Control-Theoretic Autoscaling of Node.js in Kubernetes

by

Sujit Bhandari

Bachelors of Computer Engineering, Tribhuvan University, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master's in Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, Ph.D., Computer Science
	Panos Patros, Ph.D., Computer Science
Examining Board:	Michael Fleming, Ph.D., Computer Science
	David Bremner, Ph.D., Computer Science
	Shabnam Jabari, Ph.D., Geodesy and Geomatics Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

January, 2022

Abstract

Over the years, Kubernetes has emerged as the most prominent container orchestration platform and is thus used for managing containerized workloads in cloud-based environments. The use of the Kubernetes horizontal pod autoscaler (HPA) is one of the most widely used mechanisms for satisfaction of service level objectives under varying load conditions. Node.js applications are no exception to this; however, due to the event-driven, asynchronous nature of Node.js, there is a need for runtime-specific metrics for triggering autoscaling as opposed to the prevalent CPU-utilization-based mechanism. Thus, the contribution of this thesis is threefold: first, introducing a Node.js specific metric as the target setpoint value for the HPA controller, which acts as a threshold for triggering autoscaling; second, presenting the use of an external supervisory controller, based on control-theory and the model of the cluster, to adaptively change the setpoint value based on the difference between measured and desired service level metric; and finally, proposing a three-tier adaptive component for continuously monitoring and updating the system model and the controller parameters. To verify the methodology, this research compares the performance, under various load patterns, achieved by using the proposed approach with the widely used CPU-utilization-based autoscaling in Kubernetes.

Dedication

To my family: My parents, Ghanashyam, Sabitri Bhandari and my sister Suzeeta, whose love and support gave me the confidence to take on this challenge.

Acknowledgements

I would like to thank my supervisors Dr. Kenneth B. Kent, from the University of New Brunswick, and Dr. Panos Patros, from the University of Waikato, for their belief and support towards me, Stephen MacKay, from the University of New Brunswick, for his feedback and continuous effort on improving my technical writing, Michael Dawson, from RedHat Canada and Joran Siu, from IBM Canada, for their technical input. Special thanks to Maria Patrou for her invaluable guidance on all things related to research and beyond.

This research was conducted within the Centre for Advanced Studies — Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for the colleagues and facilities of CAS Atlantic in supporting our research. I would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 501197-16. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

Finally, I also would like to acknowledge that the place I called home for the last two years is the traditional, unceded territory of the Wolastoqiyik (Maliseet) and Mi'kmaq peoples.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of the microservices architecture in software engineering, usage of OS-level virtualization technologies, such as Docker, to run services has gained significant traction. These technologies allow software engineers to package the code and all its dependencies together and ensure parity between the development and production environments, allowing them to roll-out features confidently and more frequently [35]. Such containerized applications are extensively being deployed in cloud based environments, which provide a convenient, on-demand access to resources and numerous service models. However, there are challenges that come with deploying such applications in the cloud, such as secret management, automated rollouts, failure handling, etc. Kubernetes, built with Google's years of experience with Borg, their proprietary container-oriented cluster management system, and best practices from the community, provides the functionalities needed for orchestrating containerized applications in the cloud [4].

Additionally, cloud-based applications also need to satisfy certain service level objectives (SLOs), such as 95th percentile response time for a specific period of time, as part of a much broader set of agreements with its clients, called a service level agreement (SLA). One of the most practical and ubiquitous methods for ensuring

SLO satisfaction under varying load conditions is automatically scaling the number of computing instances. This is supported in Kubernetes in the form of the horizontal pod autoscaler (HPA) [2]. As the name suggests, the HPA is a mechanism where a controller manager periodically checks the average resource utilization of a group of objects, called pods, and scales the number of pods depending upon the set target value. Due to the event-driven nature of Node.js, which by default has the event loop bound to a single thread, using a CPU/memory-based autoscaling mechanism may not yield the best possible response time nor resource utilization since CPU/memory utilization can also peak due to garbage collection pauses as well as worker-threads running intensive computations off the main event loop [60]. Therefore, to scale the deployments, this research focuses on using a Node.js specific metric, the event loop lag, which describes the delay in the event loop. This work also introduces an external supervisory controller to regulate the setpoint event loop metric value set on the HPA controller based on the measured SLO, e.g., response time. It leverages state-of-the-art system identification techniques to model the cluster and design the controller to optimally change the setpoint value in the HPA controller to ensure a timely response to various kinds of requests. Furthermore, this thesis also proposes an additional component to the presented architecture, to adaptively update the parameters of the supervisory controller and verify the system model as well as the controller gain parameters continually, in the event of a model drift or adaptation goal changes, to ensure the satisfaction of the various SLOs.

The rest of this document is structured as follows: Chapter 2 presents the necessary background for this work. Chapter 3 discusses the relevant existing literature to this research. Chapter 4 describes the design process of the PID controller as well as the system modeling technique. In Chapter 5 the detailed experimental setup and the analysis of results are presented. Finally, Chapter 6 draws several conclusions and provides a roadmap for future work.

Chapter 2

Background

This chapter discusses the existing relevant technologies to this thesis, including Node.js, Cloud computing, Kubernetes, and Control theory.

In recent times, software engineering has seen a considerable shift from the traditional architectural pattern i.e., from monolithic to microservices. Microservices architectures allow software engineers to break down large applications into independently deployables units, commonly known as services, which can be built, tested and scaled independently [35]. The services interact with each other using either REST APIs [33] or using modern message brokers such as RabbitMQ [53]. Containerized microservices are primarily deployed in the cloud due to its convenience and the availability requirements cloud providers agree to satisfy. A brief discussion about cloud computing including its characteristics, service models, and deployment models is described below.

2.1 Cloud Computing

According to the National Institute of Standards and Technology, Information Technology Laboratory (NIST), cloud computing is defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable

computing resources (e.g., network, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [58].

They further elaborate the five essential characteristics, three service models, and four deployment models in cloud computing.

2.1.1 Essential Characteristics

The five essential characteristics as listed by NIST are as follows [58]:

On-demand Self-service: As the name suggests, on-demand self service involves the ability of clients to provision compute resources such as CPU, memory, and network storage as per their need without the need for interaction with the service-provider personnel.

Broad Network Access: Broad network access involves the availability of computing capabilities over the network through a standard mechanism, such as TCP/IP, that enables the use of such services by a diverse range of client devices, such as smart phones, laptops, tablets, workstations.

Resource Pooling: Resource pooling, usually done at the service provider's end, includes pooling various computing resources at their disposal to enable using a multi-tenant subscription model, such that the physical and virtual resources can be assigned dynamically to different clients, based on the demand of each. The client generally has no information about the resources' locations, but may be able to select higher levels of selection criteria, such as the infrastructure regions in various continents or within a country.

Rapid Elasticity: Rapid elasticity entails the on-demand or automatic provisioning of additional resources rapidly. The addition of resources can take the form of horizontal scaling or vertical scaling, giving a sense of unlimited resource availability to the clients.

Measured Service: Since cloud computing has a pay-as-you-use model, metering or telemetry capabilities for the resource usage is critical for billing and transparency. Resource usage such as storage, CPU, and bandwidth are continuously monitored and appropriate alerts can also be configured to prevent over-provisioning, which may lead to unexpected financial loss.

2.1.2 Service Models

The three service models for cloud computing as listed by NIST are as follows [58]:

Infrastructure as a Service (Iaas): Infrastructure as a service is a model in cloud computing that allows clients to provision resources such as computation, storage, and network bandwidth on a pay-as-you-use model. The clients are able to deploy arbitrary software or operating systems on such infrastructure as per their need. Using an on-demand IaaS platform removes the need to purchase and maintain hardware, thereby significantly reducing the cost for clients. Some popular IaaS providers include Amazon AWS, Google Cloud, and Digital Ocean.

Platform as a Service (PaaS): Platform as a service is a model in cloud computing that allows users to run and deploy applications using the programming language of their choice, provided it is supported by the service provider. In this service model, the client has no control of the underlying infrastructure and the model is designed to provide a ready-to-use execution environment for clients to run their application, with limited configurations that are supported by the platform.

Software as a Service (SaaS): Software as a service is a model in cloud computing that provides the users with applications and services to use. The user does not have control over the underlying infrastructure or OS, but has limited control over the application configuration. Such applications are typically accessed through web clients such as web browsers, mobile applications or, in some cases, other servers. Some examples of SaaS include Dropbox and Google Suite.

Serverless Computing: Serverless computing provides an even higher level of abstraction in the cloud service model. In this paradigm, software engineers are able to register functions with their cloud providers, compose the functions into full-fledged programs and declare events to trigger each function [50]. The underlying infrastructure is responsible for monitoring the events that trigger function invocation, preparing an appropriate runtime environment and executing it. One of the benefits of this paradigm is that the user is only billed for the resources used during function invocation and not hourly as in some other service models. Some popular serverless providers include AWS Lambda and Google Cloud Functions.

2.1.3 Deployment Models

The four deployment models for cloud computing as listed by NIST are as follows [58]:

Private Cloud: In this deployment model, the cloud resources are dedicated exclusively to a sole user or organization. The underlying infrastructure can be maintained by the user, both on or off their premises.

Community Cloud: In this deployment model, the cloud resources are assigned for exclusive use by a community of users having shared concerns such as security and standards compliance. The underlying infrastructure may be maintained by a third party or one or more users from the community, either on or off their premises.

Public Cloud: In this deployment model, the cloud resources are assigned to the public on demand. The underlying infrastructure is usually on the premises of the service provider such as a business, educational institution or government.

Hybrid Cloud: In this deployment model, the cloud resources are a combination of two or more of the aforementioned cloud infrastructures that are governed by a standardized technology that allows for data and application transferability.

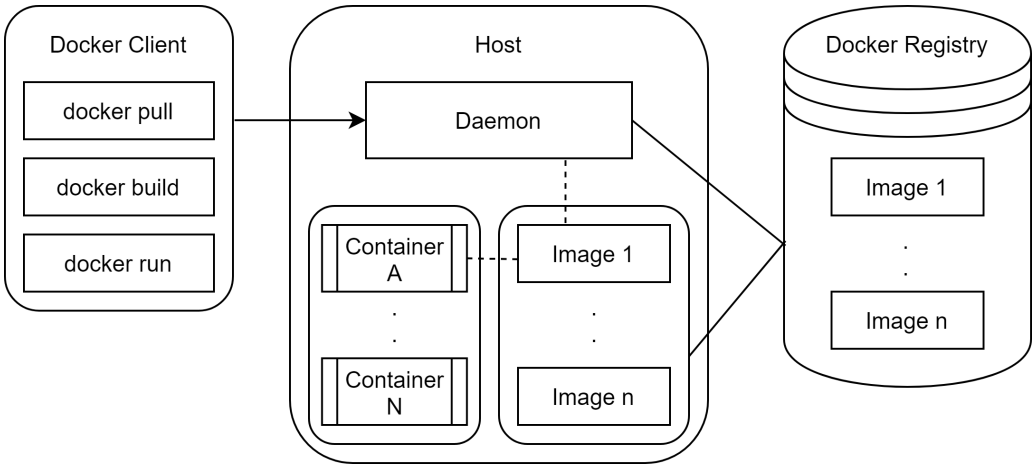


Figure 2.1: Docker Architecture [21]

The concepts described in this research will primarily be applicable to PAAS platforms. Since this research leverages Docker to bundle the application code and its dependencies together in the form of a container image, some essential concepts regarding the container runtime, Docker, is discussed below:

2.1.4 Docker

Docker is an open-source containerization platform, which allows software engineers to package their application code along with their dependencies within a container [21]. Additionally, it provides tooling and a platform to manage the container lifecycle through its APIs. As illustrated in Figure 2.1, Docker follows a client-server architecture, which consists of three components: The Docker *daemon*, the Docker *client* and the Docker *Registry*.

The Docker *daemon* and *client* communicate through *REST API* calls [21]. A Representational state transfer (REST API) is an API that enables client-server communication and conforms to the following architectural constraints: uniform interface, client-server decoupling, statelessness, availability of caching on client or server side, layered system architecture and code on demand [33]. The *daemon* listens for requests at its API endpoints and manages Docker entities such as images,

containers, storage volumes, and networks. The users interact with the Docker *client* using a Command Line Interface (CLI) to send requests to the *daemon*. A Docker *client* can also interact with more than one Docker *daemon*. Users can use a plethora of commands such as docker build, docker run, docker pull, to respectively build, run and pull a container image from a remote registry. Docker *registries* are repositories for container images. Some popular cloud *registries* include Docker Hub, Azure Container Registry, Red Hat Quay, and Google Container Registry. Furthermore, running a private container registry is also possible.

Containers remain a popular choice to bundle the code and its dependencies and run to the application. However, in production environments, a mechanism or a framework is required to ensure the resilience of the deployed application in case of container failures as well as for upgrades and rollbacks without downtime [4] during their lifecycle. Hence, orchestrators such as Kubernetes are utilized to automate tasks such as scaling and failure handling. [36]

2.1.5 Kubernetes

Kubernetes is an open-source platform for managing containerized workloads, that provides declarative configuration and automation [4]. It is a result of Google's years of experience with Borg, their proprietary container-oriented cluster management system, and best practices from the community. Kubernetes allows running containers in production resiliently by providing a framework to handle deployment patterns, scaling, failovers, and much more. Some of the features of Kubernetes are as follows:

Self Healing: It is one of the most crucial features of Kubernetes. It periodically sends health-check probes to the running containers and restarts, replaces or kills unresponsive containers, depending upon the user specified configuration, until they are ready to serve traffic again.

