* events. I/O Automata have often been used to model reactive systems [61].

Team Automata are an extension of I/O automata which are used to model collaboration in groupware systems [61]. Examples of team automata are presented by ter Beek et al. [62].

Petri Nets (PNs) are directed graphs that contain *places* and *transitions.* A transition is only enabled if the required place has sufficient tokens. Petri Nets are especially useful for modelling concurrent processes [22]. A Finite State Machine can in fact be defined as a Petri Net in which the number of inputs and outputs per transition is exactly one [25].

Timed Petri Nets (TPN) are an extension to Petri Nets which model time by assigning a time duration to each transition [25, 51].

Kripke structures are essentially finite state machines in which each state contains a set of propositions which are true in that state [18]. Kripke structures are especially useful for modelling digital electronic circuits.

Other modelling techniques used by formal methods include regular algebra, Markov models, Z notation, and ADL [70]. This thesis focuses on model checking using finite state machines.

Clarke Jr et al. discuss a practical example of the use of model checking in the design of the *Futurebus+* cache coherence protocol [18]. During the formalising and verification of the protocol, a number of errors and ambiguities

were discovered using model checking.

2.4 Discrete Event Systems

A *Discrete Event System* (DES) is a representation of a real-world system containing states, events and transitions [13]. The system starts at an *initial state*. Events cause the system to transition to another state[7]. Time is considered to be discrete, such that each events occurs at a discrete points in time. A DES may be represented in a number of ways (such as the model types listed in Section 2.3. Finite state machines (automata) are one of the simplest types of model for Discrete Event Systems [13].

In particular, a DES can be described by the combination, or composition, of several finite state machines. To this end, let us define a deterministic finite state machine as the set $(Q, q_i, Q_m, \sum, \delta)$. Q is the set of all states and q_i is the initial (starting) state. Q_m is the set or marked states; this is the set of states which the automaton should eventually be able to reach. $\sum$ is the set of all possible events, called the *event alphabet*. δ is the transition function; given a current state and an event, this defines the next state.

A DES does not model details about events, but only that they occurred (for example, the fact that a user request was received, but not the data within the request).

The automata in a DES perform *synchronisation on common events* [27, 13]. When an event fires, all automata which contain the event transition *at the same time* (they are synchronised). In addition, if one of the automata which contain the event is not in a state where it can be executed, the event cannot fire at all.

A DES typically includes one or more *specifications* that define the intended behaviour or controlling logic of the system. Specifications control the flow of the DES by specifying which events are allowed in any given state [22].

[7]It may transition to the same state

Events can be classified as controllable or uncontrollable [13]. *Controllable events* may be enabled or disabled by the specification (the controller), whereas uncontrollable events cannot.

Events are also classified as either *observable* or *unobservable* [13]. Observable events may be responded to directly by specifications, whereas unobservable events cannot.

A specification is *controllable* if it defines transitions for all possible uncontrollable events which may occur [27, 13]. This means that no uncontrollable event will put the DES in a state where it cannot be controlled. We can define this formally. The following definition is taken from Åkesson et al. [3]: Let G and K (the specification) be two automata with the same event alphabet Σ. K is controllable with respect to G if $\mathcal{L}(G \parallel K)\Sigma_u \cap \mathcal{L}(G) \subseteq \mathcal{L}(G \parallel K)$, where Σ_u is the set of uncontrollable events.

A *marked stated* or *accepting state* is a state which indicates that the system has finished its processing and is in a normal, idle state. A DES should ideally always be able to end in an accepting state; then it is called *nonblocking*. The following definition is taken from Åkesson et al. [3]: Let G be an automaton. G is nonblocking if $\mathcal{L}(G) \subseteq \mathcal{M}(G)$, where $\mathcal{M}(G)$ is the marked language of G (the set of strings which end in a marked state).

Alternatively, one can define a DES as *nonblocking* if it has no *deadlocks* or *livelocks* [27, 13]. In a *deadlock*, the system has reached a state that it cannot transition out of. In a *livelock*, the system becomes stuck in a loop of unmarked states that it cannot transition out of. It is still "live" in the sense that it is transitions between states, but it can never reach an accepting state.

2.5 WATERS

WATERS[8] (the Waikato Analysis Toolkit for Events in Reactive Systems) is the model checking used in this work. In WATERS, models are created as

[8] `http://www.supremica.org/`

Discrete Event Systems (DES) using *Extended Finite State Machines* (EFSM). Extended Finite State Machines, or Extended Finite Automata are an extension of ordinary automata, which include *variables*, *guards* and *actions* [56]. WATERS is formally called *WATERS/Supremica*, as it is based on the *Supremica* program.

There are four types of automata in WATERS: *plants* represent the system to be controlled; *specifications* represent the control logic for the plants; *properties*; and *supervisors* are used for synthesis (generating models automatically). Models can also contain variables. Transitions in WATERS can contain *guards* that only allow the transition if the guard condition is true, and *actions* that change the value of a variable [42].

WATERS is a comprehensive model-checking software package; it provides verification tools including controllability check, conflict check, deadlock check, control-loop check and property check.

- Controllability check: This checks that the specification is controllable.
- Conflict check: This checks that the system eventually reaches an accepting state.
- Deadlock check: This checks whether the system may enter a state from which it cannot escape (a deadlock).
- Control loop check: This checks whether the machine could enter a loop of controllable events and thus be prevented from reaching an accepting state (a livelock).
- Property check: This ensures the *properties* (modelled as automata) are satisfied by the language specified by the DES.

WATERS allows the definition of *variables* that can be used within transitions. This is a feature of extended finite state machines. A variable in WATERS has a type (such as numeric), a set of allowable values and a starting value.

Events in WATERS may have *guards* and *actions*. If an event has a guard, the event is only allowed to fire if the guard condition evaluates to true. An action usually modifies the value of a variable when the transition is fired. Actions should only be used within *plant* automata. In addition, if the action for an event assigns to a variable a value which is outside its allowable range, then the cannot be fired (it is blocked). In this way the action also acts as a guard.

Note that in WATERS one cannot modify the state machine itself by adding states or transitions on-the-fly. So it is not possible, for example, to model scaling out by having a transition which creates another state machine entirely.

To perform model checking, WATERS models are first compiled then verified. In general, complexity of verification in WATERS is polynomial in the number of states and transitions in the *composed* system, and exponential in the number of components (automata and variables). Different parameters can be passed to WATERS to improve its performance (for example, compiling the model to a binary decision diagram representation using the "-bdd" flag).

Other examples of model checking tools include *CZT*, *NuSMV*, *PAT*, *PRISM*, *SPIN*, and *UPPAAL* [18, 57, 70]. However, we will focus only on WATERS in this thesis.

2.6 Cloud Management Software

Cloud systems are usually managed by *Cloud Management Software*, such as Kubernetes or Google Cloud. This software manage the deployment and adaptation of the cloud, including starting and removing VMs or containers, and overseeing the health of the system. This may also be referred to as *cloud orchestration* [72].

Kubernetes (also called K8s) is a container-based cloud orchestration program developed by Google [8]. In Kubernetes, the containers are called *pods*[9],

[9]https://kubernetes.io/docs/concepts/workloads/pods/pod/

and are hosted on machines called *worker nodes*. In particular, K8s can increase or decrease the number of containers in the cloud based on certain metrics, via the Horizontal Pod Autoscaler [1]. Kubernetes is the cloud management software focused on in this thesis.

Kubernetes actually has three systems for automatic scaling: the *Horizontal Pod Autoscaler*, the *Vertical Pod Autoscaler* (VPA) and the *Cluster Autoscaler* [1]. The Horizontal Pod Autoscaler creates more instances of pods. The Vertical Pod Autoscaler assigns more resources to existing pods. The Cluster Autoscaler creates further worker nodes to contain pods. The HPA is the focus of this thesis.

Other examples of cloud management software include *CloudFormation*, *OpenStack Heat*, *Puppet*, *Chef* and *Ansible* [72].

Chapter 3

Related Work

This chapter presents related work in three categories: existing frameworks, overview papers, and investigative papers. Existing frameworks check cloud system validity at design-time or increase cloud performance at runtime; some do both. These are discussed and compared with this thesis when appropriate. Overview papers give a summary of existing research on a topic relevant to this thesis. Investigative papers describe aspects which are useful to model or at least take into account for this thesis.

3.1 Existing Frameworks

This section discusses the existing frameworks which aim to verify or improve cloud systems. These are summarised in Table 3.1. To the best of our knowledge, there is limited work on the formal modelling and verification of cloud autoscaling policies. In particular, to our knowledge no research exists that applies model checking to verify cloud horizontal autoscaling using non-probabilistic finite state machines.

The columns are used as follows: the target system is the system which the research applies to. Most apply to cloud systems specifically, but some are applicable to self-adaptive systems in general [9, 39]. Those with a checkmark in the *Optimisation* column focus on improving the performance or overall quality of the system at runtime, without necessarily providing formal guarantees.

Table 3.1: An overview of relevant frameworks towards providing guarantees for or improving cloud and self-adaptive systems.

Source	**Target System**	**Optimisation**	**Verification**	**Design Time**	**Runtime**	**Performance**	**Security**	**Model-Checking Tool Used**
[4]	Cloud			✓			✓	Z3
[9]	Self-Adaptive Systems	✓			✓	✓		PRISM
[14]	Web Services		✓	✓				SPIN
[20]	Cloud	✓	✓	✓	✓			
[21]	Cloud	✓	✓	✓		✓		PRISM
[23]	Cloud		✓	✓			✓	
[24]	Self-Adaptive Systems		✓	✓	✓	✓		PRISM
[25]	Cloud		✓	✓		✓		
[27]	Phone Application		✓	✓				WATERS
[32]	Cloud		✓	✓		✓		
[34]	Cloud		✓		✓	✓		PRISM
[33]	Cloud		✓	✓	✓	✓		Z3
[35]	Cloud		✓	✓				
[36]	Cloud	✓			✓	✓		
[39]	Self-Adaptive Systems		✓	✓	✓	✓		
[43]	Cloud		✓	✓		✓	✓	Z3
[45]	Cloud		✓	✓	✓	✓		
[46]	Cloud	✓			✓	✓		
[48]	Cloud	✓			✓	✓		
[50]	Cloud	✓			✓	✓		
[52]	Web Services	✓	✓		✓	✓		
[53]	Cloud	✓			✓	✓		
[55]	RAS		✓	✓		✓		GreatSPN
[67]	Cloud (Case Study)		✓	✓			✓	FOAM
[72]	Cloud		✓	✓				
[73]	Cloud		✓	✓			✓	Maria
This Thesis	**Cloud**		✓	✓		✓		**WATERS**

Those with a checkmark in the *Verification* column aim to verify if the system will meet its SLA (or in the case of runtime, is meeting its SLA currently). The *Design Time* and *Run Time* columns indicate whether the framework is applied at design time or runtime (or both). The *Performance* and *Security* columns indicate whether the frameworks focus on performance (availability, response time and throughput) or security (the CIA triad)[1]. Finally, the Model Checking Tool used is listed, where applicable.

Evangelidis et al. propose a probabilistic performance model using Discrete Time Markov Chains, focusing on cloud horizontal autoscaling policies [21]. This work used the PRISM model checker, and related CPU utilization with response time using K-Means clustering. Similar to our work, this falls into Wave V (Guarantees under Uncertainty) and in Wave III (Performance Models) of self-adaptation [69]. Instead, we use non-probabilistic modelling, and focus on modelling the mechanics of autoscaling (a bottom-up, "white box" approach).

Heidari et al. present a method for controller synthesis to determine a cloud system configuration that meets an SLA in [25]. In this study the cloud system is modelled using Timed Petri Nets (TPN) and could be extended to other model types. The authors provide an example for maintaining availability in the case of component failure. In contrast, we focus on maintaining availability based on horizontal autoscaling.

Raimondi et al. collected performance information at runtime to detect a state that indicated an SLO was about to be violated [52]. In contrast, we are aiming to predict SLO violations at design time.

Johnson et al. introduce the INVEST framework for efficient incremental verification of probabilistic models, integrating with the PRISM model checker [34]. The authors present a model for the availability/reliability of a multi-tier cloud service given certain probabilities of component failure. Similar to our work, the test case examines cloud availability. However, their ex-

[1]This is a broad distinction and may not apply neatly to all studies.

ample is based on probability of component failure and not autoscaling. They also use probabilistic modelling, whereas we use deterministic modelling.

A closely related work presents a method for incremental verification of adaptive systems using a satisfiability modulo theory (SMT) solver [33]. Again, the system is re-verified after undergoing change at runtime, as required. The authors suggest how the two methods could be integrated.

Idziorek presents a discrete event simulation model to investigate horizontal scaling within a cloud system [31]. This is, however, based on VM-based clouds and on simulation, whereas we focus on container-based clouds and formal verification.

Basset et al. describe a method for composition of stochastic games using *probabilistic automata* (PA) to improve the performance of autonomous systems [9]. In contrast, this thesis uses regular (non-probabilistic) automata.

The Kubernetes community is planning to implement SLO guarantees as part of the *SIG-Scalability* project [37]. The documentation distinguishes between *steady state* SLOs (during normal traffic) and *burst* SLOs (during unusually high traffic).

Yoshida et al. used the *TOSCA* (Topology and Orchestration Specification for Cloud Applications) language to model relationships between components of a system. TOSCA was used to specify the service template for a cloud application, which consisted of a topology template (resource structure) and set of plans. TOSCA designs were mapped to state machines and prove that the systems have “leads-to” properties (a class of liveness). This is a useful approach for testing availability [72].

Hinze et al. used WATERS to verify and improve the design of a mobile tourist information system [27]. While this is different from a cloud system, we also hope to improve the design of cloud systems with the same approach.

Zeng et al. used Coloured Petri Nets (CPNs) to verify security requirements for cloud systems [73]. Their system ensures that resources can only be accessed by clients with the appropriate permission level. The focus of the

research is security, whereas this thesis focuses on availability.

A framework to provide cross tenant access control (CTAC) is presented by Alam et al. [4]. The cloud resource mediation service (CRMS) runs as a third-party service and manages access to resources from one tenant to another. The system is modelled using High Level Petri Nets, and verified using SMT-Lib and the Z3 solver. The focus of the research is security, whereas this thesis focuses on availability.

Chareonsuk and Vatanawood proposed a method to formally validate orchestration of services in the cloud [14]. In this approach the topology and service functionality of the cloud are specified using the TOSCA and BPEL XML-based languages. This specification is then translated into Promela code, and the resulting Promela code is verified by the SPIN model checking tool. Safety properties are specified using linear temporal logic. In contrast, this thesis does not focus on composition of services (primarily an SaaS concern), but rather on the level of PaaS and FaaS.

Vinárek et al. present the FOAM tool for lightweight formal analysis of use cases [67]. They present an example in the context of cloud service providers, but the tool can be used for other domains as well. The use cases are annotated with required conditions (such as the fact that a resource must always be closed after being opened). Then the NuSMV model checker is used to verify that the proposed implementation of the use cases will meet these conditions. This focuses on functional requirements, whereas in this thesis we focus instead on non-functional requirements.

Malik et al. used a combination of model-checking and theorem-proving using High Level Petri Nets (HLPN) to verify cloud management software itself [43]. The focus is on IaaS, particularly the configuration of VMs within the cloud environment. However, the authors do not present an example of a violation; in all the tests given the management software works as expected. The *Z3* solver is used as the verification tool.

Etchevers et al. present the *VAMP* framework for formally specifying rela-

tionships between cloud application components, and performing the deployment of a VM-based cloud system [20]. The formal modelling is done using an extension to the XML-based specification language called OVF (*Open Vitalisation Format*). An evaluation is presented, in which the focus was on the speed of deployment. In contrast, this thesis focuses on the *post-deployment* behaviour of the cloud.

Klai and Ochi present a method to verify the composition of cloud-based services using *Symbolic Observation Graphs* (SOGs), *Linear Temporal Logic* (LTL), Petri Nets and *Labelled Kripke Structures* (LKS) [35]. The aim is to check whether a set of cloud services will function as expected when composed together. The authors present a class of Petri Net called a *resource-constrained open workflow* net (RCoWF-net), used in the verification process. In contrast, this thesis focuses on modelling the cloud system itself as opposed to the composition of services running on the cloud.

Lee et al. present the *RINGA* framework which uses model-checking in self-adaptive software at runtime [39]. The authors present a type of state machine created at design time called a self-adaptive state machine (SA-FSM). This is used to create an adaptive finite state machine (A-FSM) which is then used in a MAPE loop at runtime to make the software self-adaptive. If the trigger conditions are met, an adaptive transition is performed. The RINGA framework was tested using an IoT-based lighting controller, and its performance overhead compared to existing symbolic model checking tools. This is somewhat related to the work of Klein et al. [36] which aimed to help create self-adaptive software, and Johnson et al. [34] which used model-checking at runtime. However, it is perhaps too low-level for this thesis, which focuses on modelling high-level behaviour of a cloud system at design-time only.

The SLAC Management Framework presented by Uriarte et al. verifies that an SLA is internally consistent at design time [64]. It also monitors the cloud system at runtime to check if the SLA is being met, and sends

alerts if there violations. However, this does not seem to check if a cloud *design* meets the SLA. It also does not take into account actions such as scaling. The authors also focus on IaaS clouds, whereas this thesis focuses on PaaS. In addition, SLAC is a textual language written in terms of constraint satisfaction problems, whereas this thesis focuses on model checking using finite state machines.

Rodríguez and Campos used Petri Nets to model the throughput (performance) of systems [55]. Specifically, the authors present a technique to estimate the throughput of resource-allocation systems (RASs). A cloud system may be viewed, at least partly, as such a system.

Ficco et al. present an extension to UML to allow modelling security in the context of cloud systems [23]. The cloud is represented by a *Cloud Component Diagram* and *Deployment Diagram*, which include *Use Cases*, representing normal and intended user behaviour, and *Misuse Cases*, which pose a security threat. *Mitigation Cases* are added to specify how Misuse Cases are handled. A useful overview of security modelling frameworks for cloud systems is also provided, notably *Abstract State Machine Language*, which is based on extended finite state machines, and STATL, a state/transition-based language. However, the focus of the work is security, whereas this thesis focuses on availability.

Nawaz et al. present a framework to predict possible SLA violations in a dynamic Cloud of Things (CoT) environment [45]. The authors present a framework using probabilistic modelling to predict SLA violations based on events. This requires translating the SLA into a set of rules, and analysing past QoS data. This is used to infer the likelihood of SLA violations given a set of events. The authors also compare the existing event-driven approaches to modelling SLA violations.

3.2 Overview Papers

This section presents papers which provide a summary of relevant topics to this thesis. These are listed also in Table 3.2.

Table 3.2: Overview papers relevant to this thesis.

Citation	**Topic**
[6]	Modelling QoS in cloud systems
[61]	Web service composition approaches
[41]	Assuring QoS in reactive systems
[57]	Formal verification in cloud systems
[70]	Use of formal methods in self-adaptive systems
[71]	How to model requirements for self-adaptive systems

Souri et al. present a survey of the recent use of formal methods to verify cloud systems [57]. Notably, the authors have not listed WATERS as a tool used for checking cloud systems. Existing formal approaches are grouped into three categories: *specification process algebra*, *model or property checking* and *theorem proving* [57]. Model checking is further divided into two categories: *state-based* and *action-based*. The state-based approaches use a Kripke Structure, whereas the action-based approaches use a labeled transition system (finite state machine). Making use of this work, this thesis would be added in the category *Model Checking - Action Based - Labeled Transition System.*

Weyns et al. survey the use of formal methods in self-adaptive systems including the *Internet of Things* (IoT) and cloud systems [70]. The authors state that the main focus of these efforts has been performance and reliability, and also note that there is a need for lightweight tools which use formal methods to verify system performance at runtime.