Service Discovery and Load Balancing: Kubernetes can expose a container to traffic using one of the three techniques: Services, which is an abstraction over one of its most basic objects called Pods, DNS names or using its own IP address. Furthermore, it is also capable of distributing load across containers for a stable deployment.

Storage Orchestration: Kubernetes allows automatically mounting a storage system such as local storage, and public-cloud-provider storage. Similar to Docker, storage volumes can be used to persist data in case of Pod failures.

Automated Rollouts and Rollbacks: Kubernetes enables defining the desired state of a deployment, including the minimum number of running replicas. Techniques such as rolling upgrades ensure automatic creation of new containers or rollbacks to replace/remove old containers and adapt their resources to new containers.

Secrets and Configuration Management: Sensitive information such as passwords and SSH keys can also be stored and managed by Kubernetes. For example, information such as usernames and passwords for a database can be stored in an encoded format and can be plugged into the deployment as environment variables, without the need for them to be stored in plain-text format. Similarly, a ConfigMap can be used to store non-sensitive data as key-value pairs. These values can also be accessed to configure a container inside a Pod via environment variables.

2.1.5.1 Kubernetes Architecture

A Kubernetes cluster typically consists of a control plane and set of worker nodes, that run containerized workloads, as shown in Figure 2.2. The control plane is responsible for making global decisions about the cluster such as Pod scheduling as well as recording and responding to cluster events such as Pod failures, Pod additions, etc. The control plane consists of several controllers that monitor the current state of several Kubernetes objects such as Pods, Deployments, Services and makes sure that

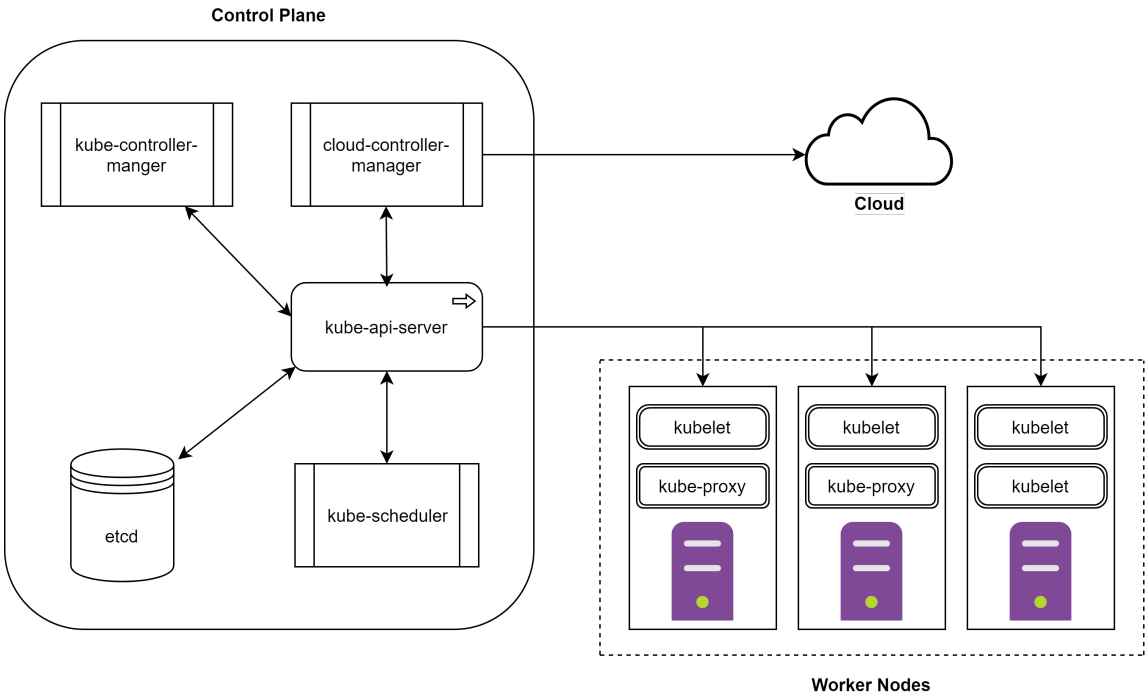


Figure 2.2: Kubernetes Components [1]

their current states match the desired state specified in their manifest [1]. Kubernetes objects are essentially persistent entities that represent the state of the cluster [22]. A brief description of these objects is presented in Table 2.1

The main components of a control plane are as follows [1]:

- **kube-api-server**: It acts as the front-end to the Kubernetes Control Plane. At its core, it exposes a REST API, which is used by the cluster administrators to create, modify or delete objects such as Pods and Services.

- **kube-controller-manager**: It is a component that runs the controller processes. Controllers run a control loop that monitors the state of an object of the cluster via the api-server and etcd to make sure that the desired state of the object is always fulfilled. It essentially works to move the current state of these objects as close as possible to the desired state. Some of the examples of controllers in existence in Kubernetes are replication controller, node controller, job controller, service account and token controller.

Objects	Description
Pod	A Pod is the most basic deployable entity in Kubernetes. It encapsulates application container/s with shared storage and network resources.
Service	A Service is an abstraction to expose an application running on a group of Pods as a network service. The concept of labels and selectors on the Pods and Service, respectively, allows a Service to direct traffic on a particular group of Pods even when new Pods are added to the group, with new IP addresses due to failures or scaling.
Deployment	A Deployment enables declarative rollouts for Pods. It allows users to describe the desired state in a manifest file and the deployment controller works towards achieving the goal at a controlled rate.
Namespaces	Namespaces enable dividing the physical cluster resources between multiple teams/users using resource quotas. It provides a scope for names of the Kubernetes objects.
Ingress	An Ingress directs external traffic to internal Services using a set of rules. Additionally, it can also be configured to load balance traffic and offer SSL/TLS termination.
Horizontal Pod Autoscaler	The Horizontal Pod Autoscaler automatically scales the number of Pods in a deployment based on the difference between observed metrics and the target metrics.

Table 2.1: Kubernetes Objects [22]

- **etcd**: It is the single source of truth for the kube-api-server. It is implemented in the form of a key-value store. All data related to the various objects in Kubernetes such as running Pods, desired number of Pods for a deployment, are stored and continuously updated in etcd.

- **kube-scheduler**: It is responsible for making suitable decisions for scheduling newly-created Pods on a set of available nodes. The decision is made based on the resource requirement and other affinity concerns described in a Pod's

specification, as well as the availability of the resources on the nodes.

- **cloud-controller-manager**: It allows the cluster administrators to link the cluster to the service provider's API and isolates the components that interact with the cloud API and the components internal to the cluster. For example, cloud-controller-manager allows plugging into the service providers' paid load balancer services.

The components of each worker node are as follows [1]:

- **kubelet**: A kubelet is an agent that runs on each of the nodes in a Kubernetes cluster. Its primary job is taking the Pod specification through the kube-api-server and creating a Pod that runs container/s with the specification. Furthermore, it is also responsible for periodically checking the status and availability of the Pods using techniques such as liveness and readiness probes.

- **kube-proxy**: A kube-proxy is another agent that runs on each node of the cluster and is responsible for implementing the service concept in Kubernetes. It maintains a set of network rules and forwards external traffic to the Pods based on these rules.

2.1.6 Client and Service Provider Agreements

In order for a cloud service provider to entice its clients to use their services, it has to meet certain client expectations and provide guarantees about the reliability of its services. Therefore, the clients and the service providers agree on a legally binding document that details a set of mutually acceptable performance goals and, in some cases, financial penalties for not fulfilling the goals, called a Service Level Agreement (SLA). The expected performance and Quality of Service (QoS) are specified by Service Level Objectives (SLOs), which describe certain thresholds the provider

is expected to maintain on a variety of Service Level Indicators (SLIs) [37]. For example, in web services, 95th percentile response time for a period of time could be an SLI, whereas the stipulation that the aforementioned SLI should be under 100ms could be an SLO.

2.1.7 Autoscaling as a Means for Satisfaction of SLOs

One of the most crucial features of cloud computing is elastic provisioning of resources. Additional computing instances are added or removed as per the dynamic requirement of the applications. There is a plethora of existing literature that describe various autoscaling methodologies, including reactive [67, 55, 53], as well as predictive [68, 69] autoscaling.

In Kubernetes, autoscaling pods in a deployment is supported in the form of Horizontal Pod Autoscaler (HPA) or Vertical Pod Autoscaler (VPA).

2.1.7.1 Kubernetes Horizontal Pod Autoscaler

In Kubernetes, the Horizontal Pod Autoscaler (HPA) automatically scales the number of Pods in a deployment based on the difference between observed metrics and the target metrics [2]. It is implemented as a control loop, with a configurable period using a flag *–horizontal-pod-autoscaler-sync-period* in the controller manager's manifest. The controller manager periodically queries the observed metric against the target metric and adjusts the number of Pods in the deployment. The HPA fetches metrics from a series of aggregated APIs such as *metrics.k8.io*, *custom.metrics.k8.io*, and *external.metrics.k8.io*. At its core, the HPA controller calculates the desired number of replicas using the following formula:

$$DesiredReplicas = ceil\left(CurrentReplicas \times \frac{ObservedMetric}{DesiredMetric}\right) \tag{2.1}$$

Using the formula above, if the observed metric is 300 and the desired metric is 100, the number of Pods will be tripled.

2.1.7.2 Kubernetes Vertical Pod Autoscaler

Vertical pod autoscaler (VPA) enables automatically updating the CPU and memory requests and limits [14]. Unlike in the HPA, where the requests and limit are static, VPA allows dynamically updating these values for optimal CPU and memory usage. Additionally, it also maintains the ratio of the resource requests and limits specified in the initial container configuration when updating these values. The VPA consists of three components:

- **Recommender**: It provides a recommendation for the values of the resource requests based on the present and past usage.
- **Updater**: It monitors whether the pods have correct resources set and terminates the pods so that they can be re-created with updated values.
- **Admission Plugin**: It is responsible for setting new resource requests on the pods based on the recommendation or the initial configuration.

In addition to scaling the number of computing instances to satisfy service level metrics, other approaches such as self-adaptive systems, which draw inspiration from machine learning and control theory, have also been discussed as an alternative to design software systems to handle disturbances at runtime [54, 57] and to provide formal guarantees about the performance in key aspects such as rise time, settling time, and overshoot [40, 48].

2.2 Control Theory

Due to highly dynamic runtime conditions, various mechanisms are required to deploy robust systems that respond to such conditions and maintain acceptable levels of performance. Self-adaptive systems have been proposed to assist engineers with this requirement [47]. Self-adaptive systems with control-theoretic backgrounds are already being used in today's software. For example, the HPA controller as part of a control loop checks the average resource utilization of a group of Pods and scales up or down the number of Pods in a deployment [2]. Similarly, the replication controller in Kubernetes periodically checks whether the number of Pods deployed matches the minimum desired replica count and creates a new Pod in the event of a Pod failure. In a similar vein, in this research, we present the use of a control-theoretic mechanism for modeling the kubernetes cluster and synthesizing a controller for regulating input to the cluster based on the measured output (response time).

Essentially, a self-adaptive system is capable of changing its structure or behaviour at runtime, depending upon the difference in its current state and desired state. From a software engineering perspective, the Monitor-Analyze-Plan-Execute (MAPE) [52] loop can be taken as the reference architecture. In this architecture, an adaptation manager is responsible for detecting changes in the controlled system, analyzing the impact of the changes, as well as planning and executing control actions to mitigate the effects of those changes. From a control engineering standpoint, a control system and the feedback control loop consists of the controlled system (the plant), and the controller based on control theory [51], as illustrated in Figure 2.3. In a discrete time domain, $y(t)$ is the output of the plant at time instant t, and the input to the controller is the difference between the desired output $\bar{y}(t)$ and $y(t)$, represented as the error $e(t)$. The input to the plant, in response to this error, is represented as $u(t)$.

At the system analysis phase, the relationship between the control input and output

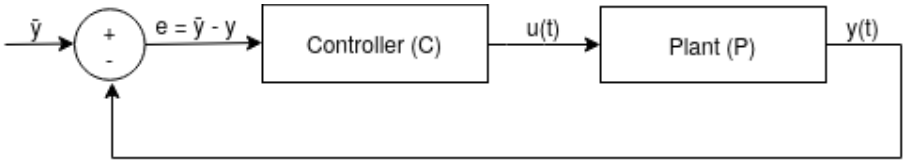


Figure 2.3: Generic Feedback Loop Architecture [48]

of the system is determined to ensure that the change in the control input, $u(t)$, correctly changes the output of the system, $y(t)$, which is also known as the directionality of the input/output relationship. In addition to changing the output of the controlled system based on the error, control theory also provides formal guarantees regarding the characteristics of the output change—such as rise time, settling time, and overshoot—after a control action has been applied. These characteristics are influenced by the design and tuning process of the controller.

With regard to the controllers, one of the most widely used controllers is the Proportional Integral Derivative (PID) controller, which consists of three terms: proportional (P), integral (I) and derivative (D), as shown in Figure 2.4. These three terms influence the magnitude and the influence of the control action $u(t)$ to the system. The weights of the P, I and D terms are determined by their gain parameters, k_p, k_i, and k_d respectively [39].

Mathematically, the control action $u(t)$, as a result of previous system outputs, current system output and the error is represented as follows:

$$u(t) = k_p e(t) + k_i \int_0^t e(t)dt + k_d \frac{de(t)}{dt} \tag{2.2}$$

The first term in Equation 2.2 indicates proportional control that responds to the tracking error with a corrective action. The magnitude of the corrective action depends on the tracking error itself as well as the proportional gain coefficient: k_p. The second term represents integral control and its output is proportional to the integral of the tracking error over time. The integral gain coefficient, k_i, influences

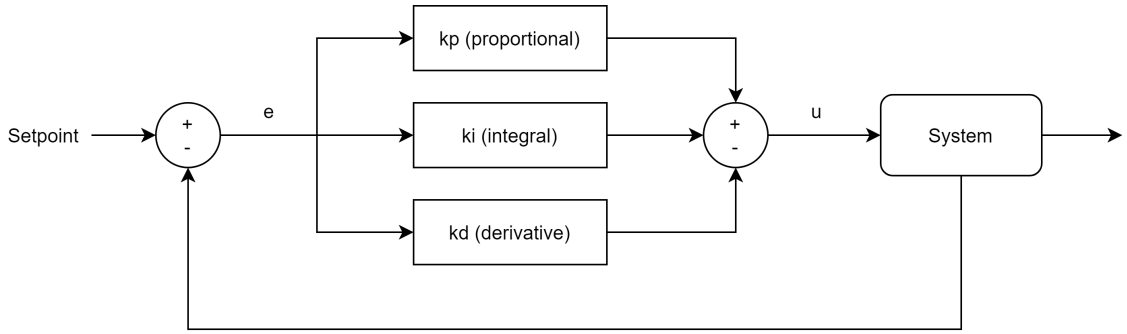


Figure 2.4: PID Controller [51]

the ability of the controller to effectively reduce cumulative errors. Finally, the third term represents derivative control. Unlike integral control, which minimizes tracking errors over time, derivative control tries to anticipate future errors by taking a derivative of the current error. Similar to proportional and integral gain coefficients, the derivative gain coefficient, k_d, also influences the magnitude of the derivative control.

2.2.0.1 Guarantees Under Uncertainty

The closed-loop system built using the controller and the system, as illustrated above in Figure 2.3, should work towards maintaining an SLO requirement, such as response time, energy consumption or throughput. From a control engineering standpoint, any such controller should provide the system with the following attributes [47]:

- **Stability**: Given an input, the system converges to a stable state that is determined by the setpoint.
- **Low Overshoot**: The system does not exceed the setpoint value after a control action is applied.
- **Low Settling Time**: The controlled system reaches the desired setpoint swiftly when a control action is applied.
- **Robustness**: A robust system continues to converge to the desired setpoint

even in the presence of disturbance, inaccuracies in measurements and variation in the system model.

These four attributes can be guaranteed analytically provided that the mathematical representation of the system is known.

2.2.1 System Identification and Controller Parameter Tuning

Since the controller gain parameters, k_p, k_i, and k_d, have the ability to influence output of the system as well as the characteristics of the output, careful consideration must be given to iteratively tune these parameters. Also, unlike in classical control theory, modeling the system using differential equations from first principles is a non-trivial task in case of dynamic systems. Hence, this research relies on modern approaches such as system identification, which relies on system response to dynamic inputs [46], to obtain a transfer function. In the same vein, tuning the controller parameters appropriately is also challenging, given the dynamic nature of the system. Therefore, the system is modeled, simulated and analyzed using Simulink and MATLAB [11] and the controller gain parameters are also tuned iteratively using the generated model and the PID tuner application in MATLAB.

2.3 MATLAB and Simulink

MATLAB, an abbreviation of "matrix laboratory", is a matrix-based language for computational mathematics developed by MathWorks Inc [26]. In addition to matrix manipulation, MATLAB's built-in graphics allow data visualization, implementation of algorithms and creation of custom user interfaces, and provide a flexible two-way integration with programming languages such as C++, C, Python, and Java.

Simulink is a graphical simulation environment for modeling and analyzing systems [25]. At its core, it provides a graphical interface for creating blocks and connectors to represent models or various other components in a system. It is tightly integrated with MATLAB, which makes it easier for users to import models, run simulations and export the results.

2.3.1 MATLAB System Identification Toolbox

The toolbox provides an application for building mathematical models of dynamic systems using the measured input-output data [5]. It takes as input the time-domain or frequency-domain data and outputs transfer functions, process models or state-space models. It is especially useful when modeling complex dynamic systems, where modeling the system using differential equations from first principles is not feasible. This study focuses only on transfer function models as they sufficiently capture the dynamics of the system and require only two parameters to get started: the number of poles and zeroes, which represent the roots of the numerator and denominator, respectively. For estimating the possible initial model orders, users can utilize a simple polynomial model structure (ARX) and subsequently use other model structures such as transfer functions with orders close to the estimated initial orders [23].

2.3.1.1 Transfer Function Models

A transfer function model represents the input-output relationship of a dynamic system by using a ratio of polynomials [24]. The two main parameters needed in a transfer function are poles and zeros. Given the input-output data of the system, the system identification toolbox only requires the number of poles and zeroes to generate the transfer function model. Furthermore, the order of the model is also equal to the order of the denominator.

In the frequency domain, for continuous-time systems, for an input $x(t)$ and output

$y(t)$, the transfer function can be represented as the ratio of a Laplace transform of output and input signals. The Laplace transform allows control designers to represent a function of time t as a function of complex frequency s. Using Laplace transform and frequency domain representation, the effect of the control input to the system can easily be calculated by multiplying the input with the transfer function.

$$H(S) = \frac{Y(S)}{U(S)} \tag{2.3}$$

Similarly, for the transfer function of discrete-time systems, it can be represented as the Z-transform of output and input signals. Similar to the case of continuous-time systems, using frequency-domain representation (Z-transform) allows control designers to represent a function of time t as a function of complex frequency z.

$$H(Z) = \frac{Y(Z)}{U(Z)} \tag{2.4}$$

2.3.2 Controller Parameter Tuning

Essentially, PID tuning uses the concept of pole placement, loop shaping or other analytical and optimizing tuning methods to meet the desired characteristics, such as rise time, settling time, and overshoot.

The position of the open-loop poles of the transfer function determine the stability of the system. In the continuous-time domain, all the open-loop poles must be on the left-hand plane and for the discrete-time domain, all the poles must lie inside the unit circle around the origin, in the complex plane. The process of pole placement is concerned with adjusting the controller gain parameters to "move" the poles in the complex plane such that it results in the desired system characteristics [51].

For systems whose transfer function is known, loop shaping is a mechanism where additional elements, such as a "lead-lag compensator", are added such that the undesired poles coincide with the introduced zero and the corresponding pole is placed

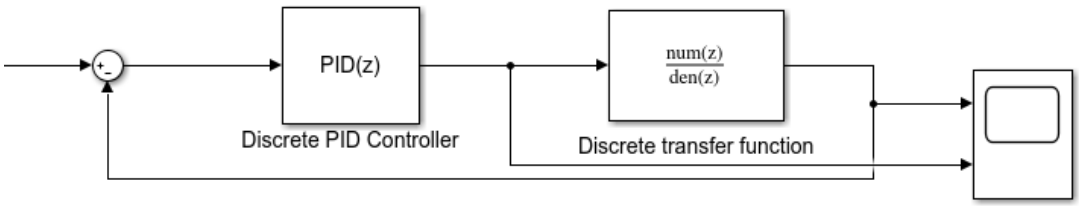


Figure 2.5: Closed loop system in Simulink

in a desired location in the complex plane [51]. Lead-lag compensators are components that introduce pairs of poles and zeroes to the transfer function, for desired characteristics. Heuristic methods such as Ziegler-Nichols and Cohen-Coon [38] do not require the system's transfer function and the parameters are derived from the observed characteristics of the step-input response of the system.

2.4 Language Runtimes

Services that are written for cloud-based environments are primarily written in high-level programming languages such as Java, JavaScript, and Python. Unlike compiled languages, such as C and C++, these high-level languages require an environment, called a runtime, to interpret and execute [61] on the host machine. In addition, high-level languages also implement an automatic memory management technique called Garbage Collection and may also be equipped with a Just-in-time (JIT) compiler, that compiles repeatedly executed code sections into optimized machine code. One such example of a language runtime is Node.js, a JavaScript runtime for server-side development. Using Node.js for the server-side script enables developers to use the same programming language for both server and client-side applications [43]. A brief introduction to the crucial concepts in Node.js is presented below.

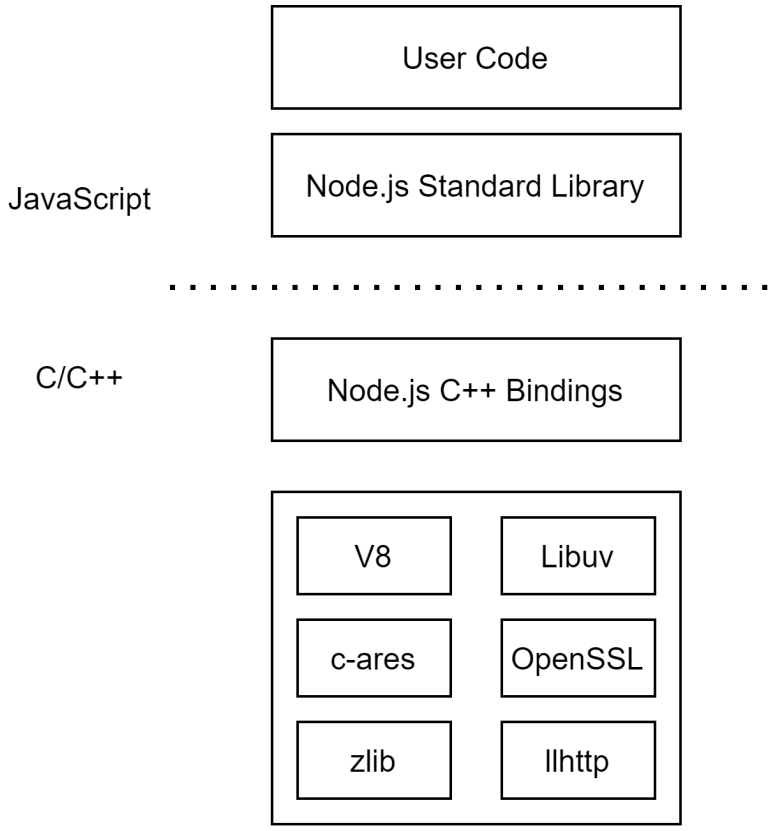


Figure 2.6: Node.js Code Architecture

2.4.1 Node.js

Node.js is an open-source JavaScript runtime environment that is built on top of Google's V8 JavaScript engine [17]. It has an event-driven architecture with non-blocking I/O operations that makes it efficient and highly scalable [42]. Internally, it uses libuv, which provides the event loop functionality, asynchronous sockets, and a thread pool for DNS resolution, file I/O, and the crypto module [16]. In addition to a JIT compiler, V8 also implements a garbage collector, and handles memory allocation, among other things. Some other dependencies of Node.js include an http parser, called llhttp [31], an asynchronous DNS library, called c-ares [8], a toolkit for TLS and SSL protocols, called OpenSSL [32], and a compression library, called zlib [10]. Figure 2.6 illustrates the code structure of Node.js. The user mainly writes code using the Node.js standard library. The C++ bindings are able to provide the

various functionalities provided by V8 and other libraries to the user.

Unlike the legacy "thread per connection" architecture, which dedicates a thread to each client connection, Node.js' event-driven architecture multiplexes multiple client connections onto a single-threaded event loop and an additional thread pool to handle the aforementioned functions [43]. As part of the event loop, the application uses an event notification mechanism, such as *epoll*, *kqueue*, or *IOCP*, to request the operating system to monitor client socket connections for events. The application can add and remove the sockets from the OS's watch list and poll the OS on the status of those connections. For instance, in Linux, it is achieved through the *epoll* group of system calls. The rest of this section will explain the concepts from a Linux perspective. Using the *epoll_wait* system call, the loop will block waiting for I/O activity on the sockets that have been added to the watch list and the associated callback will be added to an appropriate queue, to be executed eventually [15].

The event loop iteration consists of a set of phases, each with its own queue, that executes callbacks specific to that queue [18]. For example, in the timer phase, callbacks related to timer methods such as *setTimeout* and *setInterval* are executed. The lag or delay in the event loop can thus be measured using timers, as the execution of timer-related callbacks are tied to the lifecycle of the event loop [13].

As illustrated in Figure 2.7, at the beginning of each iteration, the current time is cached to reduce the number of time-related system calls. This is also useful in deciding when the timer related callbacks are due. The condition tested before entering the loop is checking whether there are any active handles or requests. For instance, this could be a TCP server handle. If there are not any, the process exits cleanly. After fulfilling the condition, the loop moves to the phase of executing timer-related callbacks. In the next phase, pending callbacks related to certain system events such as "TCP connection refused" are executed. Idle Handles and Prepare Handles callbacks run in each loop iteration before polling for I/O. In the poll phase, the

loop blocks and waits for I/O on a set of sockets for a pre-calculated period of time, determined by the closest timer, and the presence of the *setImmediate* callbacks, among others. Furthermore, the I/O related callbacks in the poll queue are also executed in this phase. In the check phase, callbacks queued using *setImmediate* are executed immediately after the poll phase. Finally, callbacks related to close handles are executed. For instance, if a handle is closed by invoking the *uv_close()* method, the corresponding callback will be executed in this phase [16].

Since the Node.js event loop is bound to a single thread, by default, any long running callbacks or cpu-intensive operations that run on the thread negatively affect the application's performance. This creates a problem as computationally intensive tasks are limited by the performance of a single core, which necessitates the use of various parallelism techniques [65]. With Worker Threads, which is a module that enables the use of threads that execute JavaScript in parallel, spawning multiple threads with their own event-loop, CPU-intensive operations can be offloaded to several other threads, thus improving performance [12].