Ardagna et al. provide an overview of Quality of Service (QoS) modelling for cloud systems [6], covering modelling techniques such as *wavelet-based methods*, *regression analysis*, *filtering*, *Fourier analysis* and *kernel based meth-*

ods. The focus is on availability, which is also the focus of this thesis.

Mahdavi-Hezavehi et al. review existing approaches to handling multiple Quality Attributes (QAs) in self-adaptive systems [41]. Performance, reliability, cost, availability and scalability were the most commonly studied QAs. Only one of the studies investigated used automata to model QAs and their characteristics. Model checking was used by 5 of the studies surveyed.

ter Beek et al. discuss the application of formal methods to the composition of Web Services [61]. Web services typically provide a description of their functionality, and formal methods may be applied to verify if the composition of different web services will produce the required behaviour. In particular, the authors mention automata - which is the focus of our work.

Yang et al. provide an overview of requirements modelling for adaptive systems (including cloud systems) [71]. The approaches modelling tools used included KAOS, pi*, and so on. The authors do not seem to list a Discrete Event Simulation tool such as WATERS. In Figure 12 the authors provide a table of Requirements Engineering activities. Our project in in the category *Modelling requirements.*

3.3 Investigative Papers

This section presents papers investigate an aspect of cloud computing which may be useful to model formally. These are summarised in 3.3.

Table 3.3: Investigative papers relevant to this thesis.

Source	**Topic**
[26]	Elasticity in Cloud Computing
[46]	Scaling in the presence of resource-intensive tenants
[49]	Effect of garbage collection in Java on SLOs
[54]	Sharing resources amongst cloud providers
[74]	Scaling of Node.js applications in the cloud

Patros et al. investigated the effects of scaling an application in a multi-tenant cloud where other tenants consume a maximum amount of resources [46]. The authors introduce a set of applications called *Cloud Burners*, which deliberately consume a maximum number of allocated resources. There is one for CPU, Cache, Resident Memory, Disk I/O and Network I/O. The results indicate that scaling does not always work as expected. For example, scaling horizontally may cause further network congestion, making the application perform worse when the Network I/O is the bottleneck.

Patros et al. reported the effects of different garbage collection policies on meeting SLOs in a cloud environment [49, 47]. The authors created a Java test program called *CloudGC* for testing garbage collection settings versus SLOs in the cloud. Using *CloudGC* the authors tested various parameters and policies and reported the relationships between them.

Patros et al. introduced a method to re-order requests in the cloud to better meet SLOs [48]. This favours the scenario where there are multiple connected clients (many users or devices communicating with the same cloud).

Rameshan et al. [53] present the *Stay-Away* framework which helps ensure SLA compliance for multi-tenant clouds. If a batch application (that is, not performance-sensitive) is co-located with a performance-sensitive application, *Stay-Away* throttles the batch application as required to ensure the performance-sensitive application has enough resources to meet its SLOs.

Hu et al. present a framework to provide formal verification as a service in a cloud system [28]. This is designed to take advantage of cloud features such as scaling, pay-as-you-go, and to be used my multiple clients (multi-tenant). Using this, models are made as bigraphs using a graphical interface. The models are then converted to SPIN code and verified using the SPIN model checker, since the tools for checking bigraphs directly were not mature yet. To be clear, the authors present a general-purpose verification service which runs *on* the cloud; they are not verifying a cloud system itself. This thesis investigates modelling the cloud system itself.

Jindal et al. modelled the performance of micro-services based cloud applications by testing the individual micro-services to determine their capacity in terms of requests per second, before SLOs are violated [32]. This was done using the *Terminus* tool.

Podolskiy et al. used machine-learning algorithms to find the best configurations for horizontal and vertical scaling to ensure SLOs are met and cost is kept to a minimum [50].

Klein et al. explain the *brownout* concept [36]. Similar to an electrical brownout, this refers to reducing the amount of work the software does, in order to maintain availability during peak periods.

Souri et al. present the *Graphical Symbolic Modelling Toolkit* (GSMT) model-checking framework for distributed systems. This allows creating a model via a graphical user interface as either labelled transition systems or Kripke structures. The tool then generates the SMV code for the model, and the verification is done using the NuSMV tool. The authors also present a useful summary of existing tools for the same purpose. However, WATERS is not listed in the summary.

Chapter 4

Theoretical Model

This chapter presents our theoretical model for a cloud system. This is a basis for the formal guarantees in the WATERS models presented in Section 5. As far as possible, we have based this model on Kubernetes. We focus on Platform as a Service (PaaS) and Function as a Service (FaaS) clouds, since providing guarantees under uncertainty in Infrastructure as a Service (IaaS) and Software as a Service (SaaS) has been studied in existing work, for example by Wang et al. [68] and Chen et al. [15].

If the theoretical model is accurate enough, then the WATERS verification results provide formal guarantees of SLA compliance (or non-compliance). It should be noted that the theoretical model is currently rather limited and acts as a proof-of-concept of our proposed methodology; the models can be expanded in the future.

The theoretical model for a cloud comprises the *workload*, *cloud* (pods), *SLA* and the *master*. Note that we work at the level of pods, not worker pods. The cloud users submit a number of requests each second. These requests are delegated to the pods of the cloud. The pods process the requests, which takes a certain duration per request. The master scales out or in by creating or removing a pod when the queue lengths reach the relevant thresholds. The SLA specifies the maximum response time for processing requests. Each of these components is explained in detail below.

4.1 Workload

Let us represent the *workload* as a set of stateless requests from users to the cloud. Each request asks the cloud system to perform a task (for example, return a webpage) which consumes various amounts of cloud resources, such as RAM, CPU or disk space. Let us assume the SLA specifies a maximum rate at which the requests may be sent by users per second, namely RPS_{max}.

Let us assume that each request takes the maximum possible processing time PT_{max}; this way the worst case scenario is modelled. In a real-world cloud, each request may take a different amount of time to process [48] (for example, a request to encrypt a file may take longer than a request to return a static webpage).

Let us consider two shapes of workload, a *constant load* and a *square wave load*. In a constant load, there is a fixed maximum rate at which requests are sent, RPS_{max}. Each second, the number of requests entering the cloud is a number between zero and this maximum. In a square wave load, this maximum rate is first set to a high level $RPS_{\text{max(high)}}$ for a set amount of time T_{high}, then a low level $RPS_{max(low)}$ for a set amount of time T_{low}; this pattern is then repeated indefinitely.

Note that we do not model the network between the user's machine and the cloud, since we are considering the cloud provider's perspective. Therefore packets that are dropped by the network between the cloud and client are not modelled.

4.2 Cloud

Let us define the *cloud* as a set of *pods*. A pod represents a single Kubernetes container within the cloud. Let as assume this is a single-tenant cloud, and that each pod runs one instance of a single (stateless) application.

When a request is submitted to the cloud, it is delegated to exactly one of the pods, and is appended to the *queue* of the designated pod. Every second,

the pod processes a certain number of requests and removes them from its queue. Let us assume that the queue length has no limit.

Since each request takes $PT_{\max}$ (Processing Time maximum) to process, the total number of requests processed by a single pod per second is $1/PT_{\max}$. Again, we assume the worst case scenario, that the processing time is the maximum possible value.

Let us assume that the cloud will always have at least one pod running ($pods_{\min} = 1$) and there is a maximum limit of pods $pods_{\max}$. Let us also assume that requests are allocated to pods via a round-robin scheme. That is, the first request is allocated to the first pod, the second to the next pod, and so on. Formally, if the number of pods currently on is $pods_{\text{current}}$, the i-th request is assigned to pod number $((i-1) \mod pods_{\text{current}}) + 1$.

4.3 SLA

Let us define an SLA as a single SLO: a maximum response time of $RT_{\max}$. Thus, all requests must be satisfied within $RT_{\max}$ seconds.

Let us determine the maximum possible queue length in order to satisfy the SLO. We assume that the requests are always served in a first-come-first-server order. This means that if there are too many existing requests in the queue (for any pod), the new request will not be serviced in time and the maximum response time SLO will be violated. This occurs when:

$$QL \times PT_{\max} > RT_{\max}$$

where QL is queue length of any pod, including the new request to be checked. This can be seen as a variation of *Little's Law* [40].

So the SLO is satisfied if and only if, for each pod, $QL \times PT_{\max} \leq RT_{\max}$. This can be rearranged as:

$$QL \leq \frac{RT_{\max}}{PT_{\max}} \tag{4.1}$$

Using this we can calculate the maximum queue length to still meet the

SLO:

$$QL_{\text{max}} = \frac{RT_{\text{max}}}{PT_{\text{max}}} \tag{4.2}$$

Therefore in order to meet the SLO, the queue length of any pod must not be greater than QL_{max} as defined above.

4.4 Master

The master represents the controlling logic and self-adaptive strategy used by the cloud. For this theoretical model, let us consider only horizontal autoscaling. Let us assume that the master scales out by one pod when the queue of the last pod reaches a threshold $QL_{\text{scale-up}}$. Let us define the threshold in terms of percentage of the maximum queue length QL_{max} as defined in (4.2). Let us introduce a scale threshold parameter ST_{up} between 0 to 1, such that:

$$QL_{\text{scale-up}} = QL_{\text{max}} \times ST_{\text{up}} \tag{4.3}$$

For example if ST_{up} is 0.80, the master will scale out when the queue length of the last pod is at 80% or more of its capacity. Let us also assume that the master will scale down (in) by one pod when the queue of the first pod reaches a threshold $QL_{\text{scale-down}}$. Let us introduce a parameter ST_{down} from 0 to 1.

$$QL_{\text{scale-down}} = QL_{\text{max}} \times ST_{\text{down}} \tag{4.4}$$

For example if ST_{down} is 0.20, the master will scale out when the queue length of the last pod is at 20% or less of its capacity.

Let us also assume that pods take a certain amount of time to start up and start processing requests after being created. This time in seconds is $T_{\text{pod-startup}}$.

Let us assume that the master does the check for scaling only at set time intervals. Let us introduce a parameter $T_{\text{scale-check}}$ to indicate that this check occurs every $T_{\text{scale-check}}$ seconds. We also assume that $T_{\text{scale-check}} > T_{\text{pod-startup}}$,

to avoid the case where the master starts a pod while one is already starting up.

This model is based on the Kubernetes *Horizontal Pod Autoscaler* (HPA) [1], but there are some notable differences. Firstly, the Kubernetes HPA scales based on CPU usage, and not queue length. We assume, however, that CPU usage is directly proportional to queue length. The relationship between queue length and CPU utilisation could perhaps be determined experimentally, but due to time constraints this was not done—related work has shown that expected response times based on CPU utilization can be predicted using Machine Learning [50]. Secondly, Kubernetes also does not scale in or out by exactly one pod each time, but rather sets a desired number of pods, and creates or removes pods to meet this desired number[1]. Scaling by one pod at a time was the simplest to model, and is reasonable for a small number of maximum pods as we have in our experiments (the maximum is 4). However, for a large number of maximum pods (such as 100), this assumption would cause the model to be very inaccurate, since the cloud might scale by a large number of pods at a time. Finally, Kubernetes does not consider CPU metrics from pods that have been recently created, as those metrics may not be available or stable yet. This is controlled by the "–horizontal-pod-autoscaler-initial-readiness-delay" parameter, which has a default value of 30 seconds. This could be addressed by having the model ignore the queue length of pods that have not been active for a certain amount of time; however, this does add complexity to the model, and the calculation is already complicated due to the CPU vs. length-difference issue explained earlier. The next chapter describes how this theoretical model is represented in WATERS.

[1]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details

Chapter 5

System Model

This chapter presents our *system model.* These are extended finite state machines created using WATERS, based on the theoretical model presented in the previous chapter. This chapter presents the system model for the master, workload, cloud, SLA, and Horizontal Pod Autoscaler.

The naming convention used within the models is that states, automata and transitions are spelled in lowercase, for example *user_submits_requests*; variables and named constants are capitalised, for example Submitted.

The requirements for our final models were that they had to compile and perform a property check (for all properties) within 10 seconds such that the model is convenient to use. We also aimed to satisfy the generally desirable properties for a DES: the models should be controllable, nonblocking, and contain no livelocks, deadlocks or control loops. These properties were confirmed by checking the final models using the WATERS controllability check, conflict check, deadlock check and control loop check features.

The model consists of automata that are synchronised on common events. Thus, if two or more automata use the same event, the event can only fire if it is enabled in all automata, and it fires in each automaton at the same time (in lock-step). For example, because the event *user_submits_requests* is present in both the *master* and user automata, it can only fire if *user* is in *idle* and *master* is in *waiting* (discussed in Sections 5.1 and 5.2).

We developed two separate, but similar models in WATERS. The first represents a cloud system running a default NGINX installation. The second represents a cloud running a custom Node.js application. The NGINX cloud has a fast and variable response time, and was tested using a constant maximum load (RPS_max). The Node.js application has a fixed and relatively slow response time, and was tested using a square wave load. Both WATERS models can be downloaded from our GitHub repository[1]. We will firstly explain the NGINX model fully, then explain differences for the Node.js model in Section 5.6.

5.1 Master

The *master* automaton controls the overall flow of the model. This is shown in Figure 5.1. The automaton starts in the state *waiting* and transitions to the state *waiting_for_user_to_finish* via event *user_submits_requests*. This event is synchronised in the automaton *user*, which is described in Section 5.2. Once the user automaton has finished, it triggers the event *user_finished*. This lets the master automaton transition to state *user_done*. The master then triggers the event *scaling_start* and goes to state *waiting_for_scaler_to_finish*. The scaling is then done by the automaton *horizontal_pod_autoscaler*, which is described in Section 5.4. The *horizontal_pod_autoscaler* automaton triggers the event *scaling_finished* when done, which lets the master transition to state *scaling_done*. The master then triggers the transition *pods_handle_requests*. This is synchronised to the *pod* automata, which handle requests and remove them from their queue. This is described in Section 5.3. Finally, the master transitions back to state *waiting* via the event *one_second*. This represents one second of time. The loop can then continue indefinitely.

[1] `https://github.com/martinvanzijl/masters-project/`

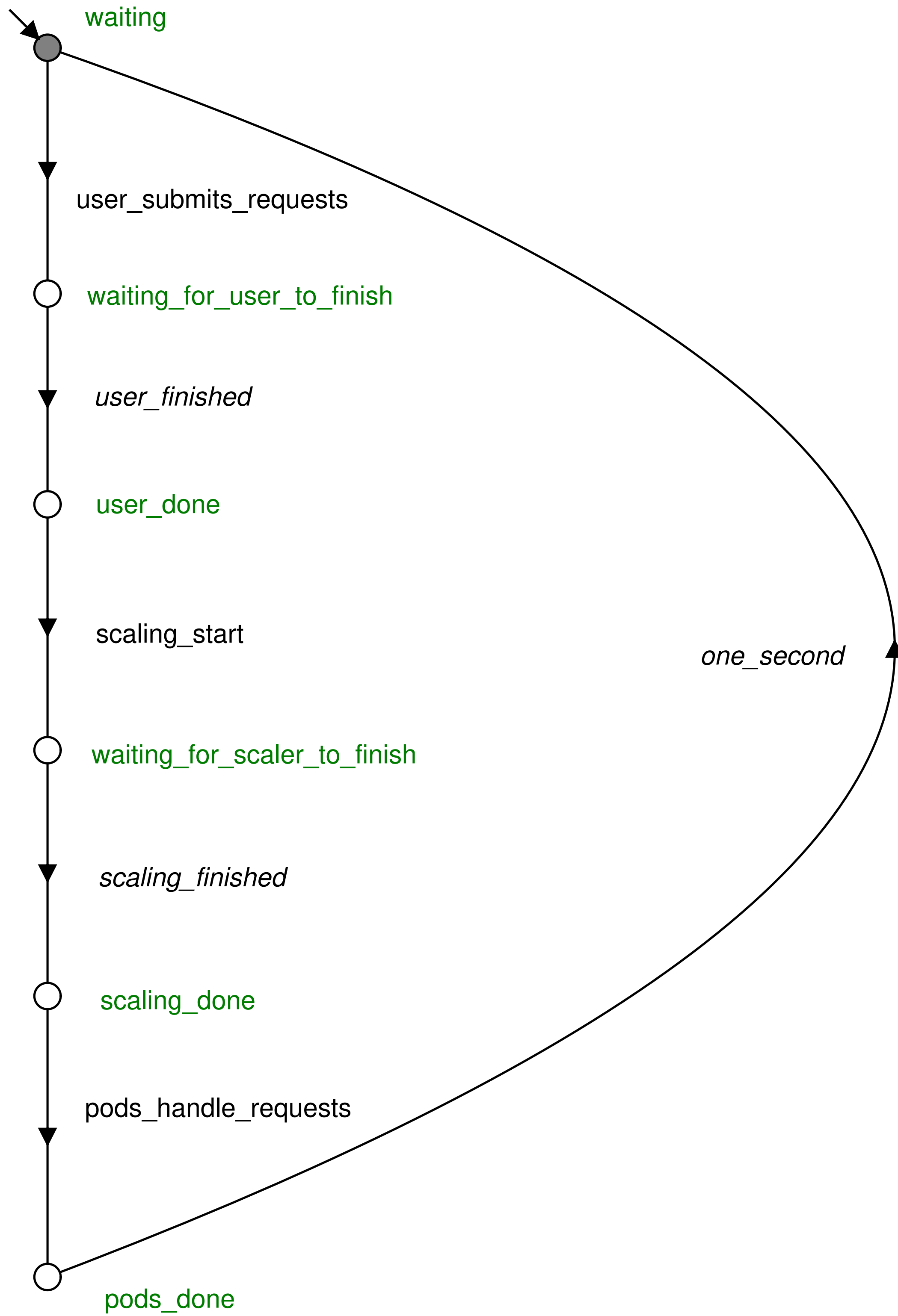

Figure 5.1: The *master* specification.

5.2 Workload

The workload is controlled by the plant called *user*. This plant represents the users of the cloud as a whole. Every second the users send a certain number of requests to the cloud. The *user* automaton for this design is shown in Figure 5.2. The corresponding *pod_queue* automaton for this is shown in Figure 5.3. Note the guard condition for the *user_submits_request* event in the user automaton: $\mathsf{Submitted}' \leq \mathsf{RPS_Max}$. The prime notation indicates the next state of the variable. Therefore this guard ensures that the transition is allowed only if the next value of Submitted (that is, Submitted′) is less than RPS_Max. Therefore when the transition is finished and the automaton is in state *submitting*, the value of Submitted is between zero and RPS_Max. Note that RPS_Max is set to the maximum number of requests per second allowed.

This highlights one of the key uncertainties that model checking allows us to explore: the incoming traffic to the cloud. Each second there may be any number of requests up to RPS_Max. For a period of 2 minutes and a maximum of 299 requests per second, the number of possible patterns is 300^{120} (300 possible values each second, for 120 seconds): it is infeasible to test all possible variations with a load testing tool; however, model checking can test this rapidly.

The number of pods currently active (on) is represented by the variable Pods_On, which is always between Pod_Min and Pod_Max. There is one *pod_queue* automaton for each pod, identified by Pod_Index from 1 to Pods_On.

The *pod_queue* automaton (Figure 5.3) represents the queue of a pod with index Pod_Index. Since we assume round-robin load-balancing, we divide the submitted requests evenly among the pods that are currently active. On the event *allocate_requests_to_pods*, there are two possible transitions: if the pod is on (that is, $\mathsf{Pods_On} \geq \mathsf{Pod_Index}$) then the action is executed: $\mathsf{QL[Pod_Index]} = \min(\mathsf{QL_Max}+1, \mathsf{QL[Pod_Index]} + \mathsf{Submitted} / \mathsf{Pods_On})$. This means the queue length of the pod will be the minimum of: 1) QL_Max+1, which is the length at which the SLO is violated, and 2) QL[Pod_Index] + Submitted /

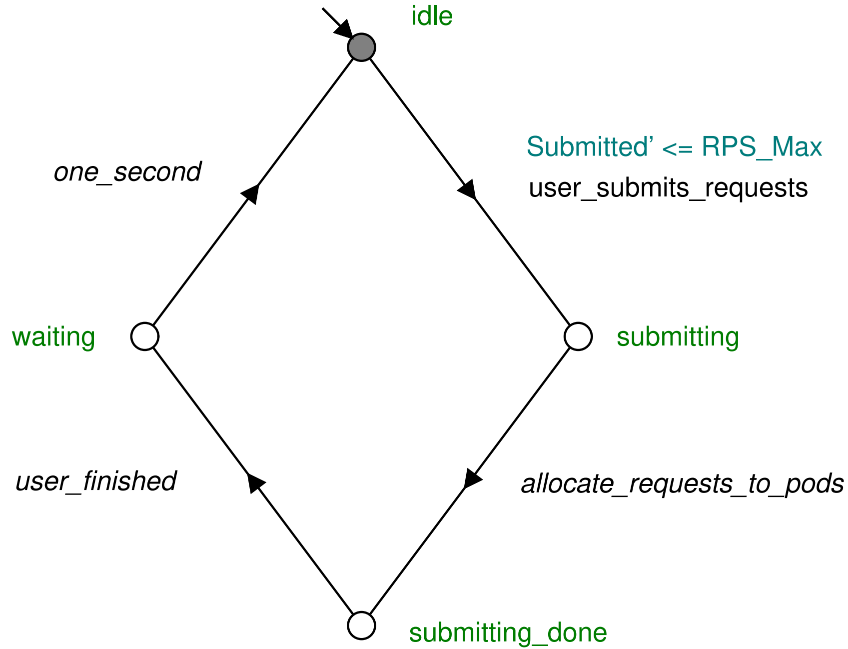

Figure 5.2: The *user* automaton. Submitted' is the next state of Submitted. The Submitted variable represents the number of requests submitted in this second. RPS_Max is the maximum number of requests per second.

Pods_On, which is the current length with the number of requests submitted this second divided by the number of available pods added. The reason for using the minimum is to avoid the model being blocking on this transition. QL_Max+1 is the maximum value of the QL[Pod_Index] variable, and the transition will be blocked if the assigned value is over this maximum. There is no need to consider any higher values, since a property check will fail for any value higher than QL_Max. If the pod is off (that is, Pods_On < Pod_Index), then the pod queue length stays the same.

One downside of this approach is that, if the amount of requests submitted does not divide evenly into the number of pods currently on (for example, 5 requests were submitted and 2 pods were on), then the remainder is lost. However, for high values of $RPS_{\max}$ such that $RPS_{\max} \gg Pods_{\max}$, this is assumed to be acceptable.

However, the compilation and verification times were quite high in some instances. This is probably due to the large possible number of values of the

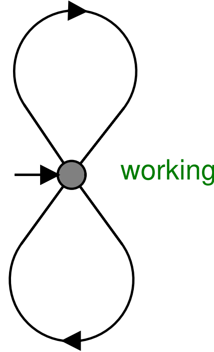

Figure 5.3: The *pod_queue* automaton. QL[Pod_Index] represents the queue length of the current pod. QL_Max represents the maximum queue length.

queue length variables. To limit this, let us introduce a new constant called QL_Max_Limit. This forces the possible values of the queue length variables to be at most this length, and thus decreases the state space, which improves performance. However, then we must adjust the parameters for maximum requests per second and requests handled per second accordingly.

First, we calculate the actual maximum queue length based on the parameters:

$$\mathsf{QL_Max_Actual} = \frac{\mathsf{Max_Response_Time_In_Ms}}{\mathsf{Processing_Time_Per_Req_In_Ms}}$$

Then limit it if required, as follows:

$$\mathsf{QL_Max} = \min(\mathsf{QL_Max_Actual}, \mathsf{QL_Max_Limit})$$

Then calculate the ratio between the actual and limited value:

$$\mathsf{Limiting_Divisor} = \mathsf{QL_Max_Actual}/\mathsf{QL_Max}$$

This will be 1 if the actual maximum queue length is below the limit. Now we can adjust the other parameters also:

$$\mathsf{RPS_Max} = \mathsf{RPS_Max_Actual}/\mathsf{Limiting_Divisor}$$

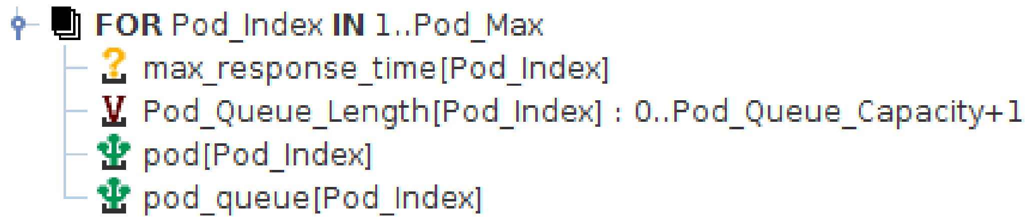

Figure 5.4: The for-each loop in the WATERS model to represent pods (pods) in the cloud.

Finally,

$$\begin{aligned}&\mathsf{Req_Handled_Per_Sec_Per_Pod}\\ &= \mathsf{Req_Handled_Per_Sec_Per_Pod_Actual/Limiting_Divisor}\end{aligned}$$

The downside of this approach is that the limit makes the model less accurate.

5.3 Cloud

To match Kubernetes, the cloud is modelled as a set of pods. The pods are defined using a "for-each" loop in WATERS [42]. This effectively creates a set amount of copies of the same set of state machines. The state machines that are copied are *max_response_time*, QL, *pod*, *pod_queue* and *load_balancer*. This is shown in Figure 5.4.

The individual members of this loop are referred to using array notation, such as pod[1] for the first pod and pod[2] for the second.

Note that the number of copies is always between the parameters Pod_Min and Pod_Max. This corresponds to $pods_{\min}$ and $pods_{\max}$ from the theoretical model. Pod_Min must be at least 1.

The *pod* state machine is shown in Figure 5.5. The event *pods_handle_requests* reduces the queue length by Req_Handled_Per_Sec_Per_Pod. It will never go below zero, since that would make no physical sense. When the *one_second* event fires, the state machine will go back to the *idle* state. The value Req_Handled_Per_Sec_Per_Pod is set to Req_Handled_Per_Sec_Per_Pod_

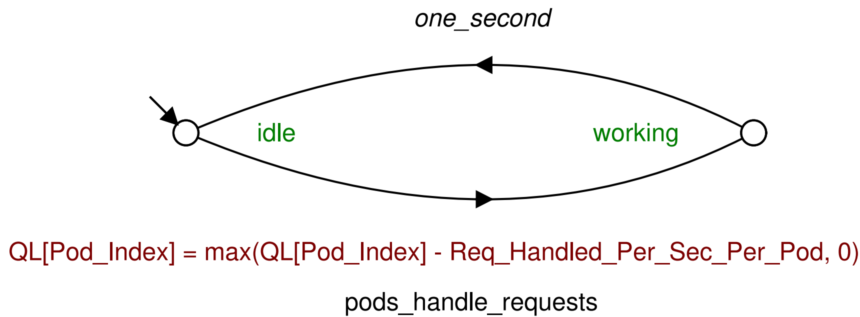

Figure 5.5: The *pod* automaton.

Actual divided by Limiting_Divisor. The value Req_Handled_Per_Sec_Per_Pod_Actual is set to 1000 divided by Processing_Time_Per_Req_In_Ms.

5.4 Horizontal Pod Autoscaler

Scaling is done by the plant *horizontal pod autoscaler* shown in Figure 5.6. This only checks scales up or down at set time intervals of HPA_Check_Interval seconds. The automaton starts in state *idle*. If HPA_Seconds_Elapsed $<$ HPA_Check_Interval, the interval has not yet finished, so the *scaling_start* event transitions to the *no_check* state. From there, the *scaling_finished* event is fired, which transitions to the *done_no_check* state. Finally, the *one_second* event transitions back to the *idle* state.

However, if HPA_Seconds_Elapsed $\geq$ HPA_Check_Interval, the *scaling_start* event transitions to the *started* state. There are two possible transitions next: if the queue length of the last pod currently on is more than or equal to the threshold for scaling up, the number of pods currently on increases by one. Else, it stays at the current value. Either way, this transitions to state *scale_up_ended*. Next, there are two possible transitions: if the queue length of the first pod is less than or equal to the threshold for scaling down, the number of pods currently on decreases by one. Else, it stays at the current value. The number of pods currently on stays within the limits 1 to Pod_Max thanks to the min and max function calls in the actions.

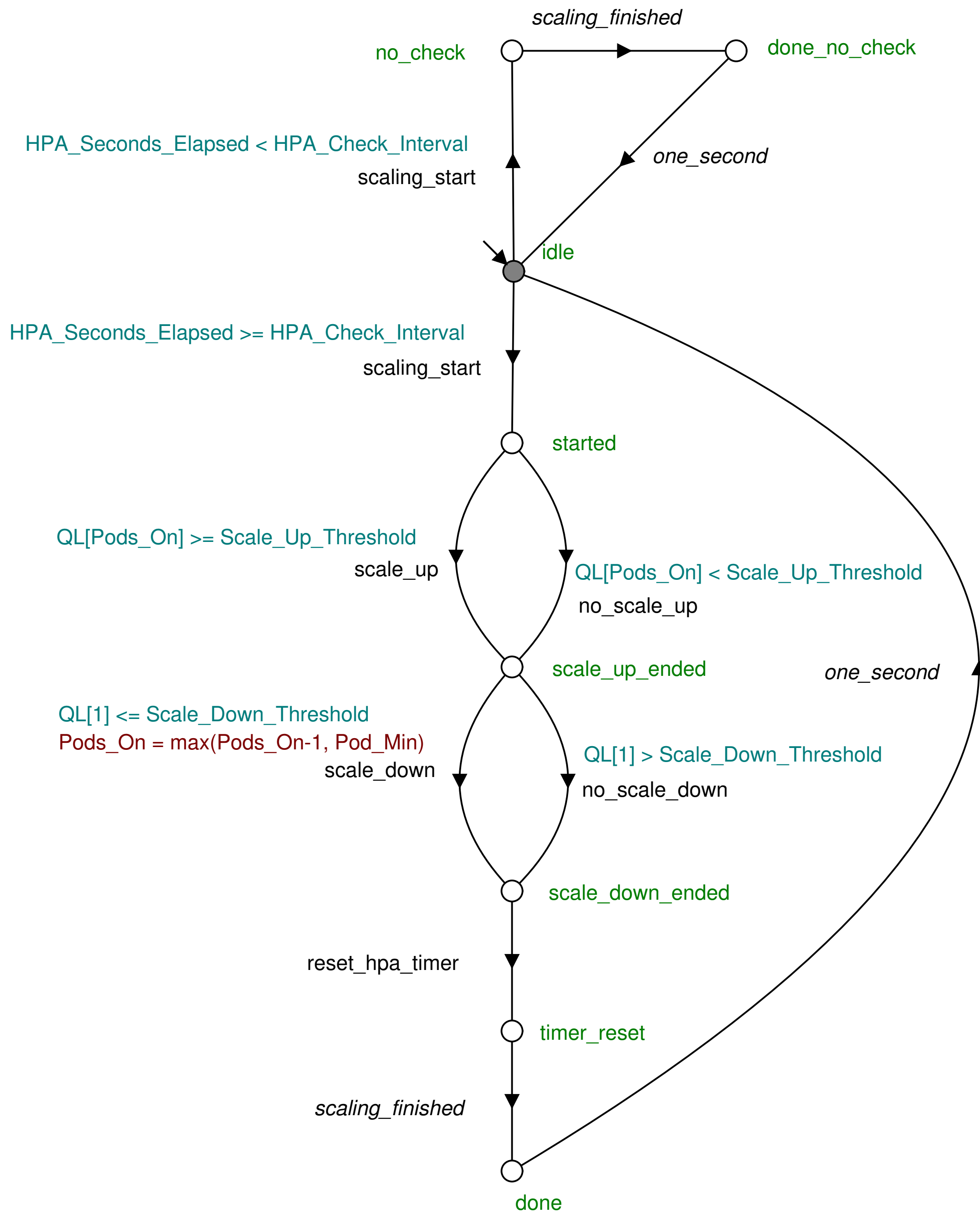

Figure 5.6: The *horizontal_pod_autoscaler* automaton.

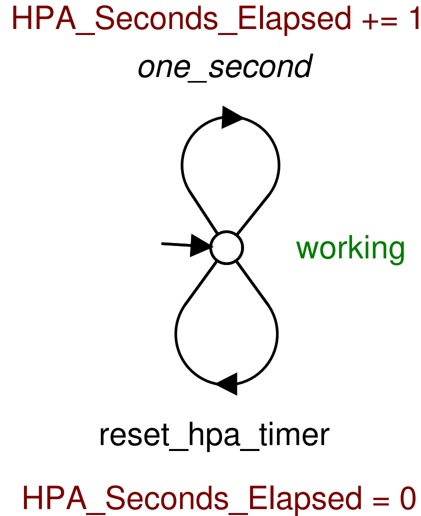

Figure 5.7: The *horizontal_pod_autoscaler_clock* automaton.

The HPA_Seconds_Elapsed variable is managed by the automaton *horizontal_pod_autoscaler_clock* automaton, shown in Figure 5.7.

The startup time for pods $T_{\text{pod-startup}}$ is specified by the parameter Pod_Startup_Time. The starting of the pod is done by the plant *pod_scheduler* shown in Figure 5.8. This starts in state *idle*. When the *horizontal_pod_autoscaler* plant fires the transition *scale_up*, the *pod_scheduler* plant goes to state *creating_pod*. It remains in this state for Pod_Startup_Time seconds via the *one_second* transition. Note that additional firings of *scale_up* while in this state have no effect; this is why HPA_Check_Interval should be greater than Pod_Startup_Time. The next firing of *one_second* will transition back to state *idle*. In doing so, the number of pods (Pods_On) increases by one, up to a maximum of Pod_Max.

One issue with this model is that, after scaling down, the queue length of the pod that was "turned off", may still be greater than zero. This means the remaining requests in the queue are effectively dropped. This can be shown by running a property check for property *unit_test_5_if_pod_is_off_its_queue_is_empty* shown in Figure 5.9. This issue may be resolved changing the model to scale down only when the queue length of the last pod is zero. Alternatively, the "dropped" requests could also be counted as failed requests

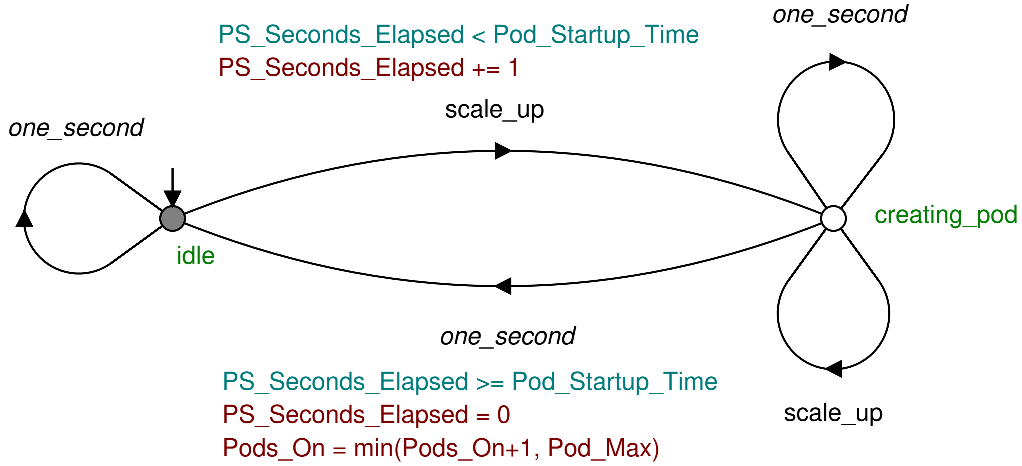

Figure 5.8: The plant *pod_scheduler*. PS_Seconds_Elapsed represents the seconds elapsed for the pod scheduler clock.

toward the SLO; however, that would require keeping track of the number of failed requests, which the model does not do.

5.5 SLA

The SLA is modelled using *properties* in WATERS. These represent the desired properties of the language represented by the model. WATERS checks whether there are any circumstances where the properties are not met (a counterexample). Each property represents one requirements of the SLA (an SLO).

The SLO for maximum response time is modelled by the property *max_response_time* in WATERS. This is shown in Figure 5.10.

This has only one state (*checking*). In order for the property to be met, the transition *slo_check* must always be able to execute. There is a guard condition that only allows this to execute if the queue length is below the maximum queue length ($\mathsf{QL} \leq \mathsf{QL_Max}$). QL_Max is calculated as

$$\mathsf{QL_Max} = \frac{\mathsf{Max_Response_Time_In_Ms}}{\mathsf{Processing_Time_Per_Request_In_Ms}}$$

which corresponds to Equation 4.2. Note that this may be limited by a common divisor as discussed in Section 5.2.

Running a *property check* within WATERS indicates if any SLOs will not

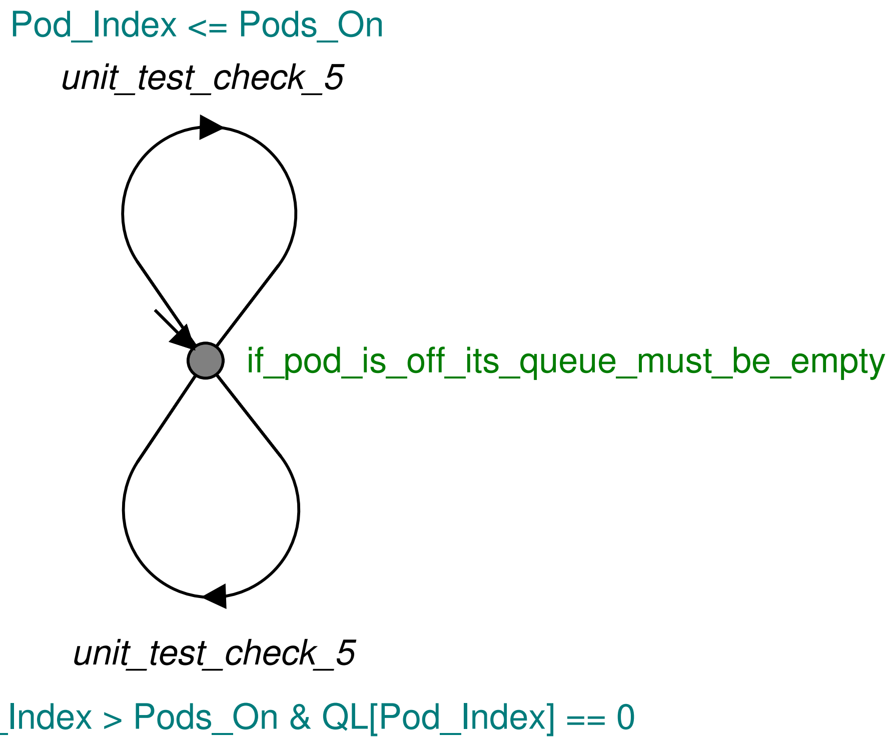

Figure 5.9: The unit test property *unit_test_5_if_pod_is_off_its_queue_is_empty*.

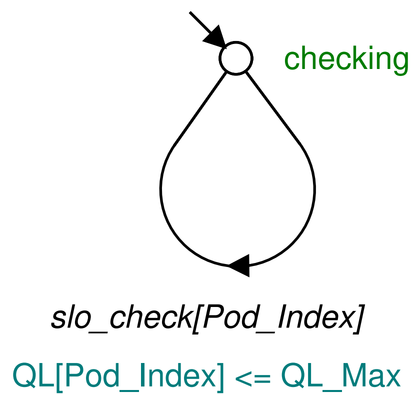

Figure 5.10: The property *max_response_time*.