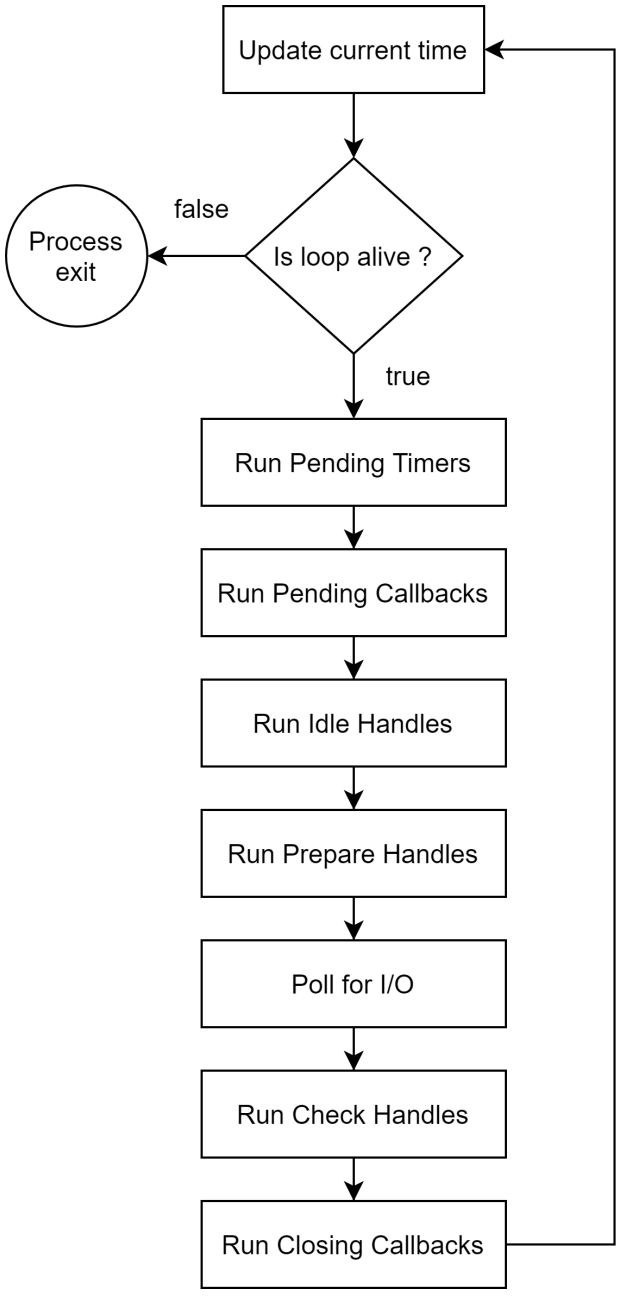


Figure 2.7: Node.js Event Loop Phases [16]

Chapter 3

Related Work

Autoscaling computing instances to ensure the satisfaction of SLOs is not new. However, autoscaling based on a runtime specific metric, with a setpoint varying mechanism based on the response time is a unique approach. This chapter presents the state-of-the-art in Node.js, various autoscaling approaches including control theoretic techniques and other procedures for the satisfaction of SLOs.

3.1 Node.js

Some of the existing literature on Node.js consists of analysis of scalability, performance engineering to improve SLI metrics, and research on the compilation process in Node.js. However, this research focuses on autoscaling as a means for satisfaction of SLOs using a Node.js event loop metric and a service level metric.

Zhu et al. present a scalability investigation of Node.js in the cloud with a benchmark suite and conduct a set of experiments including CPU intensive, I/O intensive, network intensive and memory intensive applications with both single and multiple instances of the applications running on the cloud [70]. However, unlike this work, where we propose a control-theoretic approach to scaling a Kubernetes cluster, the authors use docker swarm to deploy, manage the containers, and compare

the performance of scalability strategies such as horizontal and vertical scaling and Node.js cluster module. Patrou et al. have also performed an analysis of various parallelization techniques in Node.js under software and hardware variabilities in the system [65]. The authors present a methodology to evaluate multi-process and multi-thread techniques, in which they assess the modules in four key aspects: overhead, task execution, sharing memory and communication. However, the authors use a single host machine for their analysis unlike this thesis, where the focus is on containerized applications, orchestrated by Kubernetes.

In a similar vein, other research in performance and optimizations in Node.js includes approaches to reduce tail latency, which is a high percentile latency, such as 99.9^{th} percentile, in Node.js by introducing an event dependency graph to represent an asynchronous request in event driven systems and leverage the tail latency profile to dynamically turbo boost processors at critical phases in the application code execution [42]. Similarly, first-class timeouts have also been introduced to defend against event handler poisoning attacks in Node.js [43] and in a more recent study, the compilation performance of WebAssembly has been analyzed in Node.js and a multi-process shared compiled code cache was introduced to reduce module load times [59].

3.2 Approaches to Satisfaction of SLOs

As described in the previous chapter, one of the core features of Kubernetes is the horizontal pod autoscaler, which is used widely to ensure satisfaction of SLOs under various workloads.

Ye et al. propose a hybrid scaling strategy based on a prediction model of resource utilization to satisfy SLAs under peak load conditions [68]. They use a mixture of quick vertical scaling as well as horizontal scaling based on the time-series forecasting

of resource demand for the next interval. They compare their hybrid scaling approach to the Kubernetes horizontal pod autoscaler. However, they consider only CPU usage and no other metrics for scaling, which might not be ideal in the case of Node.js. Similarly, Song et al. demonstrate the effectiveness of the Kubernetes horizontal pod autoscaler in their API gateway system under linearly increasing load, which stabilizes at a certain value [67]. They present the use of an alternative metric, i.e., queries per second (QPS), in addition to CPU usage for the HPA target value. In both cases, the authors do not explore the use of language runtime-specific metrics like the event loop lag.

Zhao et al. also propose an algorithm to dynamically provision resources in Kubernetes (Pods) [69]. Instead of using a static threshold used by the default Kubernetes autoscaler, the authors implement a prediction module that calculates the number of pods needed based on the current workload in the cluster. For scaling down, however, they use the HPA's original strategy of triggering scale down after five minutes only. However, the authors do not disclose the type of application they are testing and also do not explore various workload patterns such as linear or seasonal, which could represent a more realistic use case.

Kumar et al. propose an autoscaling mechanism to reactively allocate resources in a cloud based environment to fulfill desired QoS metrics [55]. For a CPU under-utilization threshold of 15% and over-utilization threshold of 70%, the authors implement a methodology to dynamically provision the number of VM instances, when the measured value is not within these two limits. They compare this method with a static provisioning technique and demonstrate its effectiveness in terms of the number of SLA violations.

Dickel et al. use custom metrics, such as the number of messages per second (MSP), per active stateful connection, in web sockets, to scale IoT gateways in a Kubernetes cluster [45]. Similar to this research, the authors leverage Prometheus, which is a

time-series database, to query metrics from their application and trigger autoscaling based on the number of active stateful connections. However, this research differs in two key aspects: the use of a language-runtime-specific setpoint metric, i.e., event loop lag, and the use of an additional controller to adaptively change the setpoint value based on measured and desired response time.

Khaleq et al. implement an agnostic autoscaling algorithm to scale microservices in Kubernetes, based on alternate metrics such as memory usage and number of undelivered messages in a queue, and compare it with the default implementation in Kubernetes, i.e., CPU-utilization based [53]. Similar to this research, the authors introduce an additional controller to dynamically adjust parameters in the HPA. However, unlike in this research, the authors do not leverage control-theoretic concepts to model the cluster and design a PID controller to guarantee the overall characteristics of the system after the control action, such as rise time, overshoot, and settling time.

Podolskiy et al. present an approach to vertical scaling of containers in a resource saturated cloud and investigate the satisfaction of SLOs in such environments [66]. In contrast to this research, where we mainly focus on adaptively changing the threshold value for scaling the number of pods in a Kubernetes deployment, the authors focus on resource allocation and maintaining SLOs in co-located applications based on a prediction model generated from the relationship between SLIs, workload and the resource limits.

3.2.1 Patented Literature

Norris et al., in their patent regarding the event loop utilization metric, also advocate for using an event-loop-specific metric [60]. The inventors present the use of an alternate metric for autoscaling as opposed to the more common metrics used, such as CPU and memory usage. Event loop utilization measures the degree of event

loop utilization, i.e., the amount of time the event loop is executing callbacks of related events and not idle, waiting for events. Similarly, Oracle Inc., in their patent regarding automatically scaling a cluster based on metrics being monitored, also propose scaling nodes in a cluster by comparing measured values with a set of user configured threshold values [41]. However, unlike our approach, where we propose adaptively changing the setpoint value for autoscaling in Kubernetes based on the difference between desired and measured SLO, the authors in both patent documents suggest using a static threshold range to trigger autoscaling.

The existing autoscaling approaches mainly focus on reactive autoscaling based on either CPU metrics or, in some cases, other metrics such as queries per second, based on a static threshold value. While some authors do implement a controller to change the threshold values based on resource availability and workload, it lacks the guarantees control theory provides regarding its attributes such as rise time, settling time and overshoot.

3.2.2 Control-theoretic Techniques

Control-theoretic approaches to satisfaction of SLOs include a variety of methodologies, such as brownouts and PID controller-based systems, among others. The use cases for the following systems include satisfying certain service level metrics, ensuring proper resource utilization, etc.

Barna et al. present the use of a portable PID controller, which can be used in both public as well as private clouds and advocate for a robust control theory-based autoscaler, as opposed to the commonly used threshold-based reactive autoscaler [39]. However, they narrow their scope to CPU-heavy applications and model the number of instances in terms of the degree of CPU utilization of a tier and as posited in the previous chapters, CPU-utilization-based autoscaling may not be suitable for Node.js applications. Brownouts [54, 57], which utilize the concept of optional computation,

are applicable when there are optional components in an application. This mechanism uses the principles of control theory to synthesize a controller that determines whether a request should be served with optional components, based on the current measurements from the application. They focus on web applications and in particular use as an example the advertisements in web pages as the optional components, which could be expendable when there is a surge of requests to the application. Since the application used in this research does not have optional components, following the author's model was not an option.

Dickel, in his master's thesis, introduces a configurable PID controller in the HPA controller codebase itself in Kubernetes to replace the existing autoscaling decision-making algorithm [44]. The author also leverages MATLAB and Simulink to approximate the system's transfer function and uses auto-tuning techniques for tuning the controller. Since the author focuses on changing the Kubernetes' algorithm, the rest of their methodology differs from our use case, where we plan to introduce a controller to regulate the setpoint value of the event loop metric in the HPA controller.

Filieri et al. describe a five-step control design process for the design and analysis of a controller module with formally verifiable attributes [48]. The authors model their system using a transfer function, synthesize a controller using loop-shaping and formally verify their closed-loop system for a video encoding software. This research leverages the author's strategy for setting goals for such systems, identifying correct process and control variables to match the goals and determining an appropriate model structure for our system.

Burroughs et al. evaluate the performance of various methodologies for modeling and verifying autoscaling in Kubernetes using the Horizontal Pod Autoscaler (HPA) [40]. The authors test the performance of three modeling techniques: discrete, probabilistic and control-theoretic. In the control theoretic analysis, the authors leverage MATLAB and Simulink to model the behaviour of Kubernetes and parameterize the

controller, which is also the methodology used in this research. However, unlike this research, the authors have not detailed the results of various workload patterns and test cases for evaluating PID the controller's performance.

3.3 Other Methodologies for SLO Satisfaction

In addition to autoscaling computing instances, there is also existing literature that focuses on the impact of various components of a language runtime, such as a garbage collector on the SLOs of cloud based applications, and presents innovative methodologies to fulfill the SLO requirements.

Patros et al. investigate the effect of garbage collection on the SLOs of cloud-based applications [62]. The authors present CloudGC, which is an application that stresses the garbage collection component of the IBM J9 Java Runtime, and compare the performance of four of its garbage collection policies: Gencon, Balanced, Optavgpause and Optthroughput. Their evaluation of performance interference due to CloudGC shows response time SLOs are vulnerable to abrupt garbage collection interference [63]. In another associated work, Patros et al. propose and validate a theoretical model of cloud-based applications to calculate the ideal number of computing instances required to maintain the satisfaction of SLOs. Furthermore, the authors also illustrate the implementation of an SLO-aware request re-ordering methodology that re-orders the request in their novel SLO task queue, based on the request's previous execution time, instead of executing the request handlers on a FCFS order [64]. However, our research is much narrower in scope than the authors' work as we do not attempt to calculate the ideal number of instances or re-order requests execution. The number of instances is determined by the HPA algorithm (Equation 2.1), and requests are executed by the Node.js event loop, in the order their handlers are placed in the queue.

Chapter 4

Methodology

This chapter discusses the research goals, configuration of an on-premises multi-node Kubernetes cluster, and the control design process involving system modeling and controller gain parameter estimation. Furthermore, it also presents the architecture of the custom autoscaling pipeline and contrasts its difference with the existing CPU-utilization-based autoscaling mechanism.

4.1 Research Questions

As posited in the existing literature, relying solely on a CPU utilization-based autoscaling mechanism in Kubernetes might not yield the best possible performance for Node.js [60]. Therefore, this research aims to answer the following questions:

- Can alternate metrics such as the event loop lag be used to scale Node.js applications in a Kubernetes cluster?
- Instead of using a static setpoint for the HPA, can control-theoretic concepts be leveraged to model the system and to adaptively change the setpoint, based on the difference between measured and desired service level metrics, such as the aggregated response time of requests?