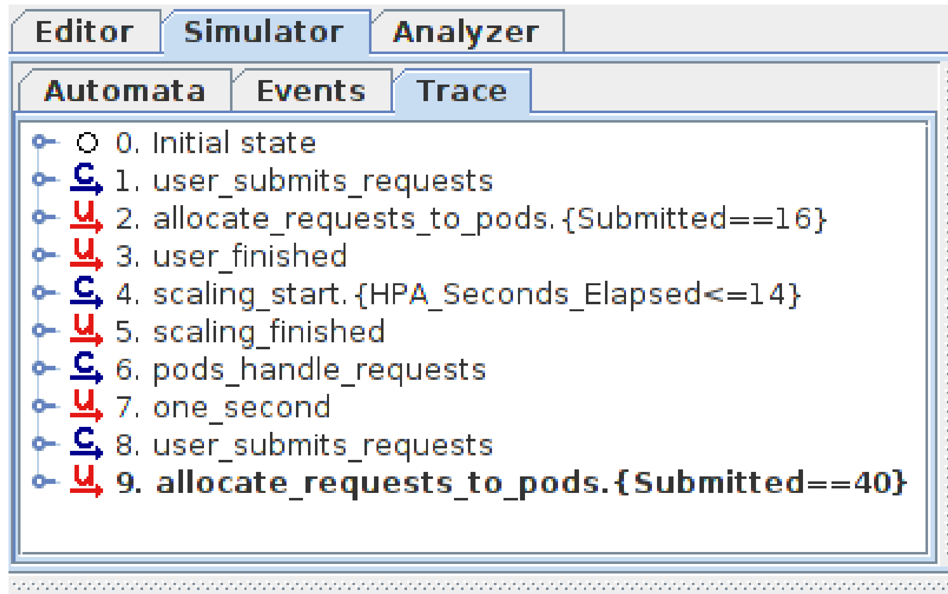

Figure 5.11: A counter-example trace to show a possible SLO violation in WATERS.

be met. WATERS also shows a trace of events that occur to cause the property violation. An example trace to show an SLA violation is presented in Figure 5.11. Each line represents one event. The first line represents the initial state, before any events are fired. The rest of the lines are described below:

1. The event *user_submits_requests* fires in automata *master* and *user*, indicating the user starts to submit requests.

2. The event *allocate_requests_to_pods* fires in automata *user* and *pod_queue*. This divides the requests among the active pods (in this case it is one pod only). Note that the value of Submitted is 16 in this case.

3. The event *user_finished* fires in automata *user* and *master*, indicating the user has finished submitting requests.

4. The event *scaling_start* fires in automata *master* and *horizontal_pod_autoscaler*. This indicates the HPA can scale if required. Note that in this case HPA_Seconds_Elapsed is less than 15, which means no autoscaling will occur, since the HPA checking period has not elapsed yet.

Editor | Simulator | Analyzer

Automata | Events | Trace

Automaton	Act	Mrk	State
HPA_Seconds_Elapsed			1
Next_Pod			1
PS_Seconds_Elapsed			0
Pods_On			1
QL[1]	✓		51
Submitted	✓		40
horizontal_pod_autoscaler		●	idle
horizontal_pod_autoscaler_clock			working
master		○	waiting_for_user_to_finish
max_response_time[1]	!		checking
pod[1]			idle
pod_queue[1]	✓	●	working
pod_scheduler		●	idle
user	✓	○	submitting_done

Figure 5.12: A warning symbol is shown next to the property *max_response_time[1]* to indicate that it is not met.

5. The event *scaling_finished* fires in automata *master* and *horizontal_pod_autoscaler*. This indicates the HPA is done for this second.

6. The event *pods_handle_requests* fires in the pod automata. The pod queues decrease as the pods process requests.

7. The event *one_second* fires in all automata which use the event. Thus, the loop can start again.

8. The event *user_submits_requests* is fired. In this case there are 40 requests fired.

9. The event *allocate_requests_to_pods* fires, this takes the queue length for pod[1] above the maximum.

This will show a warning symbol next to the property in the Automata tab of the Simulator, as shown in Figure 5.12.

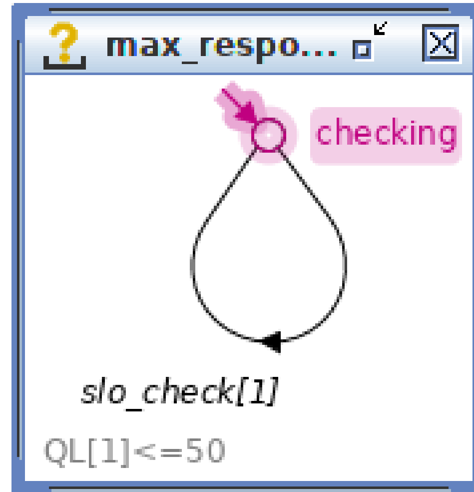

Figure 5.13: The property *max_response_time[1]* shown in the right-hand pane.

Double-clicking on the property will show the automaton on the right-hand pane, as shown in Figure 5.13. The transition line is greyed out to show that it is disabled. The guard for the transition is shown (Pod_Queue_Length[1] <= 50).

5.6 Differences in Node.js Model

The model for the Node.js application differs from the NGINX model in the following ways: 1) a square wave workload instead of a constant workload, 2) the remainder of the requests each second is divided among the queues instead of being discarded, and 3) there is no limiting divisor or related variables.

To represent a square wave workload, we add an automaton to control the workload shape. This is shown in Figure 5.14. Let us also introduce parameters T_High set to T_{high}, RPS_Max_High set to RPS_{high}, T_Low set to T_{low} and RPS_Max_Low set to RPS_{low}. RPS_Max is initially set to RPS_Max_High. The workload automaton starts in *high_load*. On the *one_second* event it increments the value of W_Seconds_Elapsed by one. It stays in this state until W_Seconds_Elapsed is equal to T_High. When this is true, it transitions to state *low_load*. During the transition, the variable RPS_Max is set to RPS_

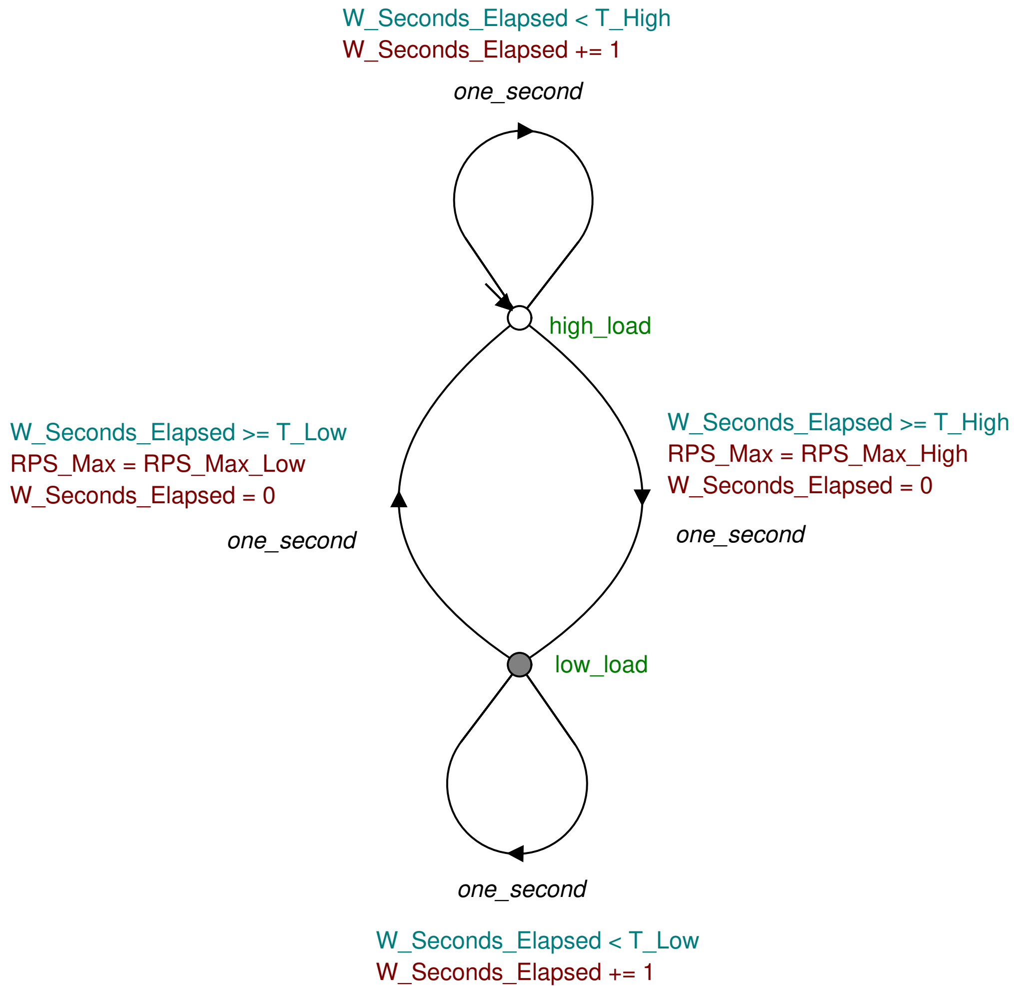

Figure 5.14: The *workload_shape* automaton.

Max_Low and W_Seconds_Elapsed is reset to zero. The automaton then stays in state *low_load* until W_Seconds_Elapsed equals T_Low. Then it transitions back to state *high_load*, setting RPS_Max back to RPS_Max_High and resetting again W_Seconds_Elapsed to zero. This cycle can repeat indefinitely.

To divide the remainder of requests, a modification is made to the user *automaton*, and a new automaton is introduced. The modified *user* automaton is shown in Figure 5.15. The key difference with respect to NGINX is the transition *allocate_req_to_pods_remainder*, which allocates the remainder of requests among the active pods. The new automaton is *pod_queue_remainder* (Figure 5.16), and this is part of the for-each loop representing the pods.

The limiting divisor is not required in the Node.js, since we found that the compilation and verification times for this model were already acceptable

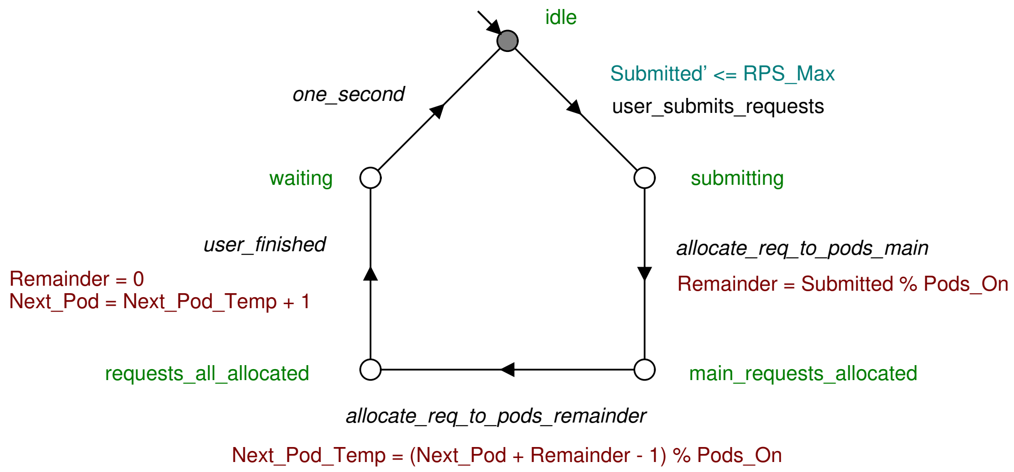

Figure 5.15: The *user* automaton for the Node.js application WATERS model.

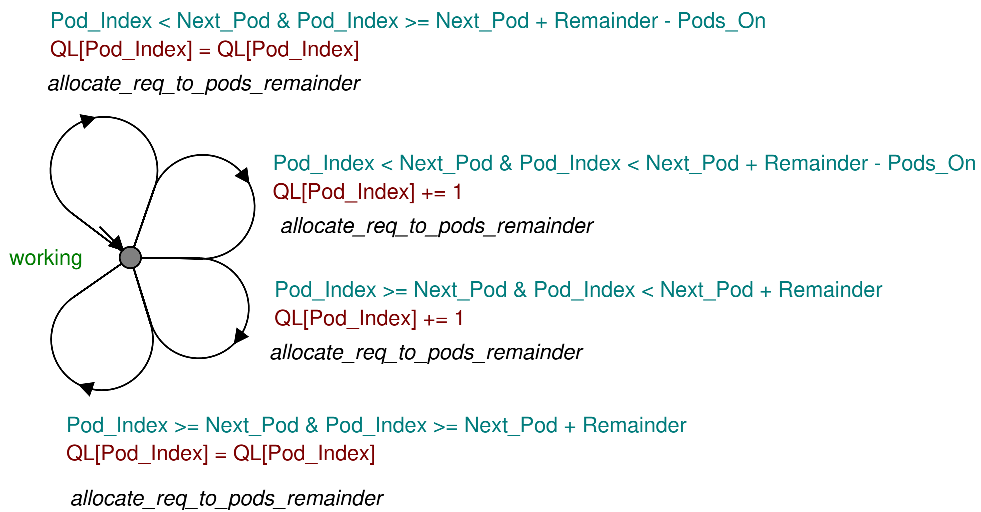

Figure 5.16: The *pod_queue_remainder* automaton that allocates the remainder of requests in the Node.js model.

without it. The maximum queue length is shorter than for NGINX, due to the longer response times of the Node.js application. This means there are fewer possible values of the queue length variables, and thus a naturally smaller state space.

The next chapter presents the experimental setup used to test whether this model is accurate to a real cloud system.

Chapter 6

Experimental Methodology

To test the accuracy of our model, we created a local Kubernetes cluster, drove a workload using JMeter, analysed the results to see whether the SLO was met, and compared this to the results from the WATERS model. This was done for a set of test cases, in which parameters were varied such as the number of requests sent per second and the maximum number of pods the cloud can scale to. Each test case was run for a number of trials. An overview of the experimental methodology is shown in Figure 6.1.

In summary, the following high-level process was used:

1. Model the cloud system in WATERS.
2. Create the real system using Kubernetes.
3. Test the Kubernetes cloud using JMeter using a comprehensive set of test cases.
4. Analyse the results from JMeter using Python scripts to see whether the SLA was met or not.
5. Run verifications in WATERS to see whether it predicts the system will meet the SLA or not.
6. Compare the results from JMeter and WATERS to see how closely they match.

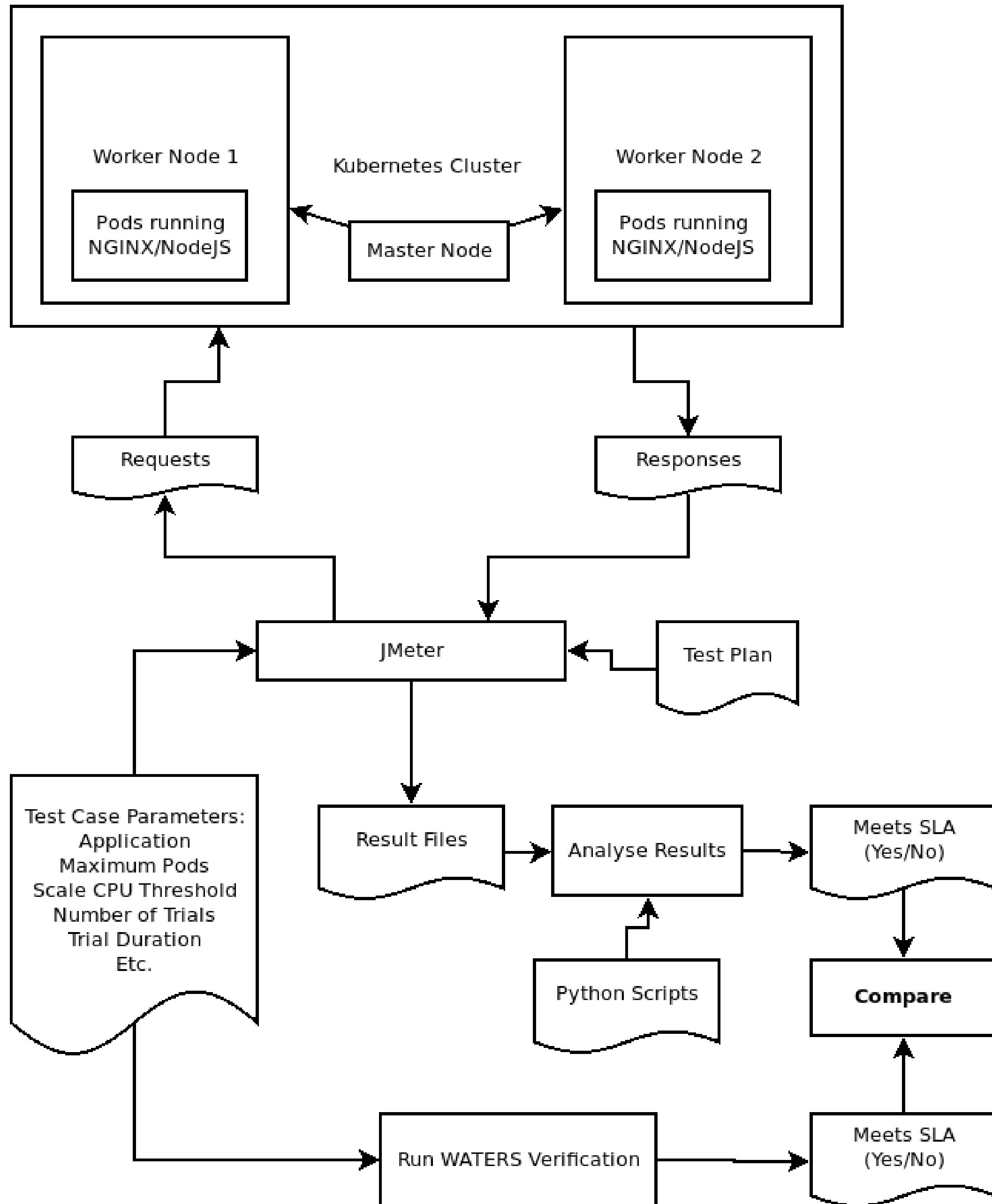

Figure 6.1: The setup for the test cases.

It was decided not to test on a public cloud, since it was likely that outside interference would cause the results to be inconsistent.

6.1 Applications

Two different applications were tested: 1) a default NGINX installation and 2) a Node.js application with a fixed response time. NGINX[1] is a load balancer and reverse proxy that provides a static "Hello World" web page to every request by default. This represents a simple, standalone application with a variable but fast response time. NGINX performs round-robin load balancing by default.

We also tested a simple Node.js application with fixed response time via a semaphore loop. The application sends a unique response to each request, in order to avoid sending "304" (not modified) responses. The default response time is 1 second, but can be set per request using the "rt" parameter (for example, "rt=500" for a response time of 500 milliseconds). In contrast to NGINX, this is a relatively slow, stable response time.

Node.js is single-threaded, allows close control over the sequence in which requests are responded to. This makes it a suitable candidate for a test using a fixed response time. It should be noted, however, that I/O operations are not single-threaded [44]. However, since there are few I/O operations in this simple application, this should not be a concern.

6.2 Kubernetes

The local Kubernetes cloud was created using a *Vagrant* installer [66]. The cloud consisted of one master node and two worker nodes. All nodes were *VirtualBox* Virtual Machines running Ubuntu. The master node had 2 GiB of RAM and 2 vCPUs. The worker nodes each had 1 GiB of RAM and 1 vCPU. The resources on the worker nodes were limited to ensure repeatable response

[1]https://www.nginx.com/

times. All VMs were located on the same physical machine with an Intel(R) Xeon(R) CPU E5-2670 @ 2.60 GHz with 32 virtual cores, 256 GiB of RAM and running Ubuntu 18.04.1 LTS (Bionic Beaver).