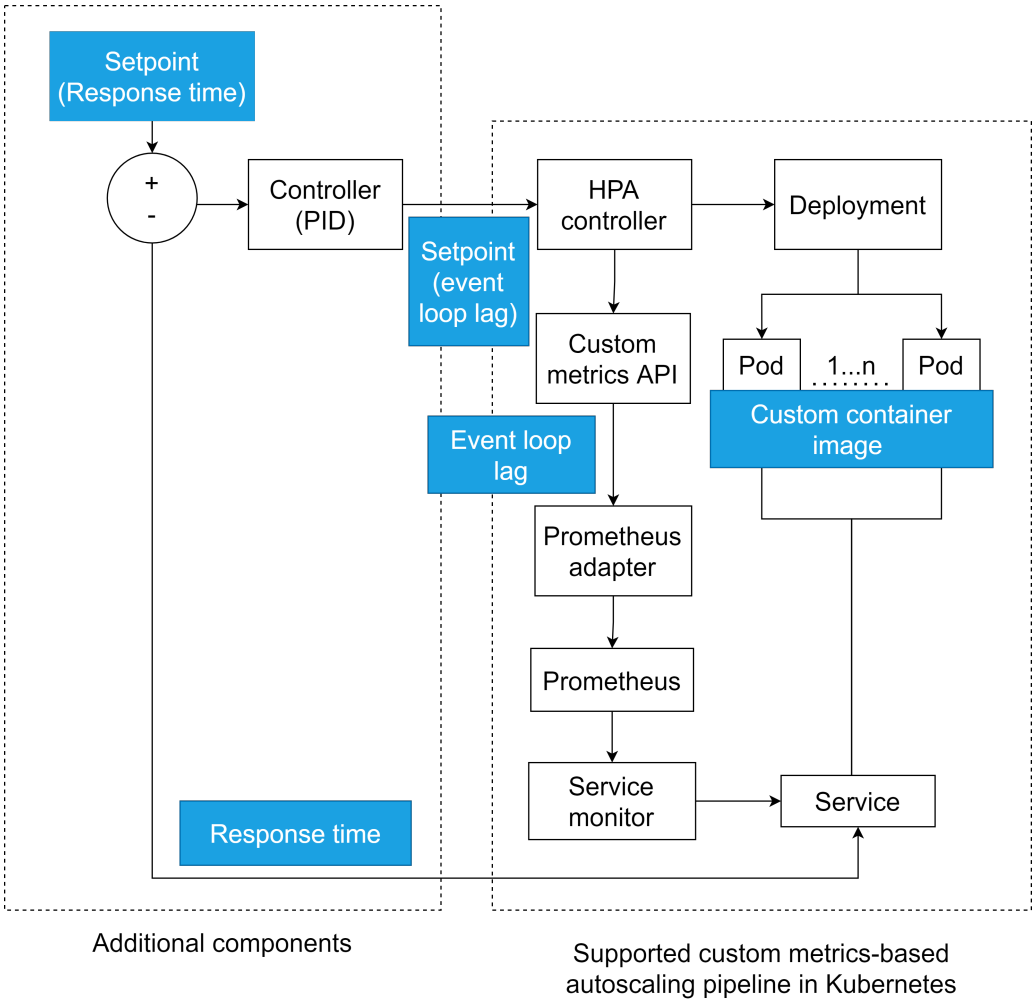


Figure 4.1: Custom Autoscaling Pipeline with a PID Controller

- Does using the custom autoscaling implementation result in better satisfaction of service level objectives?

4.2 Control Design Process

The architecture of the custom autoscaling implementation is shown in Figure 4.1. It is built in accordance with the findings of Filieri et al. and Douglas' recommendation [48, 46], following the four-step approach below for modeling the Kubernetes cluster as a black-box model, with one input and one output, and synthesizing a PID

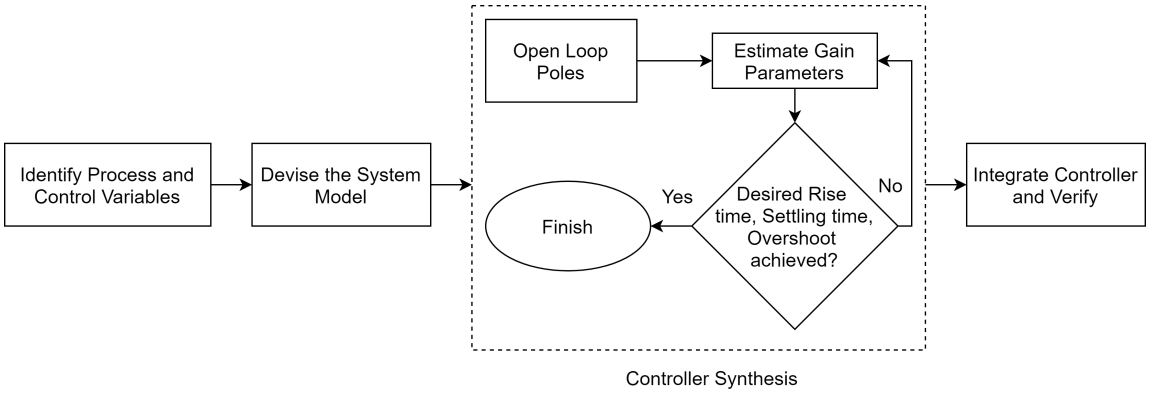


Figure 4.2: Control Design Process

controller for it, as shown in Figure 4.2.

4.2.1 Identifying Process and Control Variables

The first step in control design is to identify the process and the control variables, which determine a quantifiable value for the controlled system to strive towards. The controller is responsible for changing the control variable, such as the setpoint event loop lag in the HPA, based on the proportional, integral and derivative of the error, which is the difference between the desired and the measured process variable, i.e., response time in this case.

4.2.2 Devising a Model for the System

In classical control theory, the model of the system is often described by differential or difference equations using first principles. However, identifying a difference equation of a complex system such as Kubernetes, analyzing it and ensuring performance guarantees is a non-trivial task [46]. A natural solution in this case is system identification, which involves four basic steps [27].

- Measure the output of the system under step input, i.e., measure the response of the system when there is a sudden change in input.

- Select an appropriate model structure, such as transfer functions with adjustable poles and zeroes, state-space equations and non-linear parameterized functions that capture the mathematical relations between the process and control variables.

- Apply estimation methods to estimate the values of the model's adjustable parameters.

- Evaluate the estimated model to confirm that it reproduces the measured data to an acceptable degree.

Depending upon the information available about the system, control designers can choose either black-box modeling, which builds a model that is simply able to replicate our measured data, or use grey-box modeling, which is useful when a structure is already available, but the model's parameters are unknown. This research uses a transfer function model with adjustable poles and zeroes. It leverages MATLAB System Identification Toolbox [5] to identify a transfer function model. Figure 4.4 illustrates the actual system response and the generated model's response to a step change in input, as shown in Figure 4.3

4.2.3 Synthesizing the Controller

After model generation, the next step is to synthesize a controller that is able to change the control variable based on the measured process variable. Synthesizing a controller is a rather intuitive process and one has to take into account several adaptation goals described in previous sections. There are different methods of tuning a controller, such as the root locus technique [29], or the heuristic-based method such as Ziegler-Nichols and Cohen-Coon [51], which do not require the system's transfer function [38]. Parameters can also be derived using the system's model and the closed-loop transfer function with objectives such as pole placement or lambda

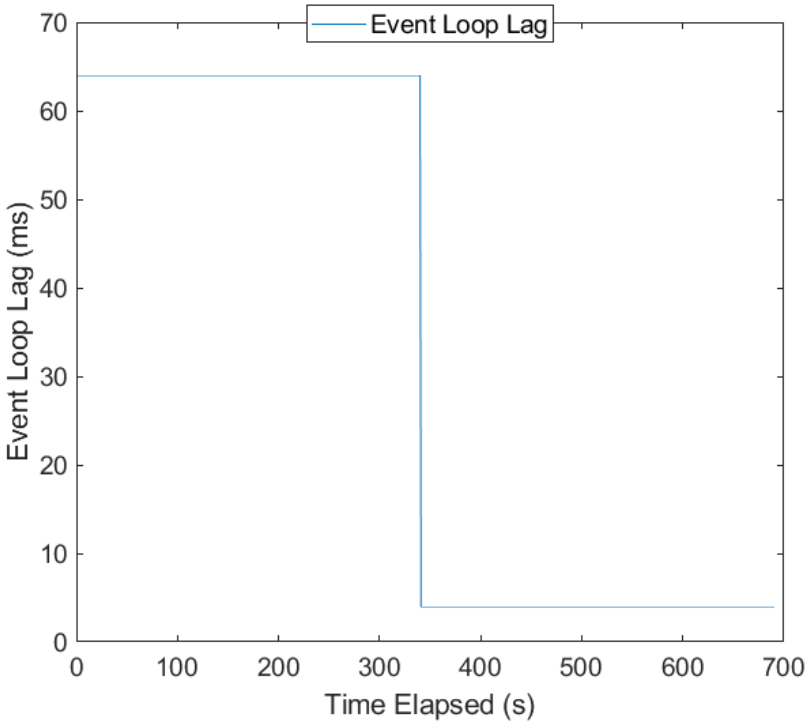


Figure 4.3: Step Change in Input

tuning [51]. Since this project requires the output response to follow certain characteristics, using heuristic-based methods is not an option. Initially, it leverages Simulink and the PID tuner app in MATLAB, to iteratively estimate controller gain parameters. Simulink allows the transfer function and the controller to be represented as configurable blocks in simulation. Users can send the same impulse input signal to the model (represented by the transfer function) as the one sent to the real system. A PID controller can then be tuned iteratively to match the model output to the system's output for the same input, using the PID tuner app and the transfer function.

4.2.4 Integrating the Controller and Verifying the System

In this phase a closed loop is formed using the system model and the controller, as shown in Figure 4.1. The response of the closed-loop system is then measured and

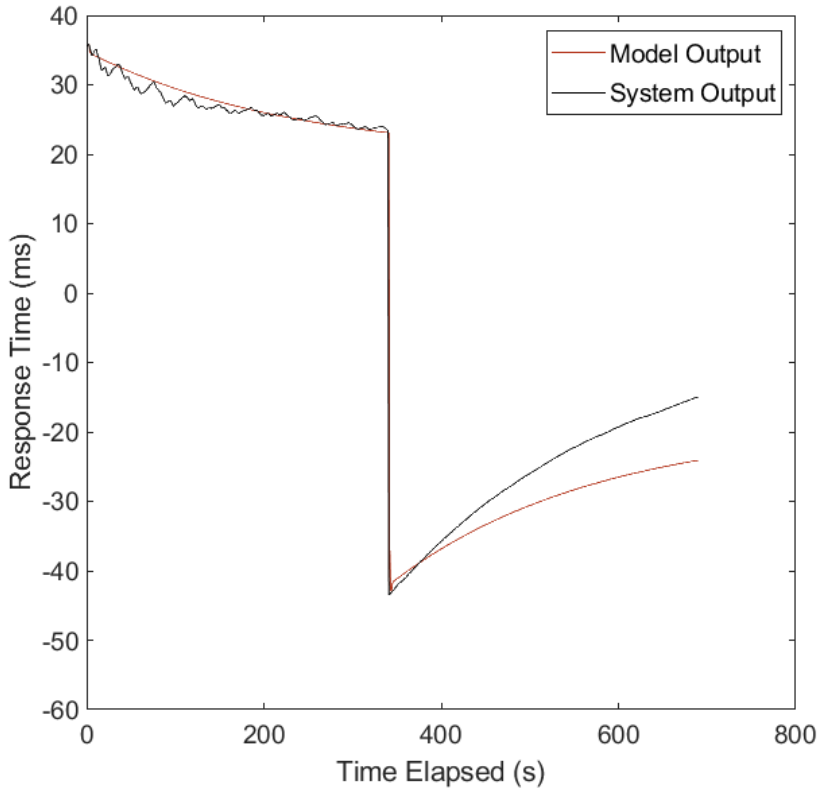


Figure 4.4: Measured System Output and the Simulated Model Output

compared with the desired set-point. After integrating the controller, the system is subject to rigorous tests to make sure that it satisfies desired response time SLOs.

4.3 On-Premises Multi-node Kubernetes Cluster Setup

In order to answer the presented research questions, a Kubernetes cluster is needed. To that end, a multi-node Kubernetes cluster is built with three physical nodes using Kubeadm [28], described below. The hardware specification of each of the three nodes is as follows:

- **CPU**: Six Intel i7-8700 cores, with two hardware threads per core

Name	Role	OS-Image	Container-Runtime
casa39	master	Ubuntu 16.04.6 LTS	docker 19.3.12
casa49	worker	Ubuntu 16.04.6 LTS	docker 19.3.12
casa50	worker	Ubuntu 16.04.6 LTS	docker 19.3.12

Table 4.1: CASA Kubernetes Cluster

- **Memory**: 32 GiB
- **Disk**: 500 GiB

Each of these three nodes is connected to the Centre for Advanced Studies-Atlantic (CASA) network and is allocated a static IP address. Each node runs Ubuntu 16.04.6 LTS and the cluster is bootstrapped with Kubeadm version 1.18.4, as detailed in Table 4.1.

4.3.1 Kubeadm Installation

Kubeadm is a tool to create a minimum viable Kubernetes cluster that complies with the best practices and also passes the Kubernetes Conformance Test [28] required for each release of Kubernetes. Using Kubeadm it is possible to get a Kubernetes cluster up and running in a wide range of devices such as laptops, cloud servers, Raspberry Pi, and more. Some of the supported architectures are: amd64, arm (32 bit), arm64, ppc64le, and s390x. The basic steps involved in configuring a single control-plane cluster using Kubeadm are as follows:

- Installing openssh-server on each node for remote access.
- Installing a container runtime such as Docker on each node.

- Installing a pod networking add-on, such as calico [9], to enable Pod-to-Pod communication.
- Installing kubeadm, kubectl and kubelet.
- Initializing the control plane using *kubeadm init* command on the master node.
- Running *kubeadm join* on the worker nodes to join the cluster.

4.4 Horizontal Pod Autoscaler Considerations

This section describes the various autoscaling pipelines available in Kubernetes and the considerations needed before using them.

4.4.1 CPU Utilization Based Autoscaling

As illustrated in Figure 4.5, kubelet, an agent that runs on each node, provides the core metrics such as CPU cumulative usage, disk usage, and memory usage to the metrics-server, which stores locally the latest values. The metrics-server exposes the available metrics to the API server. The HPA controller can then query the API server for specific metrics described in its manifest file.

4.4.2 HPA Using Custom Metrics

The architecture presented in Figure 4.6 is used to leverage autoscaling based on custom metrics. In this pipeline, a combination of node-exporter, a daemon set that runs on each node, and prom-client, a library for exporting Node.js metrics, are used to expose both core and custom metrics. Prometheus [34], which is a time-series database, collects the metrics, in a *pull-based* mechanism, from the specified API endpoint. Finally, the pipeline also utilizes a stateless API adapter [3] that pulls metrics from Prometheus and exposes them to the HPA controller.

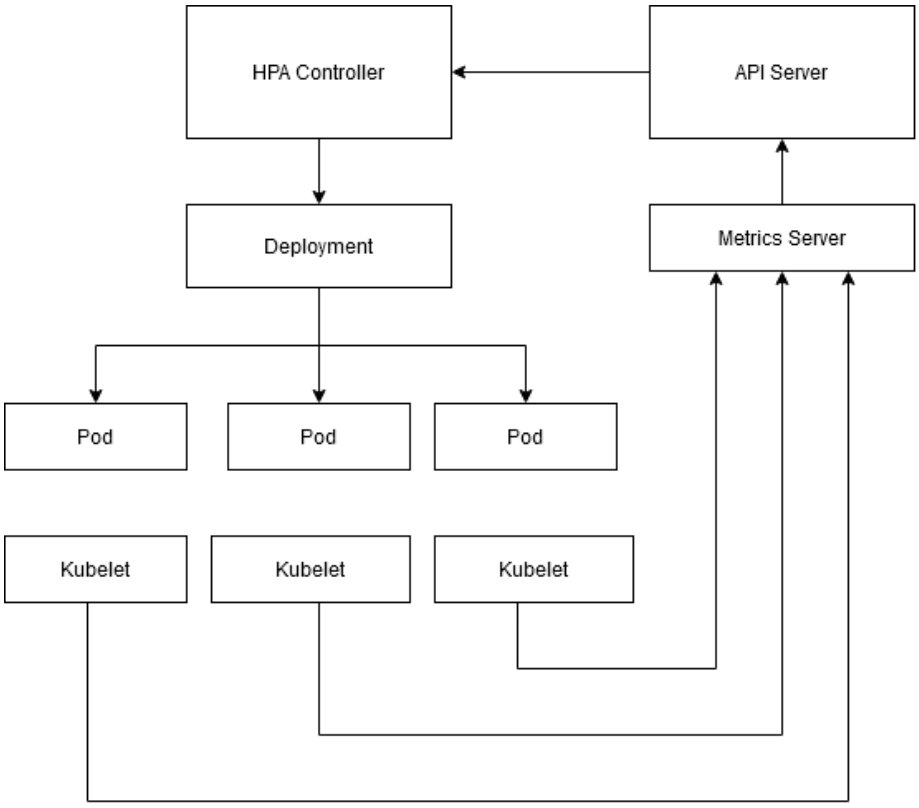


Figure 4.5: Default HPA Pipeline

4.4.2.1 Service Monitors

For Prometheus to extract metrics from target Pods, a Service Monitor must be configured using *monitoring.coreos.com/v1* API. A Service Monitor is a custom Kubernetes resource, which is not available by default and can be used to extend the Kubernetes API. Similar to how a Service is used as an abstraction on a group of Pods using labels and selectors, a Service Monitor declaratively describes a set of targets to be monitored by Prometheus. This Service Monitor resource can then be included in the Prometheus manifest file in the *serviceMonitorSelector* field.

4.4.3 Custom Autoscaling Pipeline

As illustrated in Figure 4.1, the custom autoscaling pipeline, created in this research, adds a PID controller, described in Section 4.2. The controller takes as input the

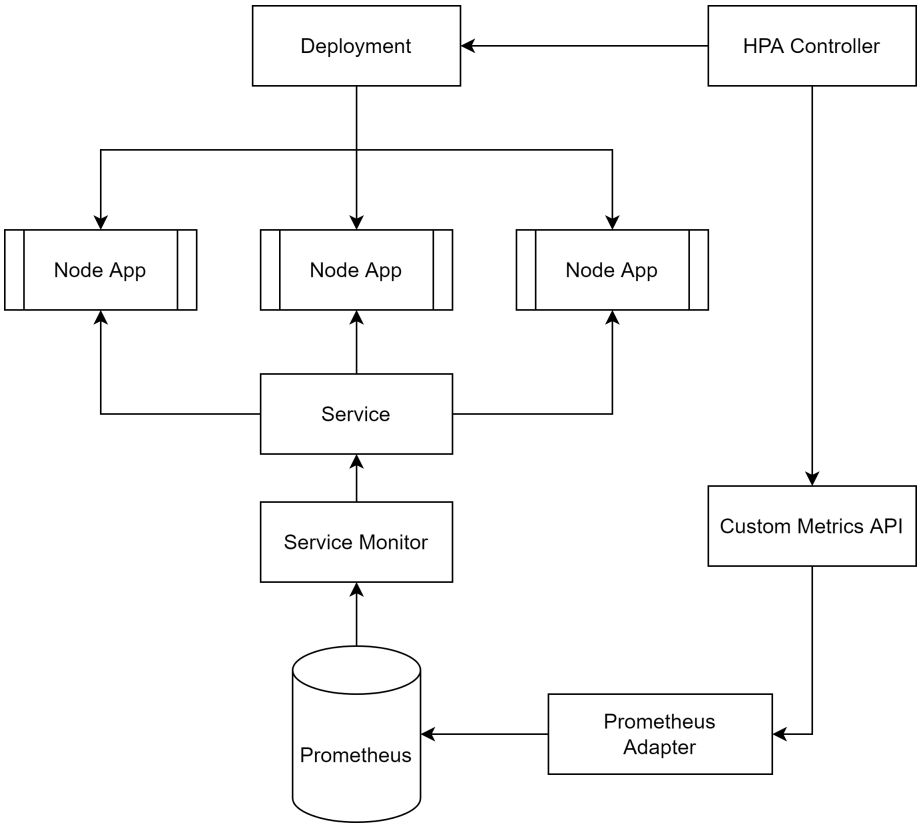


Figure 4.6: HPA Pipeline for Custom Metrics

measured probe response time and outputs the appropriate value of the event loop lag setpoint for the HPA controller. If the measured probe response time is greater than the setpoint response time, the controller outputs an appropriate value of setpoint event loop lag for the HPA controller, which could result in an addition of pods to the deployment. Conversely, if the measured probe response time is less than the setpoint response time, the PID controller outputs a value as the setpoint for the HPA controller, which could result in the decrease in the number of pods. This scale down, however, is limited by Kubernetes `--horizontal-pod-autoscaler-downscale-stabilization` flag, which is set to 5 minutes.

4.4.3.1 Complexity of the Implemented Pipeline

As illustrated in Figure 4.6, the custom autoscaling pipeline relies on several additional components to function properly. Unlike in the default CPU-based autoscaler, where the metrics server acts as the metrics aggregator and supplies the required CPU and memory usage metrics to the API server, using the Custom Metrics API involves adding components external to Kubernetes. First and foremost, the desired metrics must be exposed at an HTTP-endpoint for Prometheus to extract. This is achieved using *prom-client*, a library for exporting metrics in Prometheus format. A service monitor must then be configured to select a service object capable of reaching all the Pods with a specific label for Prometheus to target. Prometheus is responsible for storing the metrics collected from each instance of the application. Finally, an API adapter aggregates specific metrics and makes it available to the API server. A total of four additional components, as described above, are required for autoscaling based on custom metrics.

Chapter 5

Analysis

This chapter details the viability of using the event loop lag as the key setpoint metric in the HPA controller and the use of the PID controller to adaptively change the setpoint based on the difference between desired and measured response times. Using metrics such as the 95th percentile response times, percentage of SLO violation, and CPU request for both CPU utilization-based autoscalers and the custom implementation, we draw several conclusions for the variety of workloads tested.

5.1 Experimental Setup

This research leverages a distributed HTTP load generation tool called locust, which enables creating tasks that can be run by each simulated user [6]. For each running user, locust creates a greenlet, a lightweight co-routine for cooperative and sequential execution, which will invoke the task methods. This approach enables us to simulate the user behavior in an application, as opposed to the more conventional analysis of a particular type of request separately.

We developed a sample Node.js application with express.js, an open-source back-end web application framework for Node.js [20]. The application has multiple HTTP endpoints, which is designed to stress the crucial components of Node.js: the event

loop and the threadpool. There are four endpoints concerned with inserting, fetching, updating and deleting items from a MongoDB collection, which is a typical I/O operation that utilizes the event loop. There is also an endpoint that utilizes the threadpool by using *pbkdf2*, which is a password-based key derivation function from Node.js' crypto module. Finally, the last endpoint also stresses the event loop by performing a recursive Fibonacci computation up to the 25^{th} term, which is a compute-intensive task for the event loop as it blocks the event loop during the computation. This could potentially impact the overall performance when many concurrent requests are performing this computation.

In addition to the sample application that we created, we also include additional applications that are derived from an existing suite developed by Zhu et al., in which the authors study the impact of using event-driven systems, such as Node.js, in server-side web applications [71]. A summary of the included applications is listed in Table 5.1. Unlike the sample application, which is designed to stress both event loop and the threadpool, using CPU and I/O intensive requests, the applications in the author's suite only stress one of the aforementioned components of Node.js.

The containerized applications are managed by Kubernetes and exposed to external traffic with a Load Balancer-type service at a specific IP address. Each Pod running the containerized application is allocated a minimum of 100 millicores of CPU and a maximum of 250 millicores of CPU. Therefore, for the maximum number of Pods, set using $maxReplicas = 15$ field in the deployment manifest, the CPU usage ranges from a minimum of $15 \times 100 = 1500$ millicores to a maximum of $15 \times 250 = 3750$ millicores. The CPU allocation for each Pods are set to the aforementioned values for replicability in cloud environments. The sample applications use a stand-alone MongoDB and Redis instances deployed in a separate namespace using Bitnami Helm Chart [7].

The number of users on the server is varied using the *StageShape* class provided

Workload	Description
Node.js Todo	An interactive task management tool used to create and delete tasks, built using Node.js, express.js, and Redis
Word Finder	A word searching application that matches user-entered patterns against 200,000 words from an English dictionary, built using Node.js and express.js
Static Server	A file server that serves static HTML, CSS and JavaScript files, built using Node.js and express.js

Table 5.1: A Summary of Workloads

by locust. There are two usage patterns that are explored in this study: a constant number of users, and a step-increasing number of users. As mentioned in the previous section, each user is tasked with executing the methods defined in the aforementioned HTTP endpoints sequentially. The number of users are varied to sufficiently load test the application, depending on its type. For instance, for Static Server, the number of simulated users reaches up to 200, but for Word Finder 20 users are sufficient. In any case, the user ramp-up pattern is similar to Figure 5.1.

5.2 Results

To compare the custom autoscaling pipeline with the existing CPU utilization-based autoscalers, this analysis includes benchmarks that compare the response times, number of computing instances (Pods) as well as the CPU request. In addition, the output of the PID controller to the HPA controller, which is the independent variable, in response to the measured probe response time (dependent variable), is also reported.

Using the aforementioned metrics, we aim to answer the research questions from Section 4.1, regarding the use of the event loop metric as the key setpoint metric and the ability of the custom autoscaling implementation to satisfy the SLOs.