For each test case, the Kubernetes settings were configured for the cluster including minimum number of pods, maximum number of pods, initial number of pods, CPU threshold for scaling, and the RAM and CPU limit for each pod. This was done by substituting the values in a template Kubernetes YAML configuration file. These files are described in Chapter 7. In all test cases, the Horizontal Pod Autoscaler checking interval ("–horizontal-pod-autoscaler-sync-period") was kept at the default value of 15 seconds.

Note that Kubernetes may move pods between nodes as required [19], which adds an uncertainty to the system being modelled.

6.3 JMeter

The workload was created using Apache JMeter[2], a load testing tool written in Java. Requests were set to time out after 10 s (RT_{max}): requests that timed out were also counted to determine if the the 99th percentile of response time was below the SLO threshold. For NGINX, the test plan created a constant workload using a *Constant Throughput Timer*. For Node.js, the test plan created a square wave workload using a *Throughput Shaping Timer*.

For each trial, if the 99th percentile of response times was below 10 seconds, then the trial was defined as meeting the SLO. This differs from the theoretical model, since the theoretical model SLO required 100 percent (not 99) of response times to be under 10 seconds. The reason for this difference is that in testing we often encountered requests which go above the maximum response time due to effects we have not modelled, such as garbage collection [49] or JMeter stopping a test before the final few requests had time to be responded to. In addition, percentiles are most often used in real-world SLOs [63]. We

[2]https://jmeter.apache.org/

also found it difficult to model the 99th percentile in WATERS; it is much easier to check whether any request at all goes above the time limit.

Separate test plans were created for constant workload and a square wave workload. The test plan for a constant workload used a *Constant Throughput Timer*[3] to ensure that the requests were sent at a constant rate. However, this did not always provide a constant number of requests per second. "Bursting" was observed: for example, when 1 request per second was specified, JMeter might send 10 requests the first second, then none for the next nine seconds, then 10 requests again the next second. A set number of users (threads) of 1000 was used in the test plan.

The specified request rate would not always be met if it was above about 300 requests per second. If there were many time-outs, the actual request rate would be closer to 250 per second.

Note that the WATERS model allows for any level of traffic up to $RPS_{\max}$, whereas JMeter in this case attempts to provide exactly $RPS_{\max}$. A more comprehensive future test could perhaps include a random component which will send a random number of requests up to a set maximum per second. However, this would also change the predictability of the results.

The test plan for a square wave workload used a *Throughput Shaping Timer*[4] with a *feedback loop*. This allowed setting an interval with a constant high rate of requests and an interval with a constant low rate of requests. The timer then maintains the required number of threads to keep the requests per second value at the specified rate rate. As with the Constant Throughput Timer, this also would not always match the desired rate if it was over about 300 requests per second. Note that this approach can also produce a constant load (by simply having one level instead of two), possibly with less bursting than with the Constant Throughput Timer.

JMeter was run on a desktop PC with an Intel(R) Core(TM) i7-8700 CPU

[3]https://www.blazemeter.com/blog/how-use-jmeters-throughput-constant-timer/

[4]https://www.blazemeter.com/blog/using-jmeters-throughput-shaping-timer-plugin/

@ 3.20 GHz with 12 cores, 16 GiB of RAM and running Ubuntu 18.04.2 LTS. This machine was located in the same building as the Kubernetes cloud, so we assume that network latency between JMeter and Kubernetes was negligible.

A number of trials (between 1 and 5) was run for each test case. The proportion of trials meeting the SLO was recorded in each case. Where a true or false result is required, if half or more of the trials met the SLO, the result was defined as meeting the SLO.

Our goal was to include an even balance of test cases which satisfied the SLO and which did not. These are a *positive test* and *negative test*, respectively. This was most often done on a trial-and-error basis, since it was difficult to determine beforehand which parameters would cause the SLO to be violated. We also ran only a limited number of tests, due to time constraints.

6.4 WATERS

After running the JMeter tests, the same test cases were run through WATERS, using the models for NGINX and Node.js. The verifications were run using WATERS v2.5.1, built on 27 Nov 2019. This was done with the *wcheck* script that comes with the WATERS installation, using the options "wcheck -bdd -lang -q -stats". The *-bdd* option instructs WATERS to perform verification using Binary Decision Diagrams (BDDs). Using BDDs has been shown to increase speed of verification for models with large state spaces [65]. The *-lang* option instructs WATERS to perform a language inclusion check (that is, check all properties). The *-q* option tells WATERS not to list a counter-example if a property is not met. The *-stats* option produces additional statistics, including compilation and verification times.

The experimental results were compared using a spreadsheet, checking the SLA satisfaction prediction from WATERS to JMeter's results. The average accuracy was recorded as noted in Section 7.

One of our goals was to ensure the overall time to verify a single cloud

system model for one test case in WATERS was under 10s. Thus, we checked the compilation and verification times from the statistics logged by WATERS using the *-stats* option (WATERS first compiles the model, then verifies it). The assumption is made that, if the same model is verified twice, it will take the same time in both instances since WATERS runs the exact same algorithm each time. Therefore WATERS verifications were only run once for each benchmark and only the execution time for that was recorded.

However, to prove this is the case, one benchmark was verified 10 times and the standard deviation of WATERS execution time calculated. This is shown in Table 6.1. The model *model-2-02-Node.js.wmod*[5] was run with parameters RPS_Max=2, Pod_Max=4, Processing_Time_Per_Req_In_Ms=700 and Scale_ Cpu_Threshold=80.

Table 6.1: Test to check the consistency of WATERS execution times.

Trial	**Total Time**
1	0.76
2	0.62
3	0.63
4	0.60
5	0.59
6	0.59
7	0.59
8	0.60
9	0.61
10	0.60

Note that the first trial (.76 seconds) took significantly longer than the following 9 trials. This is perhaps due to caching on the machine. In any case, the first instance is the worst case scenario, and since the goal is to ensure

[5]https://github.com/martinvanzijl/masters-project/blob/master/models/model-2-02-Node.js.wmod

that compilation and verification time does not exceed 10 seconds, running each test case once is sufficient for timing purposes.

The next chapter presents the results obtained using this experimental methodology.

Chapter 7

Results and Analysis

The results from testing the applications on our Kubernetes cluster and comparing with WATERS predictions are presented in this chapter.

Each test case has four possible results: a *true positive* indicates the JMeter results and WATERS property check result agree the SLO is met. A *true negative* indicates the JMeter and WATERS result agree the SLO is not met. A *false positive* occurs when the JMeter results show the SLO is not met, but the WATERS property check predicts the SLO is met. A *false negative* occurs when the JMeter results show the SLO was met, but the WATERS property check predicts the SLO is not met.

Our aim was to avoid false positives, but false negatives are somewhat acceptable. This is because firstly, the penalty to the provider for over-provisioning due to a false negative is likely to be less than the penalty for violating the SLA due to a false positive. Secondly, a false negative may occur because WATERS has discovered a failing scenario that was not encountered in the JMeter tests: essentially, WATERS verifies all possible combinations; whereas, JMeter validates only a small subset.

7.1 Application 1: NGINX

The first application tested was NGINX. The system model is as presented in Chapter 5, using a constant workload, and without allocating the remainder

of requests. The test cases for this application all used a high number of requests per second with a relatively low number of maximum pods, so that $RPS_{\max} \gg pods_{\max}$. The parameters to this model are listed in Table 7.1. Each of these may be set in the WATERS GUI or on the command line when testing the model. If no value is specified then the default value was used.

Table 7.1: Parameters for the NGINX model.

Parameter	Symbol	Default	Description
HPA_Check_Interval	$T_{\text{scale-check}}$	15	Number of seconds between checks by the HPA.
RPS_Max_Actual	$RPS_{\max}$	200	Maximum requests per second.
Max_Response_Time_In_Ms	$RT_{\max}$	10000	Maximum response time in milliseconds.
Pods_Initially_On	$Pods_{initial}$	1	Number of starting pods.
Pod_Min	$Pods_{\min}$	1	Minimum number of pods to scale down do.
Pod_Max	$Pods_{\max}$	2	Maximum number of pods to scale up to.
Processing_Time_Per_Req_In_Ms	$PT_{\max}$	6	Processing time per request in milliseconds.
QL_Max_Limit		50	Highest possible value of QL_Max.
Pod_Startup_Time	$T_{\text{pod-startup}}$	5	Time(s) pod to start processing requests after being scheduled.
Scale_Down_CPU_Threshold	ST_{down}	20	Percentage of QL_Max for first pod before scaling down.
Scale_CPU_Threshold	ST_{up}	80	Percentage of QL_Max for last pod before scaling up.

The pod startup time $T_{\text{pod-startup}}$ for this application was measured by configuring the HPA to have 1 pod minimum, killing all existing pods, waiting for the HPA to start a new pod, waiting for the pod to be ready, then inspecting the logs to see how long the pod took to start up. The difference in time between the *Scheduled* and *Started* states was taken as the pod startup time, taken from the "kubectl describe pods" command. This was done 10 times and the average used for the final figure. In this case the average was 5 seconds.

The response time for the application was measured by running a one-minute JMeter test sending one request per second. In this trial, the average response time was 18 ms, the minimum was 8 ms and the maximum was 112 ms. According to the theoretical model, the maximum value (112 ms) should be used for $RT_{\max}$. However, when testing the model using different values of the Processing_Time_Per_Req_In_Ms parameter, we found that the most accuracy is obtained when the value is between 6 and 10 milliseconds. Using a value of 6 ms was most accurate overall (68%) but yielded a relatively high

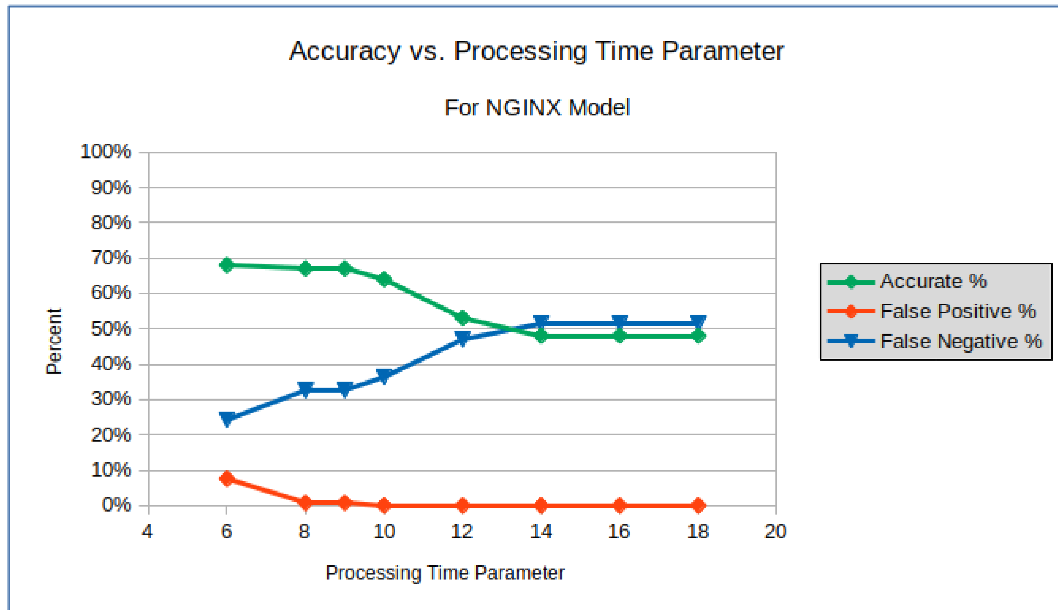

Figure 7.1: A graph of accuracy versus the processing time parameter Processing_Time_Per_Req_In_Ms for the NGINX model. All other parameters were kept at their default values.

number of false positives (8%). Since false positives are to be avoided, using a value of 8 ms is more suitable. This had an overall accuracy of 67% and only 1% false positives. An even safer value is 10 ms, which had an overall accuracy of 64% and no false positives. The percentage of verifications that are accurate, false positives and false negatives for different values of Processing_Time_Per_Req_In_Ms is graphed in Figure 7.1.

The raw results are presented in Appendix A. In these results value of Processing_Time_Per_Req_In_Ms was set to 6. It is convenient to summarise the results in terms of true and false positives and negatives. We will show this for two different values of the Processing_Time_Per_Req_In_Ms parameter. Table 7.2 shows the results when Processing_Time_Per_Req_In_Ms is set to 6, and Table 7.3 shows the results when Processing_Time_Per_Req_In_Ms is set to 10.

As discussed previously, the case where Processing_Time_Per_Req_In_Ms set to 6 is more accurate overall, but has a high amount of false positives (20). The case where Processing_Time_Per_Req_In_Ms is set to 10 is slightly less

Table 7.2: Analysis of NGINX results, with Processing_Time_Per_Req_In_Ms set to 6.

	Test: SLA Met	**Test: Not Met**
WATERS: SLA Met	75	20
WATERS: Not Met	20	29

Table 7.3: Analysis of NGINX results, with Processing_Time_Per_Req_In_Ms set to 10.

	Test: SLA Met	**Test: Not Met**
WATERS: SLA Met	36	0
WATERS: Not Met	48	86

accurate, but importantly has no false positives. We hypothesise that setting this parameter to slightly above the minimum response time is appropriate, to account for delays not modelled currently, such as network latency and performance interference. Further experiments using different applications with variable response times would be required to test this.

A few false negatives are to be expected, since WATERS checks more scenarios than are tested by JMeter. In JMeter, a near-constant load was produced for each test, but WATERS checks every possible load combination up to the maximum requests per second. So, it is to be expected that WATERS finds counter-examples that were not triggered by JMeter. A good example of this is the scenario where Processing_Time_Per_Req_In_Ms = 10, Min_Pods = 1, Max_Pods = 4, Initial_Pods = 1, Scale_CPU = 75. The test results from JMeter indicate that this meets the SLA, but WATERS indicates it does not. Examining the counter-example trace in WATERS shows that, upon the HPA check, the queue length is exactly one less than the value required to scale out. It is unlikely that this exact worst-case scenario was encountered in the JMeter tests.

One limitation of this test is that the noise-to-signal ratio is possibly quite high. The reason for this is that the server (NGINX) responds very quickly

with a small web-page. Often it uses a "cached" 304 response. This means that the delay caused by networking latency may be more than that caused by the cloud system items.

We also investigated the effect of the limiting divisor discussed in Chapter 5. In general, the variable with the largest possible number of values is QL, and there are effectively Max_Pods copies of this variable, since there is a copy for each pod. Therefore, we expect the compilation and verification (T_{c+v}) of WATERS to increase according to the formula:

$$T_{c+v} \propto \mathsf{QL_Max}^{(\mathsf{Max_Pods}+1)}$$

A graph of the average WATERS compilation and verification time versus the limiting parameter QL_Max_Limit is shown in Figure 7.2. A graph of the accuracy of the model versus the limiting parameter QL_Max_Limit is shown in Figure 7.3. Here the processing time parameter Processing_Time_Per_Req_In_Ms is set to 6.

This shows that the accuracy increases up to a point (in this case at 68% accuracy when QL_Max_Limit = 50), then the accuracy stays constant. The key is the ratio of incoming to processed requests:

$$\mathsf{RPS_Max/Req_Handled_Per_Sec_Per_Pod}$$

This is analysed further in Table 7.4, and shows that this value changes relatively little for values of QL_Max_Limit 50 and above. This is most likely the reason why the accuracy does not improve any further.

An interesting effect is observed when the response time parameter Processing_Time_Per_Req_In_Ms is set to 10. All test cases with $RPS_{\max}$ set to 150 or above are predicted to fail the SLA, and all test cases with $RPS_{\max}$ set to 100 or below are predicted to satisfy the SLA. This suggests the model is not fine-grained enough for values above 10 for $RPS_{\max}$.

In contrast, when Processing_Time_Per_Req_In_Ms is set to 6, all test cases where $RPS_{\max}$ is set to 250 or above are predicted to fail the SLA. This still suggests that the model is "under-fitting" the real process. All test cases with

Table 7.4: Analysis of effect of the limiting divisor on accuracy. Omitted are the variables which are the same in each case.

RPS_Max_ Actual	**QL_Max_ Limit**	**Limiting_ Divisor**	**RPS_ Max**	**Req_Handled_ Per_Sec_Per_Pod**	**RPS_Max / Req_Handled_Per_Sec_Per_Pod**
50	6	277	0	0	N/A
50	12	138	0	1	0.00
50	25	66	0	2	0.00
50	50	33	1	5	**0.20**
50	100	16	3	10	**0.30**
50	200	8	6	20	**0.30**
50	400	4	12	41	**0.29**
250	6	277	0	0	N/A
250	12	138	1	1	1.00
250	25	66	3	2	1.50
250	50	33	7	5	**1.40**
250	100	16	15	10	**1.50**
250	200	8	31	20	**1.55**
250	400	4	62	41	**1.51**
400	6	277	1	0	N/A
400	12	138	2	1	2.00
400	25	66	6	2	3.00
400	50	33	12	5	**2.40**
400	100	16	25	10	**2.50**
400	200	8	50	20	**2.50**
400	400	4	100	41	**2.44**

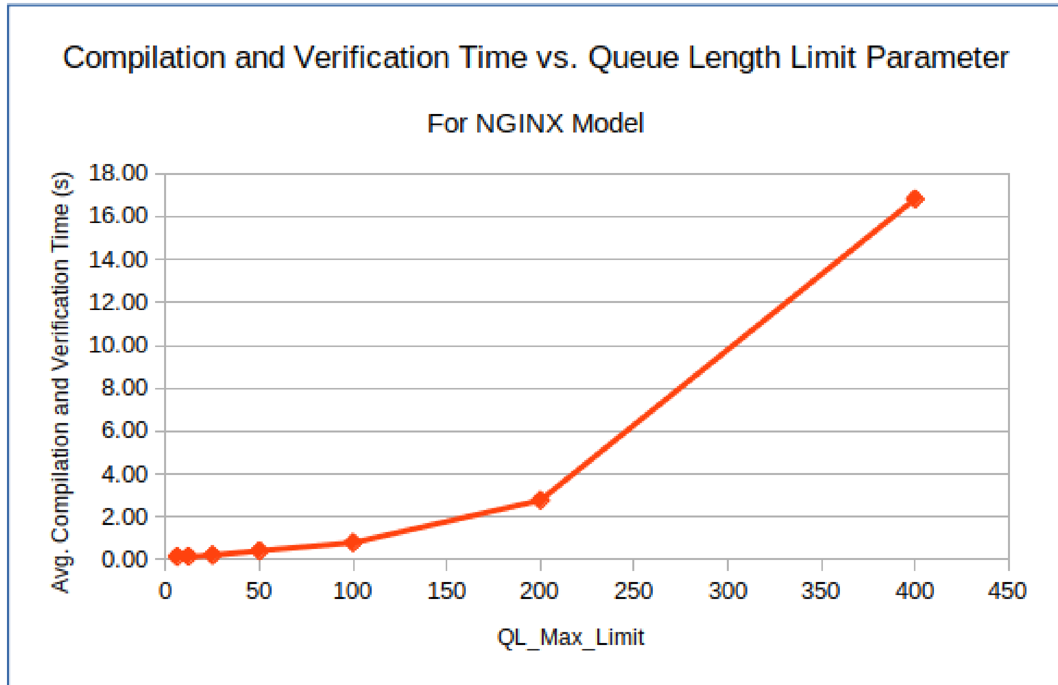

Figure 7.2: A graph of the average WATERS compilation and verification time versus the QL_Max_Limit parameter. For each point, all test cases were verified in WATERS using the given value of QL_Max_Limit. Processing_Time_Per_Req_In_Ms was set to 6.

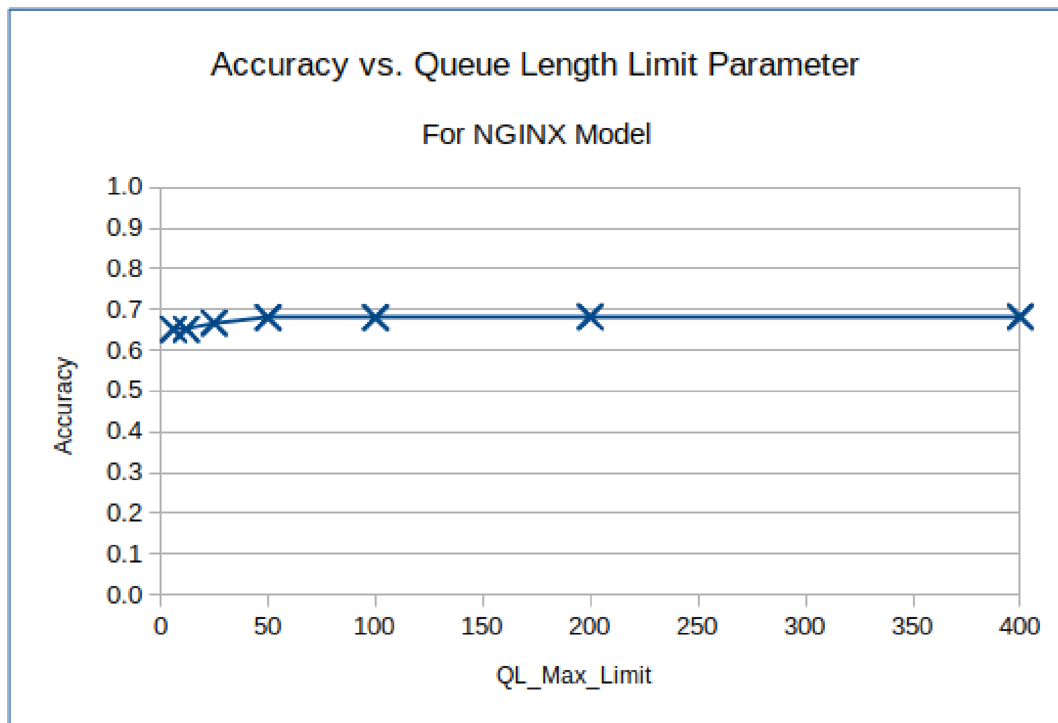

Figure 7.3: A graph of the model accuracy versus the QL_Max_Limit parameter.

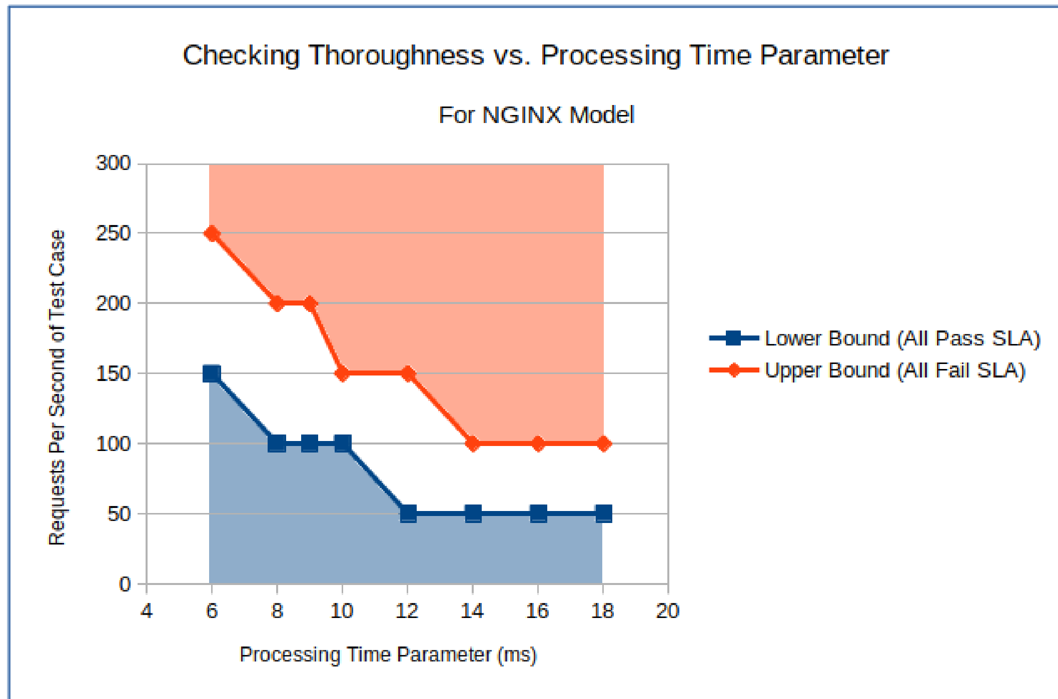

Figure 7.4: A graph showing the impact of the processing time parameter for the NGINX model. Test cases with $RPS_{\max}$ equal to or above the top line are predicted to fail the SLA. Test cases with $RPS_{\max}$ equal to or below the bottom line are predicted to meet the SLA. Test cases between the two lines have a mixture of predictions.

$RPS_{\max}$ set to 150 or below are predicted to satisfy the SLA. This suggests that for each value of Processing_Time_Per_Req_In_Ms there is a lower bound below which each test case is predicted to satisfy the SLA, and an upper bound above which each test case is predicted to satisfy the SLA.

In fact, we can see a general pattern for this, where for all values of Processing_Time_Per_Req_In_Ms, there is such a lower bound and upper bound, which is demonstrated in Figure 7.4. The actual bound where there is a mixture of true or false predictions is rather narrow.

When Processing_Time_Per_Req_In_Ms is set to 6, then only for test cases where $RPS_{\max} = 200$ are the effects of autoscaling actually investigated by the model. All other cases are marked as true or false regardless of the values of the minimum and maximum pods, initial pod, scale CPU threshold and so on. For Processing_Time_Per_Req_In_Ms set to 10, only for test cases where $RPS_{\max} = 150$ are the effects of autoscaling investigated by the model.

Max Pods					
4	TP	TP	FN	FN	FN
3	FP	TP	FN	FN	FN
2	FP	FP	TN	FN	TN
1	TN	TN	TN	TN	TN
	25	50	75	80	100
	Scale CPU Threshold				

TP: True Positive
FP: False Positive
TN: True Negative
FN: False Negative

Figure 7.5: Results of NGINX model for processing time parameter set to 6 ms and maximum requests per second equals 200.

Also of note is that for values of Processing_Time_Per_Req_In_Ms of 14 and above, the upper bound is 100 and the lower bound is 50. This means that the real effects of auto-scaling are not investigated at all by the model in these cases.

According to the experimental results, the SLA was actually met in all cases when RPS was 100 or less. This is the true "lower bound". In this regard the results when the processing time parameter is between 8 and 10 matches. There is no real "upper bound" since there were true and false results even for the highest value of $RPS_{\max}$ that was tested (400).

Let us inspect an example where the effects of autoscaling were checked, namely Processing_Time_Per_Req_In_Ms = 6 and RPS_Max = 200, shown in Figure 7.5. In the experimental results, the maximum number of pods seems to have a greater effect than Scale CPU Threshold on meeting the SLA. In cases where Max Pods is 2, 3, and 4, WATERS predicts those with Scale CPU of 50% and below to meet the SLA, but with 75% and above to fail. In contrast, the experimental results mostly indicate that configurations with Max Pods of 2 and below fail the SLA, but with 3 and above meet it; the Scale CPU has little effect. This suggests that the effect of Scale CPU Threshold in the WATERS model is overestimated.

The same pattern manifests itself for other values. For example, with Processing_Time_Per_Req_In_Ms = 8 and $RPS_{\max}$ = 150. The WATERS model still relies too heavily on Scale CPU whereas maximum number of pods

has the greatest influence in the experimental results.

7.2 Application 2: Node.js Program

The system model differs from that used for NGINX in that it uses a square wave workload and does divide the request remainder. The ratio of requests per second to maximum pods is quite low, so dividing the remainder was considered appropriate. The parameters for the model are listed in Table 7.5.

Table 7.5: Parameters for the Node.js model.

Parameter	Symbol	Default	Description
HPA_Check_Interval	$T_{\mathrm{scale-check}}$	15	Number of seconds between checks by the HPA.
Max_Response_Time_In_Ms	$RT_{\max}$	10000	Maximum response time in milliseconds.
Pods_Initially_On	$Pods_{initial}$	1	Number of starting pods.
Pod_Min	$Pods_{\min}$	1	Number of starting pods.
Pod_Max	$Pods_{\max}$	4	Minimum number of pods to scale down do.
Processing_Time_Per_Req_In_Ms	$PT_{\max}$	1000	Maximum number of pods to scale up to.
RPS_Max_High	$RPS_{\max(\mathrm{high})}$	2	Max. requests per second in high part of square wave.
RPS_Max_Low	$RPS_{\max(\mathrm{low})}$	1	Max. requests per second in low part of square wave.
T_High	T_{high}	60	Duration of high part of square wave.
T_Low	T_{low}	60	Duration of low part of square wave.
Pod_Startup_Time	$T_{\mathrm{pod-startup}}$	14	Time(s) pod to start processing requests after being scheduled.
Scale_Down_CPU_Threshold	ST_{down}	20	Percentage of QL_Max for first pod before scaling down.
Scale_CPU_Threshold	ST_{up}	80	Percentage of QL_Max for last pod before scaling up.

The template Kubernetes YAML file and source code for the application are available on GitHub. Note that the desired response time may be given as a parameter to the HTTP request; for example, “rt=500” for a response time of 500 milliseconds. The default is 1000 milliseconds (or 1 second).

The pod startup time $T_{\mathrm{pod-startup}}$ for this application was measured in the same way as for NGINX. The average startup time was 14 seconds. Note that this is almost the same as the HPA checking period of 15 seconds.

The raw results are presented in Appendix A. The results can be summarised in terms of true and false positives and negatives as per Table 7.6. One observation is that the vast majority of test cases did not meet the SLA (190), and only a small number actually met the SLA (38). Ideally the test cases should have included a more even distribution.

Table 7.6: Analysis of Node.js results.

	Test: SLA Met	**Test: Not Met**
WATERS: SLA Met	31	41
WATERS: Not Met	7	149

Overall, the accuracy of the model is quite high (79%) but there is a large number of false positives (41). This indicates that perhaps the processing time parameter in the model should have an addition made to it to account for effects of networking delays and so on. Ideally, there should be no false positives.

This model does not seem to suffer from the "Upper Bound" and "Lower Bound" problem encountered by the NGINX model. In fact, all test cases with processing time less than 500 ms are predicted to meet the SLA, but this matches the experimental results. Even for the maximum tested value, the WATERS predictions include a mixture of true and false values. From this, we can infer that the effects of auto-scaling are always investigated by the model. This suggests that perhaps the limiting divisor approach used in the NGINX model is flawed.

There was one instance of a verification time over 10 seconds (11.03s), which occurred when $RPS_{\text{low}} = 1$, $RPS_{\text{high}} = 1$, $T_{\text{low}} = 60$, $T_{\text{high}} = 60$, $PT_{\text{max}} = 1000$, minimum, maximum and initial pods are all 4, and the Scale CPU Threshold is 80%. The prediction is for the test case to fail the SLA (which is correct), and it is possible that the verification was slow because the counter-example is quite long. This suggests that for a maximum number of pods of 5 or greater, verification times may grow rather large.

7.3 Discussion

We can classify the inaccuracy of our models into the following reasons: Firstly, the model produces counter-examples that are not encountered in the JMeter

tests. This produces additional "false negatives", which are acceptable since they may in fact occur during runtime.

Secondly, inaccuracies may arise because our models do not follow Kubernetes autoscaling functionality exactly, as discussed in Chapter 5. This is the primary area we wish to avoid inaccuracies for, since our goal is to model Kubernetes autoscaling. This in turn can be divided into two reasons: Firstly due to scaling based on queue length and not CPU (although we do assume these are directly proportional), and secondly due to the difference in algorithm for how many pods to scale up or down by. In order to determine the amount of inaccuracy caused by each reason, we must examine 1) how does average CPU utilisation relate to queue length? and 2) what proportion of times did the HPA scale out or in by more than one pod at a time?

In order to determine the CPU utilisation of a pod versus the queue length of the pod, we must measure both. Unfortunately, this is difficult to do, since by default there is no queue length metric measured. It is possible to set this up using Prometheus[1], but due to time constraints this was not done.

The proportion of times the HPA scaled by more than one pod can be most accurately determined by inspecting the Kubernetes logs. However, the relevant log files were not kept, so unfortunately due to time constraints this was not determined.

Thirdly, inaccuracies occur due to the incompleteness of the model. Aspects we have not modelled include the effects of garbage collection, variability in times taken to create or remove additional pods, possible failures to start up new pods, and so on. Since the current models are simple prototypes, this is not of great concern.

Since NGINX is written in C++, we hypothesise that garbage collection had little to no effect on processing time. In the Node.js program, there are few objects created on the heap, and due to the fixed response time, the code is run infrequently. Therefore we also hypothesise that garbage collection had

[1] `https://prometheus.io/`

little effect on it.

Variations in pod start-up times are likely to have had a greater effect. For the Node.js program, the standard deviation of the startup times measured was 5.5 seconds, which is 0.4 of the mean (14 seconds). For NGINX, the standard deviation was 2.25 seconds, which is about 0.5 of the mean (5 seconds). However, for NGINX the effect is not as pronounced as the statistics suggest, since the mean is small.

Overall, the results are promising, but more work is required to improve their accuracy and reduce false positives. We hypothesise that this includes making the models more closely match the Kubernetes HPA, and perhaps to add a buffer to the processing times to account for aspects we have not modelled.

Chapter 8

Conclusions and Future Work

In this thesis, we introduced a framework for verifying cloud autoscaling policies using WATERS. The cloud is modelled using extended finite state machines in WATERS. Parameters are passed to the model to represent the autoscaling policy, and WATERS verifies if this will meet an SLO related to maximum response time.

We used WATERS to model a homogeneous cloud running one of two applications: NGINX, and a Node.js program with fixed response time. In both cases, the SLA consisted of a single SLO — that the maximum response time is 10 seconds. To check the accuracy of our models, we ran tests using JMeter and Kubernetes to see in what scenarios the SLA is met (or not), and compared this with the predictions from WATERS.

Results indicate that, with suitable parameters, the approach is useful for filtering out policies that do not meet the SLA. There is, however, a rather high rate of false negatives. Nevertheless, much of this can be attributed to WATERS checking the worst-case scenario, which may not have occurred during experiment execution.

A key uncertainty that model checking allowed us to explore is the traffic pattern incoming to a cloud system. WATERS allowed checking any traffic pattern up to a given maximum requests per second against the SLA. It would take a very long time to test all similar possibilities using a load-testing tool.

One limitation was that the compilation and verification times were high for large parameter values (for example, 400 requests per second). In order to keep the compilation and verification times under our targeted maximum of 10 seconds, we used a work-around of a “limiting divisor” in the NGINX model, which limited the state space but lost some precision.

Our approach suffers somewhat from simplifying assumptions and incompleteness [69], since we have only modelled a certain part the cloud system and made assumptions about the rest. This is necessary to keep model compilation and verification time reasonable, and to avoid the models being overly complex. It also suffers from model drift, because of checking only at design time and not at runtime. However, the models could potentially be made more complete and assumptions removed.

A key difference to related research is the use of non-probabilistic model checking. Another difference was the modelling of the inner working of the autoscaling process, instead of using machine learning or related approaches which abstract away this type of detail. This type of model checking would be useful for mission-critical applications where formal guarantees are required, or for creation of strict SLAs by cloud service providers.

8.1 Future Work

Future work could explore different methods to reduce compilation and verification times of the NGINX model, and thus eliminate the need for the limiting divisor. For example, we have used the “-bdd” flag in WATERS, but other flags may prove more efficient in some circumstances.

The models presented in this thesis have only been tested using a limited range of parameter values. For example, the Pod_Max parameter was only tested from values 1 to 4. Future work could include testing this for values above 5, and seeing whether the checking process is still under 10 seconds.

Finally, scaling based on queue length instead of CPU in Kubernetes could

be implemented and tested (for example, with a custom metric using Prometheus[1]). This may provide a more direct measurement of the accuracy of our models.

[1] https://prometheus.io/

Appendix A

Raw Results

This appendix presents the raw results from testing the NGINX and Node.js applications, as summarised in Chapter 7. Testing was performed over a number of days. If the same test case was run on two separate days, only the later day's results were kept.

A.1 NGINX

The raw results from testing the NGINX application are presented in Tables A.1 through A.7. Each row represents one test case. Each test case was performed for a number of trials.

A.2 Node.js

The raw results from testing the Node.js application are presented in Table A.8 through A.13.

Table A.1: Raw results from testing the NGINX application for RPS = 50.

RPS	Min Pods	Max Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
50	1	1	1	25	2/3	Yes	2/3	0.17
50	1	1	1	50	3/3	Yes	3/3	0.19
50	1	1	1	75	2/3	Yes	2/3	0.17
50	1	1	1	100	3/3	Yes	3/3	0.17
50	1	2	1	25	2/3	Yes	2/3	0.19
50	1	2	1	50	3/3	Yes	3/3	0.19
50	1	2	1	75	3/3	Yes	3/3	0.20
50	1	2	1	100	2/3	Yes	2/3	0.20
50	1	3	1	25	3/3	Yes	3/3	0.22
50	1	3	1	50	2/3	Yes	2/3	0.22
50	1	3	1	75	3/3	Yes	3/3	0.22
50	1	3	1	100	3/3	Yes	3/3	0.22
50	1	4	1	25	3/3	Yes	3/3	0.24
50	1	4	1	50	2/3	Yes	2/3	0.23
50	1	4	1	75	3/3	Yes	3/3	0.23
50	1	4	1	100	3/3	Yes	3/3	0.28

Table A.2: Raw results from testing the NGINX application for RPS = 100.

RPS	**Min Pods**	**Max Pods**	**Initial Pods**	**Scale CPU**	**Prop. Meeting**	**WATERS Pred.**	**Accuracy**	**Ver. Time (s)**
100	1	1	1	25	2/3	Yes	2/3	0.19
100	1	1	1	50	2/3	Yes	2/3	0.19
100	1	1	1	75	3/3	Yes	3/3	0.19
100	1	1	1	80	3/3	Yes	3/3	0.22
100	1	1	1	100	3/3	Yes	3/3	0.20
100	1	2	1	25	3/3	Yes	3/3	0.27
100	1	2	1	50	3/3	Yes	3/3	0.24
100	1	2	1	75	3/3	Yes	3/3	0.24
100	1	2	1	80	3/3	Yes	3/3	0.27
100	1	2	1	100	2/3	Yes	2/3	0.25
100	1	3	1	25	3/3	Yes	3/3	0.27
100	1	3	1	50	2/3	Yes	2/3	0.26
100	1	3	1	75	3/3	Yes	3/3	0.26
100	1	3	1	80	3/3	Yes	3/3	0.27
100	1	3	1	100	2/3	Yes	2/3	0.28
100	1	4	1	25	3/3	Yes	3/3	0.31
100	1	4	1	50	3/3	Yes	3/3	0.28
100	1	4	1	75	3/3	Yes	3/3	0.28
100	1	4	1	80	3/3	Yes	3/3	0.28
100	1	4	1	100	3/3	Yes	3/3	0.30

Table A.3: Raw results from testing the NGINX application for RPS = 150.