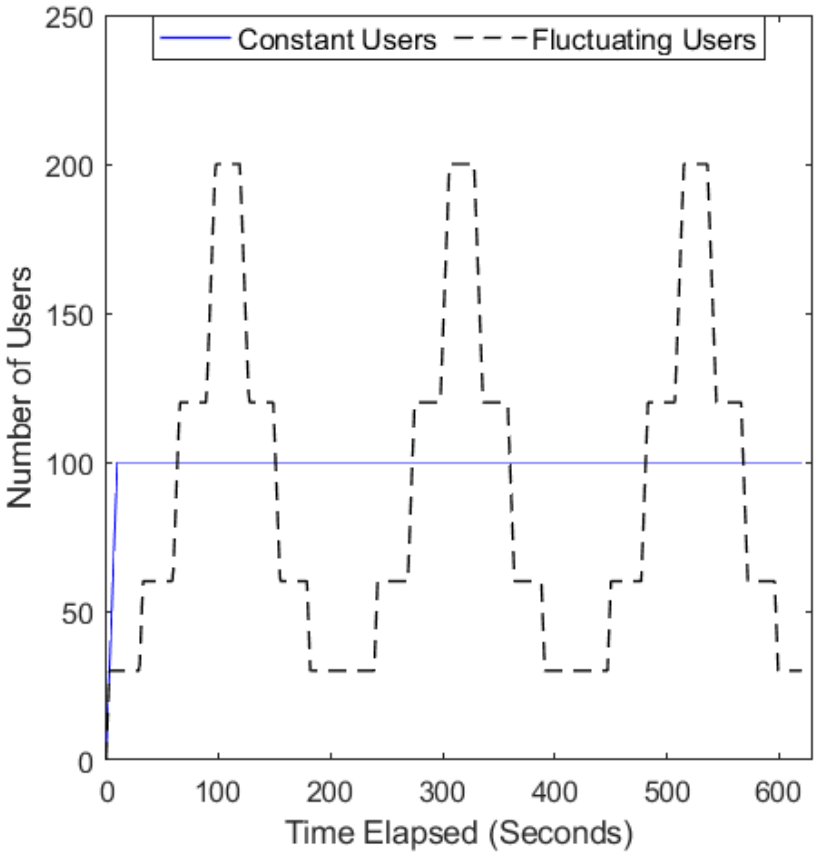


Figure 5.1: User Ramp-up Pattern

5.2.1 System Model and Controller Gain Coefficients

A range of models is available in the MATLAB System Identification Toolbox to select a best fit for the measured input-output data, such as non-linear models, transfer function models, state-space models, and correlation models. In this case, we use a transfer function model, with poles and zeroes, as it provided the best fit to the estimation data [24]. We specifically utilize a frequency-domain transfer function since time-domain representation of the system could involve complex convolutions. The discrete-time transfer function, with four poles and two zeroes, created using non-linear least squares search-based updates is as follows:

$$tf(z) = \frac{1.003z^{-1} - z^{-2}}{1 - 1.092z^{-1} + 0.1014z^{-2} + 0.009748z^{-3} - 0.01548z^{-4}} \tag{5.1}$$

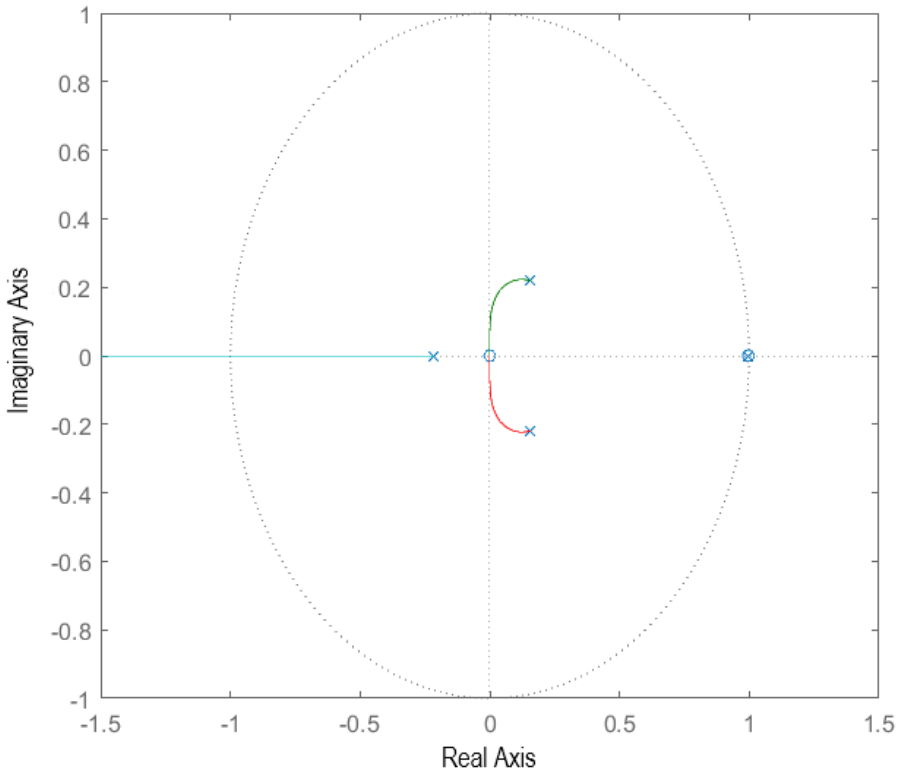


Figure 5.2: Root Locus Plot

The transfer function, generated using the System Identification Toolbox, shows a 61.86% fit to the estimation data. As can be seen from Figure 5.2, the poles, denoted by x in the diagram, all lie within the unit circle around the origin in the complex plane. Therefore, the system represented by the transfer function is stable [51].

When synthesizing a controller, a designer is either expected to have deep knowledge of the system's internal mechanics or be in possession of a model of the system that describes the system, from which the controller gain parameters can be calculated iteratively using a range of tools such as MATLAB PID tuner and Simulink. The tuning of the PID controller gain parameters must ensure specific performance requirements of the closed-loop system, such as rise time, settling time, overshoot, and stability, and must follow a desired trajectory. This step generally involves making certain engineering trade-offs as fulfilling all four characteristics simultaneously is not

possible [51]. Since we require the closed-loop system to be stable and the control action to result in a speedy response, we tune the controller gain parameters accordingly. The controller gain coefficients, computed iteratively using the open-loop transfer function and the PID Tuner app in MATLAB is:

$k_p = 0.81605$, $k_i = 0.26051$, and $k_d = 0.64149$

As discussed in the previous chapters, the proportional, integral, and derivative terms affect the current error, cumulative past errors, and future errors. Therefore, higher value of the proportional gain coefficient means we prioritize eliminating present error. Similarly, we put a lower priority on the significance of future errors with the slightly lower derivative gain coefficient. Since we are aware of the load that is expected on the application, it can enable prediction of rate of change of error and react accordingly. Finally, due to higher values of proportional and derivative gain coefficients and the known load patterns, the integral gain coefficient is set to a lower value since we do not want transient errors to heavily affect the output of the controller.

5.2.2 Response Times

In order to provide a comparison with the CPU-utilization-based autoscaling, we utilize a box plot to contrast the overall performance when the average CPU utilization threshold set to 20%, 40%, 60%, 80%, and 90% with the performance of our custom autoscaling pipeline. The purpose of a box plot is to illustrate the spread and centers of the measured response times, by including metrics such as the mean and the median of the measured response times. The data points outside the box and whiskers are the outliers. The whiskers are lines that extend outward from the boxes and indicate the variability outside $Q1$ and $Q3$. The lower whisker represents the values up to $Q1 - 1.5 * IQR$, whereas the upper whisker represents the values up to $Q3 + 1.5 * IQR$, where IQR stands for the inter-quartile range ($Q3 - Q1$). The

median response time is denoted by the horizontal line with a notch at the end, the mean response time is represented by a square, and the 95^{th} percentile response time is denoted by the circle, in each box plot.

The performance of CPU utilization-based autoscaling with various thresholds is taken as a baseline in the experiments. The resulting response times when CPU utilization-based autoscaling is used and the ones achieved by using our custom autoscaling implementation for the aforementioned different workloads are as follows:

5.2.2.1 Custom Workload

As can be seen from the plot in Figure 5.3, for the custom workload when the number of users is constant, the response time of our autoscaling implementation is on par with the performance achieved by the CPU utilization-based autoscaling when the threshold is set to 20%, 40%, and 60%. For thresholds of 80% and 90%, the 95^{th} percentile response time is $81ms$ and $95ms$, respectively, which is significantly greater than the $47ms$ achieved using the custom implementation.

For a fluctuating number of users, however, the response time increases with the increase in CPU utilization threshold. As can be seen from the plot in Figure 5.4, the response time is minimal when the CPU utilization threshold is set to 20% and 40%, but starts increasing drastically for thresholds of 60%, 80%, and 90%. In the case of the custom implementation, there is a large variation in the measured response times, with 95^{th} percentile response time of 670ms and a standard deviation of 192.77, which is greater than all other cases. CPU utilization-based autoscalers with thresholds of 20%, 40%, 60%, 80%, and 90% produce a 95^{th} percentile response time of $17ms$, $23ms$, $200ms$, $170ms$, and $310ms$, respectively.

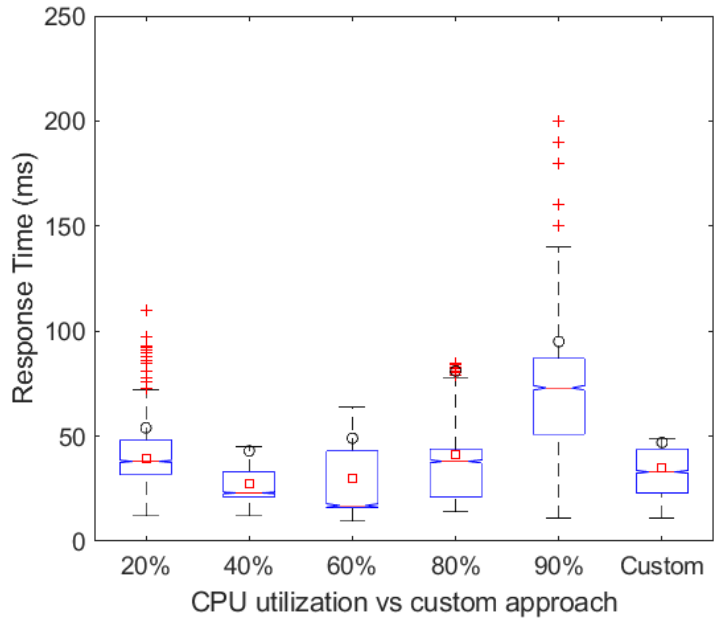


Figure 5.3: Response Times for Custom Workload (Constant Users)

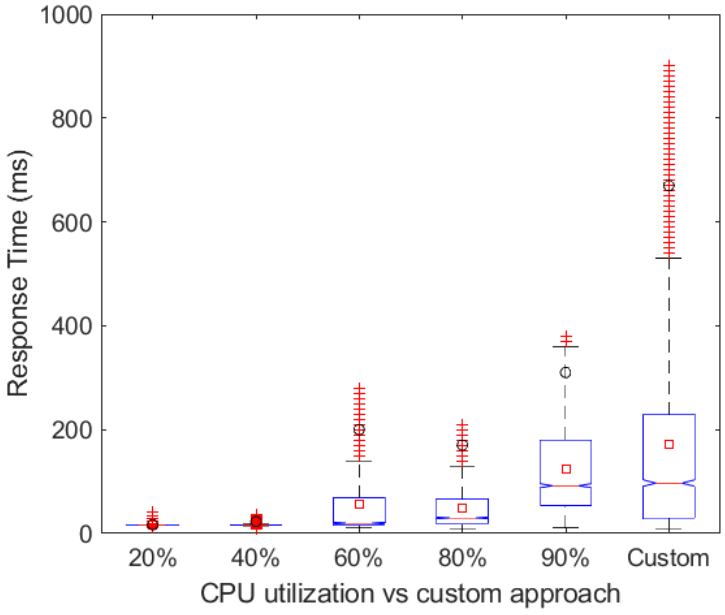


Figure 5.4: Response Times for Custom Workload (Fluctuating Users)

5.2.2.2 Static Server

As can be seen from Figure 5.5, for a constant number of users, the performance of the custom autoscaling implementation is on par with CPU utilization-based autoscalers.

The custom implementation has a 95^{th} percentile response time of $20ms$, and a standard deviation of 2.01. The 95^{th} percentile response times for CPU utilization thresholds of 20%, 40%, 60%, 80%, and 90% are $21ms$, $23ms$, $22ms$, $21ms$, and $22ms$, respectively.

For a fluctuating number of users, however, the custom autoscaling implementation does not perform as well as the CPU utilization-based autoscalers. As can be seen from the plot in Figure 5.6, its 95^{th} percentile response time and the standard deviation of $60ms$ and 12.75, respectively, are higher than all of the CPU utilization-based thresholds. For CPU utilization-based autoscalers, like for the custom workload, the response times increase with the increase in CPU utilization threshold. The best performance is achieved when CPU utilization threshold is set to 20%, with a 95^{th} percentile response time of $31ms$ and a standard deviation of 4.68. Other thresholds, such as 40%, 60%, 80%, and 90% yield 95^{th} percentile response times of $34ms$, $37ms$, $41ms$, and $42ms$, respectively.

5.2.2.3 Node.js Todo

Unlike in the custom workload, this application does not contain any CPU or thread-pool bound tasks, and can thus be used to compare the performance of the custom autoscaling implementation with the CPU utilization-based autoscaling when only network I/O is involved.

As can be seen from the plot in Figure 5.7, for a constant number of users, the 95^{th} percentile response times are on par for both the custom autoscaling implementation and CPU utilization-based autoscaling. The custom implementation yields a 95th percentile response time of 10ms, with a standard deviation of 0.62, whereas the best-performing CPU utilization threshold is 40%, with a 95^{th} percentile response time of $9ms$ and a standard deviation of 0.89.