RPS	**Min Pods**	**Max Pods**	**Initial Pods**	**Scale CPU**	**Prop. Meeting**	**WATERS Pred.**	**Accuracy**	**Ver. Time (s)**
150	1	1	1	25	1/3	Yes	1/3	0.20
150	1	1	1	50	2/3	Yes	2/3	0.20
150	1	1	1	75	1/3	Yes	1/3	0.23
150	1	1	1	80	1/3	Yes	1/3	0.20
150	1	1	1	100	0/3	Yes	0/3	0.21
150	1	2	1	25	1/3	Yes	1/3	0.37
150	1	2	1	50	2/3	Yes	2/3	0.25
150	1	2	1	75	1/3	Yes	1/3	0.25
150	1	2	1	80	3/3	Yes	3/3	0.25
150	1	2	1	100	1/3	Yes	1/3	0.25
150	1	3	1	25	3/3	Yes	3/3	0.32
150	1	3	1	50	2/3	Yes	2/3	0.30
150	1	3	1	75	2/3	Yes	2/3	0.31
150	1	3	1	80	3/3	Yes	3/3	0.29
150	1	3	1	100	2/3	Yes	2/3	0.28
150	1	4	1	25	3/3	Yes	3/3	0.32
150	1	4	1	50	3/3	Yes	3/3	0.31
150	1	4	1	75	3/3	Yes	3/3	0.34
150	1	4	1	80	3/3	Yes	3/3	0.32
150	1	4	1	100	3/3	Yes	3/3	0.34

Table A.4: Raw results from testing the NGINX application for RPS = 200.

RPS	Min Pods	Max Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
200	1	1	1	25	0/3	No	3/3	0.51
200	1	1	1	50	0/3	No	3/3	0.47
200	1	1	1	75	0/3	No	3/3	0.43
200	1	1	1	80	0/3	No	3/3	0.42
200	1	1	1	100	0/3	No	3/3	0.42
200	1	2	1	25	1/3	Yes	1/3	0.72
200	1	2	1	50	0/3	Yes	0/3	0.91
200	1	2	1	75	1/3	No	2/3	0.72
200	1	2	1	80	2/3	No	1/3	0.72
200	1	2	1	100	0/3	No	3/3	0.72
200	1	3	1	25	1/3	Yes	1/3	0.78
200	1	3	1	50	2/3	Yes	2/3	0.99
200	1	3	1	75	3/3	No	0/3	0.80
200	1	3	1	80	3/3	No	0/3	0.79
200	1	3	1	100	3/3	No	0/3	0.80
200	1	4	1	25	2/3	Yes	2/3	0.89
200	1	4	1	50	3/3	Yes	3/3	1.13
200	1	4	1	75	3/3	No	0/3	0.88
200	1	4	1	80	2/3	No	1/3	0.87
200	1	4	1	100	2/3	No	1/3	0.89

Table A.5: Raw results from testing the NGINX application for RPS = 250.

RPS	**Min Pods**	**Max Pods**	**Initial Pods**	**Scale CPU**	**Prop. Meeting**	**WATERS Pred.**	**Accuracy**	**Ver. Time (s)**
250	1	1	1	25	0/3	No	3/3	0.35
250	1	1	1	50	0/3	No	3/3	0.35
250	1	1	1	75	0/3	No	3/3	0.36
250	1	1	1	80	0/3	No	3/3	0.33
250	1	1	1	100	0/3	No	3/3	0.34
250	1	2	1	25	1/3	No	2/3	0.79
250	1	2	1	50	0/3	No	3/3	0.71
250	1	2	1	75	0/3	No	3/3	0.57
250	1	2	1	80	1/3	No	2/3	0.50
250	1	2	1	100	0/3	No	3/3	0.53
250	1	3	1	25	1/3	No	2/3	0.96
250	1	3	1	50	2/3	No	1/3	0.85
250	1	3	1	75	3/3	No	0/3	0.62
250	1	3	1	80	1/3	No	2/3	0.55
250	1	3	1	100	2/3	No	1/3	0.56
250	1	4	1	25	3/3	No	0/3	0.64
250	1	4	1	50	2/3	No	1/3	0.61
250	1	4	1	75	2/3	No	1/3	0.53
250	1	4	1	80	3/3	No	0/3	0.47
250	1	4	1	100	3/3	No	0/3	0.51

Table A.6: Raw results from testing the NGINX application for RPS = 300.

RPS	Min Pods	Max Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
300	1	1	1	25	0/3	No	3/3	0.30
300	1	1	1	50	0/3	No	3/3	0.30
300	1	1	1	75	0/3	No	3/3	0.30
300	1	1	1	80	0/3	No	3/3	0.31
300	1	1	1	100	0/3	No	3/3	0.30
300	1	2	1	25	0/3	No	3/3	0.39
300	1	2	1	50	0/3	No	3/3	0.39
300	1	2	1	75	0/3	No	3/3	0.38
300	1	2	1	80	1/3	No	2/3	0.40
300	1	2	1	100	0/3	No	3/3	0.38
300	1	3	1	25	2/3	No	1/3	0.43
300	1	3	1	50	1/3	No	2/3	0.42
300	1	3	1	75	1/3	No	2/3	0.43
300	1	3	1	80	0/3	No	3/3	0.42
300	1	3	1	100	3/3	No	0/3	0.45
300	1	4	1	25	2/3	No	1/3	0.49
300	1	4	1	50	2/3	No	1/3	0.47
300	1	4	1	75	2/3	No	1/3	0.50
300	1	4	1	80	2/3	No	1/3	0.49
300	1	4	1	100	2/3	No	1/3	0.48

Table A.7: Raw results from testing the NGINX application for RPS = 400. Note that for these tests cases, the pods were *not* killed after each trial. This means that the initial number of nodes may vary per trial. In all other test cases, the pods were killed after each trial.

RPS	**Min Pods**	**Max Pods**	**Initial Pods**	**Scale CPU**	**Prop. Meeting**	**WATERS Pred.**	**Accuracy**	**Ver. Time (s)**
400	1	1	1	80	0/5	No	5/5	0.31
400	1	2	1	25	0/5	No	5/5	0.41
400	1	2	1	50	0/5	No	5/5	0.38
400	1	2	1	75	1/5	No	4/5	0.46
400	1	2	1	80	1/5	No	4/5	0.41
400	1	2	1	100	2/5	No	3/5	0.41
400	1	3	1	25	5/5	No	0/5	0.47
400	1	3	1	50	5/5	No	0/5	0.46
400	1	3	1	75	5/5	No	0/5	0.45
400	1	3	1	80	5/5	No	0/5	0.45
400	1	3	1	100	5/5	No	0/5	0.44
400	1	4	1	25	5/5	No	0/5	0.52
400	1	4	1	50	5/5	No	0/5	0.56
400	1	4	1	75	5/5	No	0/5	0.53
400	1	4	1	80	5/5	No	0/5	0.49
400	1	4	1	100	5/5	No	0/5	0.50

Table A.8: Results from testing the Node.js application, part 1.

RPS (low)	**RPS (high)**	**High Dur. (s)**	**Low Dur. (s)**	**Max. RT (ms)**	**Min. Pods**	**Max. Pods**	**Initial Pods**	**Scale CPU**	**Prop. Meeting**	**WATERS Pred.**	**Accuracy**	**Ver. Time (s)**
1	2	60	60	50	1	1	1	80	1/1	Yes	1/1	2.70
1	2	60	60	100	1	1	1	80	1/1	Yes	1/1	1.84
1	2	60	60	150	1	1	1	80	1/1	Yes	1/1	1.74
1	2	60	60	200	1	1	1	80	1/1	Yes	1/1	1.68
1	2	60	60	250	1	1	1	80	1/1	Yes	1/1	1.70
1	2	60	60	300	1	1	1	80	1/1	Yes	1/1	1.70
1	2	60	60	400	1	1	1	80	1/1	Yes	1/1	1.68
1	1	60	60	500	1	1	1	25	1/3	Yes	1/3	1.44
1	1	60	60	500	1	2	1	25	1/3	Yes	1/3	0.22
1	1	60	60	500	1	3	1	25	2/3	Yes	2/3	0.24
1	1	60	60	500	1	4	1	25	1/3	Yes	1/3	0.29
1	2	60	60	500	1	1	1	25	3/3	Yes	3/3	1.67
1	2	60	60	500	1	2	1	25	3/3	Yes	3/3	0.23
1	2	60	60	500	1	3	1	25	2/3	Yes	2/3	0.28
1	2	60	60	500	1	4	1	25	3/3	Yes	3/3	0.34
1	3	60	60	500	1	1	1	25	0/3	No	3/3	0.23
1	3	60	60	500	1	2	1	25	0/3	No	3/3	0.23
1	3	60	60	500	1	3	1	25	1/3	No	2/3	0.29
1	3	60	60	500	1	4	1	25	3/3	No	0/3	0.33
1	4	60	60	500	1	1	1	25	0/3	No	3/3	0.23
1	4	60	60	500	1	2	1	25	0/3	No	3/3	0.24
1	4	60	60	500	1	3	1	25	0/3	No	3/3	0.29
1	4	60	60	500	1	4	1	25	0/3	No	3/3	0.36
1	5	60	60	500	1	1	1	25	0/3	No	3/3	0.22
1	5	60	60	500	1	2	1	25	0/3	No	3/3	0.26
1	5	60	60	500	1	3	1	25	0/3	No	3/3	0.30
1	5	60	60	500	1	4	1	25	1/3	No	2/3	0.37
1	1	60	60	500	1	1	1	50	1/3	Yes	1/3	1.44
1	1	60	60	500	1	2	1	50	1/3	Yes	1/3	0.23
1	1	60	60	500	1	3	1	50	0/3	Yes	0/3	0.24
1	1	60	60	500	1	4	1	50	0/3	Yes	0/3	0.29
1	2	60	60	500	1	1	1	50	3/3	Yes	3/3	1.66
1	2	60	60	500	1	2	1	50	3/3	Yes	3/3	0.24
1	2	60	60	500	1	3	1	50	3/3	Yes	3/3	0.27
1	2	60	60	500	1	4	1	50	2/3	Yes	2/3	0.34
1	3	60	60	500	1	1	1	50	0/3	No	3/3	0.25
1	3	60	60	500	1	2	1	50	0/3	No	3/3	0.23
1	3	60	60	500	1	3	1	50	1/3	No	2/3	0.33
1	3	60	60	500	1	4	1	50	0/3	No	3/3	0.33

Table A.9: Results from testing the Node.js application, part 2.

RPS (low)	RPS (high)	High Dur. (s)	Low Dur. (s)	Max. RT (ms)	Min. Pods	Max. Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
1	4	60	60	500	1	1	1	50	0/3	No	3/3	0.24
1	4	60	60	500	1	2	1	50	0/3	No	3/3	0.24
1	4	60	60	500	1	3	1	50	0/3	No	3/3	0.30
1	4	60	60	500	1	4	1	50	0/3	No	3/3	0.37
1	5	60	60	500	1	1	1	50	0/3	No	3/3	0.23
1	5	60	60	500	1	2	1	50	0/3	No	3/3	0.27
1	5	60	60	500	1	3	1	50	0/3	No	3/3	0.30
1	5	60	60	500	1	4	1	50	0/3	No	3/3	0.38
1	1	60	60	500	1	1	1	75	2/3	Yes	2/3	1.43
1	1	60	60	500	1	2	1	75	2/3	Yes	2/3	0.22
1	1	60	60	500	1	3	1	75	1/3	Yes	1/3	0.25
1	1	60	60	500	1	4	1	75	1/3	Yes	1/3	0.28
1	2	60	60	500	1	1	1	75	3/3	Yes	3/3	1.68
1	2	60	60	500	1	2	1	75	3/3	Yes	3/3	0.23
1	2	60	60	500	1	3	1	75	3/3	Yes	3/3	0.28
1	2	60	60	500	1	4	1	75	3/3	Yes	3/3	0.34
1	3	60	60	500	1	1	1	75	0/3	No	3/3	0.24
1	3	60	60	500	1	2	1	75	0/3	No	3/3	0.25
1	3	60	60	500	1	3	1	75	1/3	No	2/3	0.28
1	3	60	60	500	1	4	1	75	2/3	No	1/3	0.35
1	4	60	60	500	1	1	1	75	0/3	No	3/3	0.24
1	4	60	60	500	1	2	1	75	0/3	No	3/3	0.24
1	4	60	60	500	1	3	1	75	0/3	No	3/3	0.30
1	4	60	60	500	1	4	1	75	1/3	No	2/3	0.36
1	5	60	60	500	1	1	1	75	0/3	No	3/3	0.22
1	5	60	60	500	1	2	1	75	0/3	No	3/3	0.24
1	5	60	60	500	1	3	1	75	0/3	No	3/3	0.30
1	5	60	60	500	1	4	1	75	0/3	No	3/3	0.36
1	2	60	60	500	1	1	1	80	1/1	Yes	1/1	1.66
1	1	60	60	500	1	1	1	100	0/3	Yes	0/3	1.45
1	1	60	60	500	1	2	1	100	1/3	Yes	1/3	0.22
1	1	60	60	500	1	3	1	100	1/3	Yes	1/3	0.24
1	1	60	60	500	1	4	1	100	1/3	Yes	1/3	0.28
1	2	60	60	500	1	1	1	100	3/3	Yes	3/3	1.72
1	2	60	60	500	1	2	1	100	3/3	Yes	3/3	0.23
1	2	60	60	500	1	3	1	100	1/3	Yes	1/3	0.27
1	2	60	60	500	1	4	1	100	3/3	Yes	3/3	0.33
1	3	60	60	500	1	1	1	100	0/3	No	3/3	0.23
1	3	60	60	500	1	2	1	100	0/3	No	3/3	0.24

Table A.10: Results from testing the Node.js application, part 3.

RPS (low)	RPS (high)	High Dur. (s)	Low Dur. (s)	Max. RT (ms)	Min. Pods	Max. Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
1	3	60	60	500	1	3	1	100	0/3	No	3/3	0.29
1	3	60	60	500	1	4	1	100	1/3	No	2/3	0.34
1	4	60	60	500	1	1	1	100	0/3	No	3/3	0.23
1	4	60	60	500	1	2	1	100	0/3	No	3/3	0.26
1	4	60	60	500	1	3	1	100	1/3	No	2/3	0.31
1	4	60	60	500	1	4	1	100	0/3	No	3/3	0.37
1	5	60	60	500	1	1	1	100	0/3	No	3/3	0.21
1	5	60	60	500	1	2	1	100	0/3	No	3/3	0.24
1	5	60	60	500	1	3	1	100	0/3	No	3/3	0.33
1	5	60	60	500	1	4	1	100	0/3	No	3/3	0.38
1	2	60	60	600	1	1	1	80	0/1	No	1/1	0.23
1	2	60	60	600	1	2	1	80	1/1	No	0/1	0.24
1	2	60	60	600	1	3	1	80	1/1	No	0/1	0.27
1	2	60	60	600	1	4	1	80	1/1	No	0/1	0.33
1	2	60	60	700	1	1	1	80	0/1	No	1/1	0.23
1	2	60	60	700	1	2	1	80	0/1	No	1/1	0.25
1	2	60	60	700	1	3	1	80	0/1	No	1/1	0.26
1	2	60	60	700	1	4	1	80	0/1	No	1/1	0.32
1	2	60	60	800	1	1	1	80	0/1	No	1/1	0.21
1	2	60	60	900	1	1	1	80	0/1	No	1/1	0.21
1	1	60	60	1000	1	1	1	25	0/3	Yes	0/3	1.45
1	1	60	60	1000	1	2	1	25	0/3	Yes	0/3	0.21
1	1	60	60	1000	1	3	1	25	0/3	Yes	0/3	0.23
1	1	60	60	1000	1	4	1	25	2/3	Yes	2/3	0.27
1	2	60	60	1000	1	1	1	25	0/3	No	3/3	0.21
1	2	60	60	1000	1	2	1	25	0/3	No	3/3	0.23
1	2	60	60	1000	1	3	1	25	0/3	No	3/3	0.26
1	2	60	60	1000	1	4	1	25	0/3	No	3/3	0.29
1	3	60	60	1000	1	1	1	25	0/3	No	3/3	0.20
1	3	60	60	1000	1	2	1	25	0/3	No	3/3	0.22
1	3	60	60	1000	1	3	1	25	0/3	No	3/3	0.28
1	3	60	60	1000	1	4	1	25	0/3	No	3/3	0.40
1	4	60	60	1000	1	1	1	25	0/3	No	3/3	0.21
1	4	60	60	1000	1	2	1	25	0/3	No	3/3	0.23
1	4	60	60	1000	1	3	1	25	0/3	No	3/3	0.27
1	4	60	60	1000	1	4	1	25	0/3	No	3/3	0.32
1	5	60	60	1000	1	1	1	25	0/3	No	3/3	0.20
1	5	60	60	1000	1	2	1	25	0/3	No	3/3	0.23
1	5	60	60	1000	1	3	1	25	0/3	No	3/3	0.41

Table A.11: Results from testing the Node.js application, part 4.

RPS (low)	RPS (high)	High Dur. (s)	Low Dur. (s)	Max. RT (ms)	Min. Pods	Max. Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
1	5	60	60	1000	1	4	1	25	0/3	No	3/3	0.42
1	1	60	60	1000	1	1	1	50	0/3	Yes	0/3	1.44
1	1	60	60	1000	1	2	1	50	0/3	Yes	0/3	0.21
1	1	60	60	1000	1	3	1	50	2/3	Yes	2/3	0.23
1	1	60	60	1000	1	4	1	50	1/3	Yes	1/3	0.25
1	2	60	60	1000	1	1	1	50	0/3	No	3/3	0.22
1	2	60	60	1000	1	2	1	50	0/3	No	3/3	0.22
1	2	60	60	1000	1	3	1	50	0/3	No	3/3	0.25
1	2	60	60	1000	1	4	1	50	0/3	No	3/3	0.30
1	3	60	60	1000	1	1	1	50	0/3	No	3/3	0.21
1	3	60	60	1000	1	2	1	50	0/3	No	3/3	0.25
1	3	60	60	1000	1	3	1	50	0/3	No	3/3	0.27
1	3	60	60	1000	1	4	1	50	0/3	No	3/3	0.32
1	4	60	60	1000	1	1	1	50	0/3	No	3/3	0.21
1	4	60	60	1000	1	2	1	50	0/3	No	3/3	0.22
1	4	60	60	1000	1	3	1	50	0/3	No	3/3	0.31
1	4	60	60	1000	1	4	1	50	0/3	No	3/3	0.32
1	5	60	60	1000	1	1	1	50	0/3	No	3/3	0.20
1	5	60	60	1000	1	2	1	50	0/3	No	3/3	0.24
1	5	60	60	1000	1	3	1	50	0/3	No	3/3	0.27
1	5	60	60	1000	1	4	1	50	0/3	No	3/3	0.36
1	1	60	60	1000	1	1	1	75	0/3	Yes	0/3	1.43
1	1	60	60	1000	1	2	1	75	1/3	Yes	1/3	0.21
1	1	60	60	1000	1	3	1	75	0/3	Yes	0/3	0.25
1	1	60	60	1000	1	4	1	75	0/3	Yes	0/3	0.26
1	2	60	60	1000	1	1	1	75	0/3	No	3/3	0.21
1	2	60	60	1000	1	2	1	75	0/3	No	3/3	0.22
1	2	60	60	1000	1	3	1	75	0/3	No	3/3	0.26
1	2	60	60	1000	1	4	1	75	1/3	No	2/3	0.28
1	3	60	60	1000	1	1	1	75	0/3	No	3/3	0.22
1	3	60	60	1000	1	2	1	75	0/3	No	3/3	0.23
1	3	60	60	1000	1	3	1	75	0/3	No	3/3	0.27
1	3	60	60	1000	1	4	1	75	0/3	No	3/3	0.31
1	4	60	60	1000	1	1	1	75	0/3	No	3/3	0.21
1	4	60	60	1000	1	2	1	75	0/3	No	3/3	0.23
1	4	60	60	1000	1	3	1	75	0/3	No	3/3	0.44
1	4	60	60	1000	1	4	1	75	0/3	No	3/3	0.33
1	5	60	60	1000	1	1	1	75	0/3	No	3/3	0.20
1	5	60	60	1000	1	2	1	75	0/3	No	3/3	0.23

Table A.12: Results from testing the Node.js application, part 5.

RPS (low)	RPS (high)	High Dur. (s)	Low Dur. (s)	Max. RT (ms)	Min. Pods	Max. Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
1	5	60	60	1000	1	3	1	75	0/3	No	3/3	0.27
1	5	60	60	1000	1	4	1	75	0/3	No	3/3	0.34
1	1	60	60	1000	1	1	1	80	0/1	Yes	0/1	1.43
1	1	60	60	1000	1	2	1	80	1/1	Yes	1/1	0.21
1	1	60	60	1000	1	3	1	80	0/1	Yes	0/1	0.24
1	1	60	60	1000	1	4	2	80	0/1	Yes	0/1	0.25
1	1	60	60	1000	1	4	3	80	0/1	Yes	0/1	0.26
1	1	60	60	1000	1	4	4	80	1/1	Yes	1/1	0.26
1	1	60	60	1000	1	4	1	80	0/1	Yes	0/1	0.25
1	1	60	60	1000	2	4	2	80	0/1	Yes	0/1	0.23
1	1	60	60	1000	3	4	3	80	0/1	Yes	0/1	2.98
1	1	60	60	1000	4	4	4	80	0/1	Yes	0/1	11.03
1	2	60	60	1000	1	1	1	80	0/1	No	1/1	0.21
1	2	60	60	1000	1	2	1	80	0/1	No	1/1	0.21
1	2	60	60	1000	1	3	1	80	1/1	No	0/1	0.26
1	2	60	60	1000	1	4	2	80	0/1	No	1/1	0.31
1	2	60	60	1000	1	4	3	80	0/1	No	1/1	0.30
1	2	60	60	1000	1	4	4	80	1/1	No	0/1	0.29
1	2	60	60	1000	1	4	1	80	0/1	No	1/1	0.31
1	2	60	60	1000	2	4	2	80	0/1	Yes	0/1	0.26
1	2	60	60	1000	3	4	3	80	0/1	Yes	0/1	3.92
1	2	60	60	1000	4	4	4	80	0/1	Yes	0/1	4.73
1	3	60	60	1000	1	1	1	80	0/1	No	1/1	0.21
1	3	60	60	1000	1	2	1	80	0/1	No	1/1	0.22
1	3	60	60	1000	1	3	1	80	0/1	No	1/1	0.28
1	3	60	60	1000	1	4	2	80	0/1	No	1/1	0.31
1	3	60	60	1000	1	4	3	80	0/1	No	1/1	0.32
1	3	60	60	1000	1	4	4	80	0/1	No	1/1	0.31
1	3	60	60	1000	1	4	1	80	0/1	No	1/1	0.36
1	3	60	60	1000	2	4	2	80	0/1	No	1/1	0.29
1	3	60	60	1000	3	4	3	80	0/1	Yes	0/1	2.25
1	3	60	60	1000	4	4	4	80	0/1	Yes	0/1	4.33
1	4	60	60	1000	1	1	1	80	0/1	No	1/1	0.24
1	4	60	60	1000	1	2	1	80	0/1	No	1/1	0.22
1	4	60	60	1000	1	3	1	80	0/1	No	1/1	0.27
1	4	60	60	1000	1	4	2	80	0/1	No	1/1	0.32
1	4	60	60	1000	1	4	3	80	0/1	No	1/1	0.32
1	4	60	60	1000	1	4	4	80	0/1	No	1/1	0.32
1	4	60	60	1000	1	4	1	80	0/1	No	1/1	0.33

Table A.13: Results from testing the Node.js application, part 6.

RPS (low)	RPS (high)	High Dur. (s)	Low Dur. (s)	Max. RT (ms)	Min. Pods	Max. Pods	Initial Pods	Scale CPU	Prop. Meeting	WATERS Pred.	Accuracy	Ver. Time (s)
1	4	60	60	1000	2	4	2	80	0/1	No	1/1	0.29
1	4	60	60	1000	3	4	3	80	0/1	No	1/1	0.41
1	4	60	60	1000	4	4	4	80	0/1	Yes	0/1	3.91
1	5	60	60	1000	1	1	1	80	0/1	No	1/1	0.19
1	5	60	60	1000	1	2	1	80	0/1	No	1/1	0.22
1	5	60	60	1000	1	3	1	80	0/1	No	1/1	0.27
1	5	60	60	1000	1	4	2	80	0/1	No	1/1	0.35
1	5	60	60	1000	1	4	3	80	0/1	No	1/1	0.33
1	5	60	60	1000	1	4	4	80	0/1	No	1/1	0.33
1	5	60	60	1000	1	4	1	80	0/1	No	1/1	0.34
1	5	60	60	1000	2	4	2	80	0/1	No	1/1	0.30
1	5	60	60	1000	3	4	3	80	0/1	No	1/1	0.34
1	5	60	60	1000	4	4	4	80	0/1	No	1/1	0.71
1	1	60	60	1000	1	1	1	100	1/3	Yes	1/3	1.44
1	1	60	60	1000	1	2	1	100	0/3	Yes	0/3	0.28
1	1	60	60	1000	1	3	1	100	1/3	Yes	1/3	0.24
1	1	60	60	1000	1	4	1	100	2/3	Yes	2/3	0.25
1	2	60	60	1000	1	1	1	100	0/3	No	3/3	0.22
1	2	60	60	1000	1	2	1	100	0/3	No	3/3	0.24
1	2	60	60	1000	1	3	1	100	0/3	No	3/3	0.26
1	2	60	60	1000	1	4	1	100	0/3	No	3/3	0.32
1	3	60	60	1000	1	1	1	100	0/3	No	3/3	0.21
1	3	60	60	1000	1	2	1	100	0/3	No	3/3	0.22
1	3	60	60	1000	1	3	1	100	0/3	No	3/3	0.27
1	3	60	60	1000	1	4	1	100	0/3	No	3/3	0.31
1	4	60	60	1000	1	1	1	100	0/3	No	3/3	0.20
1	4	60	60	1000	1	2	1	100	0/3	No	3/3	0.23
1	4	60	60	1000	1	3	1	100	0/3	No	3/3	0.27
1	4	60	60	1000	1	4	1	100	0/3	No	3/3	0.33
1	5	60	60	1000	1	1	1	100	0/3	No	3/3	0.19
1	5	60	60	1000	1	2	1	100	0/3	No	3/3	0.24
1	5	60	60	1000	1	3	1	100	0/3	No	3/3	0.27
1	5	60	60	1000	1	4	1	100	0/3	No	3/3	0.34

References

[1] M. Ahmed. (2018) Kubernetes autoscaling 101: Cluster autoscaler, horizontal pod autoscaler, and vertical pod autoscaler. Medium. [Online]. Available: https://medium.com/magalix/kubernetes-autoscaling-101-cluster-autoscaler-horizontal-pod-autoscaler-and-vertical-pod-2a441d9ad231

[2] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica — an integrated environment for verification, synthesis and simulation of discrete event systems," in *2006 8th International Workshop on Discrete Event Systems*. IEEE, 2006, pp. 384–385.

[3] K. Åkesson, M. Fabian, and H. Flordal, "Formal synthesis of control functions with Supremica," Department of Signals and Systems, Chalmers University of Technology, Tech. Rep., 2007.

[4] Q. Alam, S. U. R. Malik, A. Akhunzada, K. R. Choo, S. Tabbasum, and M. Alam, "A cross tenant access control (CTAC) model for cloud computing: Formal specification and verification," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1259–1268, June 2017.

[5] A. Alston. (2018) SLIs and error budgets: What these terms mean and how they apply to your platform monitoring strategy. Pivotal. [Online]. Available: https://content.pivotal.io/blog/slis-and-error-budgets-what-these-terms-mean-and-how-they-apply-to-your-platform-monitoring-strategy

[6] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: modeling techniques and their applications," *Journal of Internet Services and Applications*, vol. 5, no. 1, p. 11, 2014.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[8] T. K. Authors. (2019) What is Kubernetes. Kubernetes.io. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes

[9] N. Basset, M. Kwiatkowska, and C. Wiltsche, "Compositional controller synthesis for stochastic games," in *CONCUR 2014 – Concurrency Theory*, P. Baldan and D. Gorla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 173–187.

[10] K. Bernsmed, M. G. Jaatun, and A. Undheim, "Security in service level agreements for cloud computing," in *CLOSER 2011-Proceedings of the 1st International Conference on Cloud Computing and Services Science*, 2011.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media, Inc., 2016.

[12] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.995

[13] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems.* Springer Science & Business Media, 2009.

[14] W. Chareonsuk and W. Vatanawood, “Formal verification of cloud orchestration design with TOSCA and BPEL,” in *2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON).* Chiang Mai, Thailand: IEEE, 2016, pp. 1–5.

[15] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt, “Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds,” in *2015 IEEE 23rd international symposium on modeling, analysis, and simulation of computer and telecommunication systems.* Atlanta, GA, USA: IEEE, 2015, pp. 164–173.

[16] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai, “SLA decomposition: Translating service level objectives to system level thresholds,” in *Proceedings of the Fourth International Conference on Autonomic Computing*, ser. ICAC ’07. USA: IEEE Computer Society, 2007, p. 3. [Online]. Available: https://doi.org/10.1109/ICAC.2007.36

[17] D. M. Chess and J. O. Kephart, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, jan 2003.

[18] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking.* MIT press, 2018.

[19] “Monitoring in the Kubernetes era,” https://www.datadoghq.com/blog/monitoring-kubernetes-era/, Datadog, accessed: 2019-04-16.

[20] X. Etchevers, T. Coupaye, F. Boyer, and N. De Palma, “Self-configuration of distributed applications in the cloud,” in *2011 IEEE 4th International Conference on Cloud Computing.* IEEE, 2011, pp. 668–675.

[21] A. Evangelidis, D. Parker, and R. Bahsoon, "Performance modelling and verification of cloud-based auto-scaling policies," *Future Generation Computer Systems*, vol. 87, pp. 629–638, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17312475

[22] M. Fabian, *Discrete Event Systems—Lecture Notes*, 2004.

[23] M. Ficco, F. Palmieri, and A. Castiglione, "Modeling security requirements for cloud-based system development," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 8, pp. 2107–2124, 2015.

[24] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: continuous assurance of non-functional requirements," *Formal Aspects of Computing*, vol. 24, no. 2, pp. 163–186, 2012.

[25] P. Heidari, H. Boucheneb, and A. Shami, "A formal approach for QoS assurance in the cloud," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Vancouver, BC, Canada: IEEE, 2015, pp. 629–634.

[26] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, 2013, pp. 23–27.

[27] A. Hinze, P. Malik, and R. Malik, "Interaction design for a mobile context-aware system using discrete event modelling," in *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ser. ACSC '06. AUS: Australian Computer Society, Inc., 2006, pp. 257–266.

[28] K. Hu, L. Lei, and W.-T. Tsai, "Multi-tenant verification-as-a-service (VaaS) in a cloud," *Simulation Modelling Practice and Theory*, vol. 60, pp. 122–143, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X15001227

[29] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and cloud computing: from parallel processing to the internet of things*. Morgan Kaufmann, 2013.

[30] "What is load balancing?" https://www.ibm.com/cloud/learn/load-balancing, IBM Corp., accessed: 2019-10-31.

[31] J. Idziorek, "Discrete event simulation model for analysis of horizontal scaling in the cloud computing model," in *Proceedings of the 2010 Winter Simulation Conference*. Baltimore, MD, USA: IEEE, 2010, pp. 3004–3014.

[32] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 25–32. [Online]. Available: https://doi.org/10.1145/3297663.3310309

[33] K. Johnson and R. Calinescu, "Efficient re-resolution of SMT specifications for evolving software architectures," in *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, ser. QoSA '14. New York, NY, USA: ACM, 2014, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/2602576.2602578

[34] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. CBSE '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 33–42. [Online]. Available: https://doi.org/10.1145/2465449.2465456

[35] K. Klai and H. Ochi, "A formal approach for service composition in a cloud resources sharing context," in *2016 16th IEEE/ACM International*

Symposium on Cluster, Cloud and Grid Computing (CCGrid). Cartagena, Colombia: IEEE, 2016, pp. 458–461.

[36] C. Klein, M. Maggio, K. E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 700–711. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568227

[37] "Kubernetes scalability and performance SLIs/SLOs," https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md, Kubernetes Scalability Special Interest Group, accessed: 2019-03-21.

[38] D. D. Lamanna, J. Skene, and W. Emmerich, "SLAng: A language for defining service level agreements," in *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, ser. FTDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 100–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795675.797134

[39] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. Baik, "RINGA: Design and verification of finite state machine for self-adaptive software at runtime," *Information and Software Technology*, vol. 93, pp. 200–222, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917304792

[40] J. D. C. Little and S. C. Graves, "Little's law," in *Building Intuition: Insights From Basic Operations Management Models and Principles*, D. Chhajed and T. J. Lowe, Eds. Boston, MA: Springer US, 2008, pp. 81–100. [Online]. Available: https://doi.org/10.1007/978-0-387-73699-0_5

[41] S. Mahdavi-Hezavehi, V. H. Durelli, D. Weyns, and P. Avgeriou, "A systematic literature review on methods that handle multiple quality at-

tributes in architecture-based self-adaptive systems," *Information and Software Technology*, vol. 90, pp. 1–26, 2017.

[42] R. Malik, M. Fabian, and K. Åkesson, "Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata," *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 7000–7005, 2011.

[43] S. R. Malik, S. U. Khan, and S. K. Srinivasan, "Modeling and analysis of state-of-the-art VM-based cloud management platforms," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 50–63, Jan 2013.

[44] "How node.js single thread mechanism work? Understanding event loop in nodejs," https://codeburst.io/how-node-js-single-thread-mechanism-work-understanding-event-loop-in-nodejs-230f7440b0ea, Medium, accessed: 2019-04-26.

[45] F. Nawaz, N. K. Janjua, O. K. Hussain, F. K. Hussain, E. Chang, and M. Saberi, "Event-driven approach for predictive and proactive management of SLA violations in the cloud of things," *Future Generation Computer Systems*, vol. 84, pp. 78–97, 2018.

[46] P. Patros, S. A. MacKay, K. B. Kent, and M. Dawson, "Investigating resource interference and scaling on multitenant paas clouds," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '16. USA: IBM Corp., 2016, pp. 166–177.

[47] P. Patros, K. B. Kent, and M. Dawson, "Investigating the effect of garbage collection on service level objectives of clouds," in *2017 IEEE International Conference on Cluster Computing (CLUSTER).* Honolulu, HI, USA: IEEE, 2017, pp. 633–634.

[48] ——, "SLO request modeling, reordering and scaling," in *Proceedings of the 27th Annual International Conference on Computer Science and Soft-*

ware Engineering, ser. CASCON '17. USA: IBM Corp., 2017, pp. 180–191.

[49] ——, "Why is garbage collection causing my service level objectives to fail?" *International Journal of Cloud Computing*, vol. 7, no. 3-4, pp. 282–322, 2018.

[50] V. Podolskiy, M. Mayo, A. Koey, M. Gerndt, and P. 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)*. IEEE, 2019, pp. 72–81.

[51] L. Popova-Zeugmann, *Timed Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 139–172. [Online]. Available: https://doi.org/10.1007/978-3-642-41115-1_4

[52] F. Raimondi, J. Skene, and W. Emmerich, "Efficient online monitoring of web-service SLAs," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, pp. 170–180. [Online]. Available: https://doi.org/10.1145/1453101.1453125

[53] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 301–312. [Online]. Available: https://doi.org/10.1145/2663165.2663327

[54] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres *et al.*, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 4–1, 2009.

[55] R. J. Rodríguez and J. Campos, "On throughput approximation of resource-allocation systems by bottleneck regrowing," *IEEE Transactions on Control Systems Technology*, no. 99, pp. 1–8, 2017.

[56] M. Sköldstam, K. Åkesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables," in *2007 46th IEEE Conference on Decision and Control*. IEEE, 2007, pp. 3387–3392.

[57] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal verification approaches and standards in the cloud computing: A comprehensive and systematic review," *Computer Standards & Interfaces*, vol. 58, pp. 1–22, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0920548917303112

[58] A. Souri, A. M. Rahmani, N. J. Navimipour, and R. Rezaei, "A symbolic model checking approach in formal verification of distributed systems," *Human-centric Computing and Information Sciences*, vol. 9, no. 1, p. 4, Jan 2019. [Online]. Available: https://doi.org/10.1186/s13673-019-0165-x

[59] "ISO 9126 software quality model definition," http://www.sqa.net/iso9126.html, SQA.net, accessed: 2019-02-26.

[60] "Confidentiality, integrity, and availability (CIA triad)," https://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA/, TechTarget, accessed: 2019-03-13.

[61] M. ter Beek, A. Bucchiarone, and S. Gnesi, "A survey on service composition approaches: From industrial standards to formal methods," Istituto di Scienza e Tecnologie dell'Informazione, Pisa, Italy, Tech. Rep., 2006.

[62] M. H. ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg, "Synchronizations in team automata for groupware systems," *Computer Supported Co-operative Work (CSCW)*, vol. 12, no. 1, pp. 21–69, 2003.

[63] Z. Tong. (2014) Averages can be misleading: Try a percentile. Elastic. [Online]. Available: https://www.elastic.co/blog/averages-can-dangerous-use-percentile

[64] R. B. Uriarte, F. Tiezzi, and R. D. Nicola, "SLAC: A formal service-level-agreement language for cloud computing," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, ser. UCC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 419–426. [Online]. Available: http://dx.doi.org/10.1109/UCC.2014.53

[65] A. Vahidi, "Efficient analysis of discrete event systems," Ph.D. dissertation, Department of Signals and Systems, Chalmers University of Technology, 2004.

[66] G. Venzi. (2018) Deploying a Kubernetes cluster with Vagrant on Virtual Box. Gerald on IT. [Online]. Available: https://geraldonit.com/2018/03/26/deploying-a-kubernetes-cluster-with-vagrant-on-virtual-box/

[67] J. Vinárek, V. Ŝimko, and P. Hnětynka, "Verification of use-cases with FOAM tool in context of cloud providers," in *2015 41st euromicro conference on software engineering and advanced applications*. IEEE, 2015, pp. 151–158.

[68] W. Wang, X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong, "Application-level CPU consumption estimation: Towards performance isolation of multi-tenancy web applications," in *2012 IEEE Fifth International Conference on Cloud Computing*. Honolulu, HI, USA: IEEE, 2012, pp. 439–446.

[69] D. Weyns, "Software engineering of self-adaptive systems: an organised tour and future challenges," in *Handbook of Software Engineering*, K. C. K. Richard Taylor and S. Cha, Eds. Springer, 2017.

[70] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad, "A survey of formal methods in self-adaptive systems," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, ser. C3S2E '12. New York, NY, USA: ACM, 2012, pp. 67–79. [Online]. Available: http://doi.acm.org/10.1145/2347583.2347592

[71] Z. Yang, Z. Li, Z. Jin, and Y. Chen, "A systematic literature review of requirements modeling and analysis for self-adaptive systems," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2014, pp. 55–71.

[72] H. Yoshida, K. Ogata, and K. Futatsugi, "Formalization and verification of declarative cloud orchestration," in *Formal Methods and Software Engineering*, M. Butler, S. Conchon, and F. Zaïdi, Eds. Cham: Springer International Publishing, 2015, pp. 33–49.

[73] W. Zeng, M. Koutny, P. Watson, and V. Germanos, "Formal verification of secure information flow in cloud computing," *Journal of Information Security and Applications*, vol. 27-28, pp. 103–116, 2016, special Issues on Security and Privacy in Cloud Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2214212616300102

[74] J. Zhu, P. Patros, K. B. Kent, and M. Dawson, "Node.js scalability investigation in the cloud," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '18. USA: IBM Corp., 2018, pp. 201–212.