For a fluctuating number of users, as can be seen from Figure 5.8, the custom au-

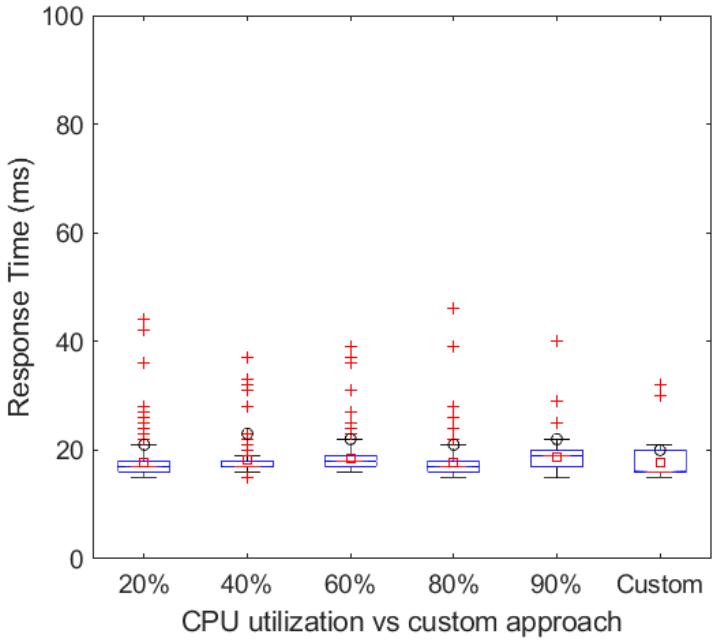


Figure 5.5: Response Times for Static Server (Constant Users)

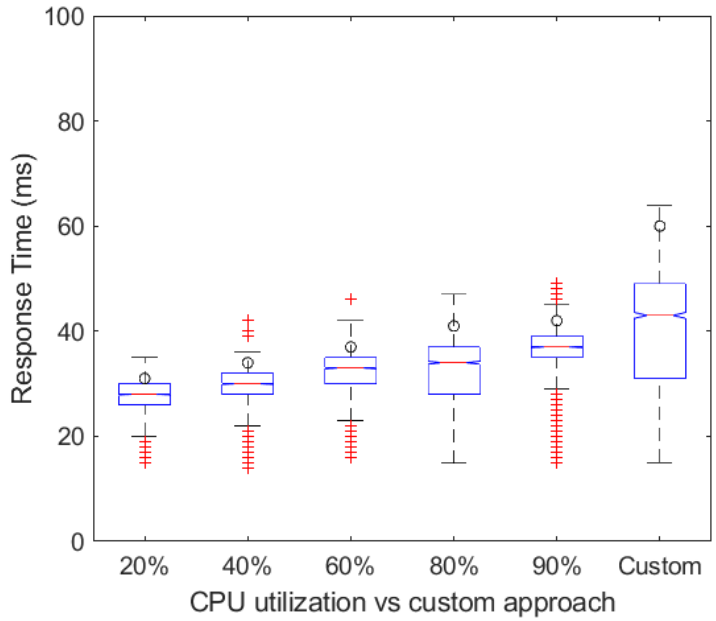


Figure 5.6: Response Times for Static Server (Fluctuating Users)

toscaling implementation produces a 95^{th} percentile response time of 10ms, which is on par with all the CPU utilization-based autoscaling thresholds. However, the standard deviation of 1.85 is higher than the ones observed in CPU utilization-based

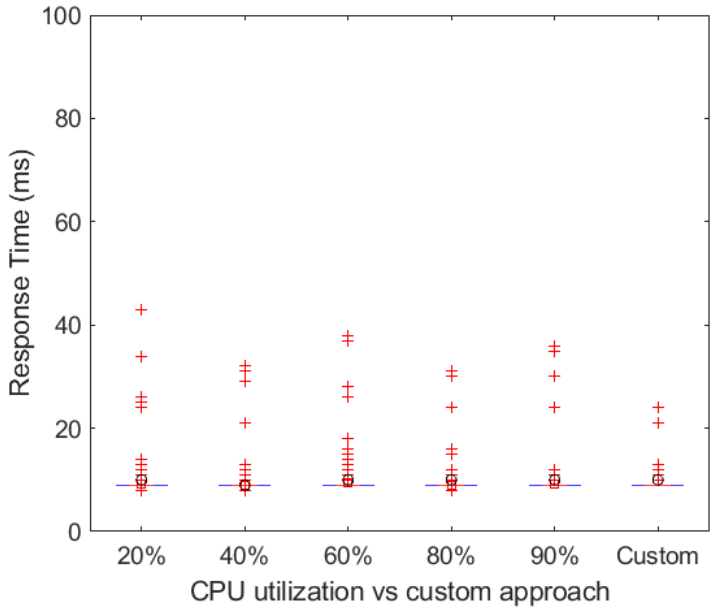


Figure 5.7: Response Times for Node Todo (Constant Users)

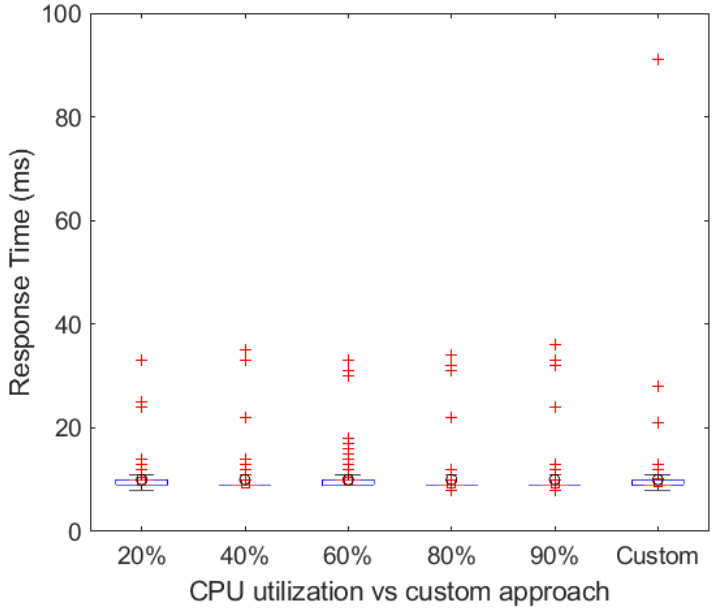


Figure 5.8: Response Times for Node Todo (Fluctuating Users)

autoscalers. The best performance is achieved when the CPU utilization threshold is set to 90%, which yields a 95^{th} percentile response time of $9ms$.

5.2.2.4 Word Finder

For a constant number of users, as can be seen from the plot in Figure 5.9, there is no clear trend for response time with the increase in CPU utilization threshold in CPU utilization based autoscaling. From 20% to 40%, the 95^{th} percentile response time increases, but it starts to decrease at 60% and continues to decrease up to 90%. As is evident from the graph, the custom autoscaling pipeline performs better than the 40% and 60% CPU utilization threshold, with a 95^{th} percentile response time of $1,390ms$ and a standard deviation of 362.22.

For a fluctuating number of users, however, there is a clear trend for response time. The 95^{th} percentile response time seems to increase with the increase in CPU utilization threshold up to 80%; however, at 90%, the 95th percentile response time decreases. For the custom implementation, as seen in the plot in Figure 5.10, the measurements contain a significant number of outliers, with a 95^{th} percentile response time of $1,251ms$ and a large standard deviation of 362.09.

5.2.2.5 Response Times Summary

The following Tables 5.2, and 5.3 illustrate the performance difference between the baseline implementations and the custom autoscaling implementation. The tables serve to provide a contrast between the percentage of SLO violations when the CPU utilization-based autoscaler is used with various thresholds and the custom autoscaling implementation. SLO violations are instances where the measured 95th percentile response time is above the setpoint response time of 100ms for each second.

As is evident from Table 5.2, there are no SLO violations for the custom workload when there is a constant number of users. For 20% and 90% CPU utilization thresholds, there is a SLO violation of 0.04% and 2.25%, respectively. However, for the custom autoscaling implementation, like other CPU utilization thresholds, there are no SLO violations. In case of a fluctuating number of users, as can be seen from

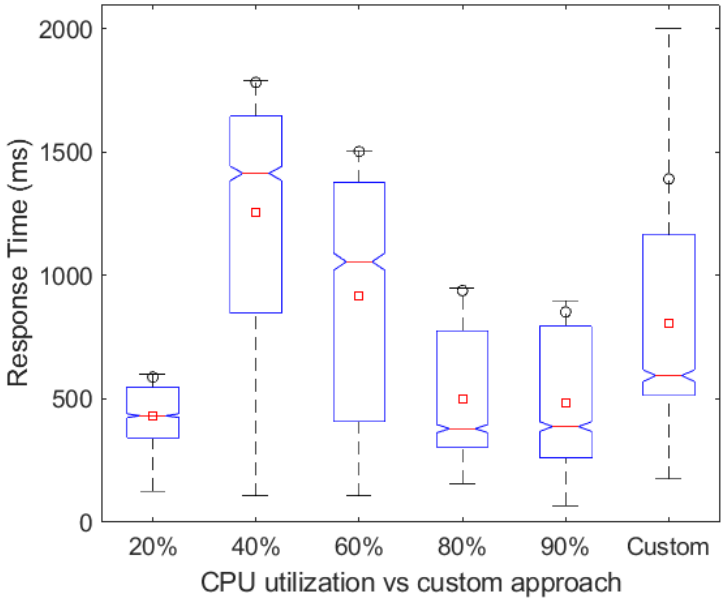


Figure 5.9: Response Times for Word Finder (Constant Users)

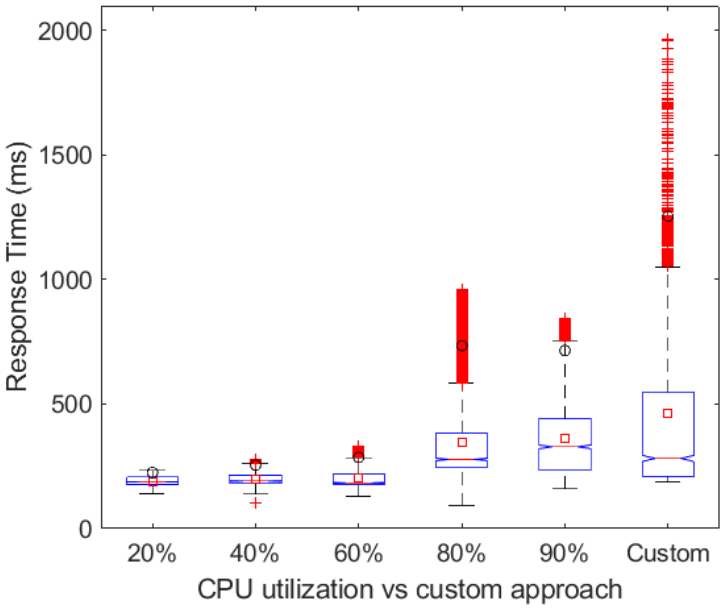


Figure 5.10: Response Times for Word Finder (Fluctuating Users)

Table 5.2, the performance suffers with the additional users and subsequent requests. For a CPU utilization threshold up to 40%, there are no SLO violations. However, from 60% onwards, the SLO violation percentage increases, with a dip at 80% and then again increases at 90%. The custom autoscaling implementation performs worse

Autoscaler Threshold	20%	40%	60%	80%	90%	Custom
SLO violations % (constant users)	0.04	0	0	0	2.25	0
SLO violations % (fluctuating users)	0	0	18.46	11.25	40.32	48.62

Table 5.2: SLO Violations for Custom Workload

Autoscaler Threshold	20%	40%	60%	80%	90%	Custom
SLO violations % (constant users)	99.625	99.625	99.625	99.625	99.464	99.625
SLO violations % (fluctuating users)	99.625	99.625	99.625	99.464	99.625	99.625

Table 5.3: Response Times (in milliseconds) for Word Finder

than all the CPU utilization-based autoscaling thresholds, with an SLO violation of 48.62%.

For Static Server and Node Todo there are no SLO violations during the course of the benchmarks as the tasks are not compute-intensive. In the case of Word Finder, as can be seen from Table 5.3, almost all requests result in an SLO violation. This can be explained by the computationally intensive nature of each request. For each request, the server uses regular expression to match the user query with $200,000$ words from an English dictionary. Therefore, the running time of each task influences the overall response time.

5.2.3 Controller Performance

To measure the controller's performance, a probe request is sent to the application every $200ms$, so as to not interfere with the actual benchmarking, which is done through locust. We then calculate the 95^{th} percentile response time of the last 25 sample response times, which is used by the PID controller as input. The controller,

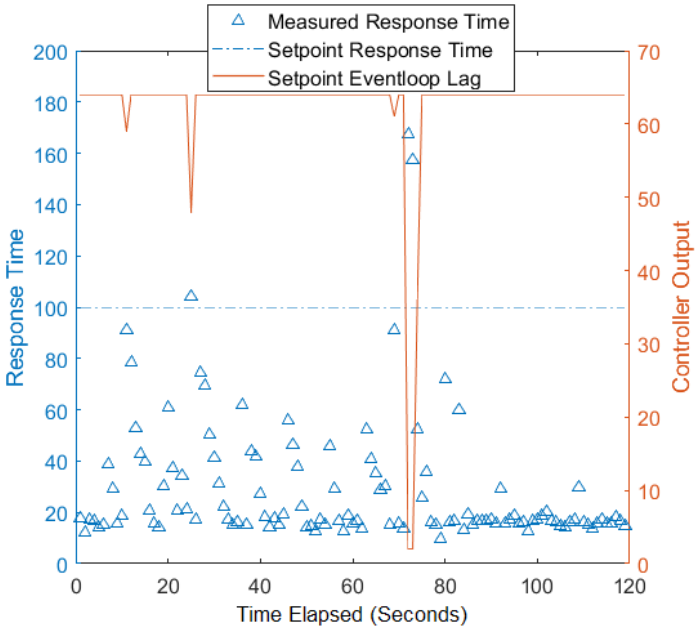
then, produces a setpoint event loop lag value for the HPA controller. In other words, the setpoint event loop lag is the independent variable in the experiment, whereas the 95^{th} percentile of the measured probe response time is the dependent variable.

5.2.3.1 Custom Workload

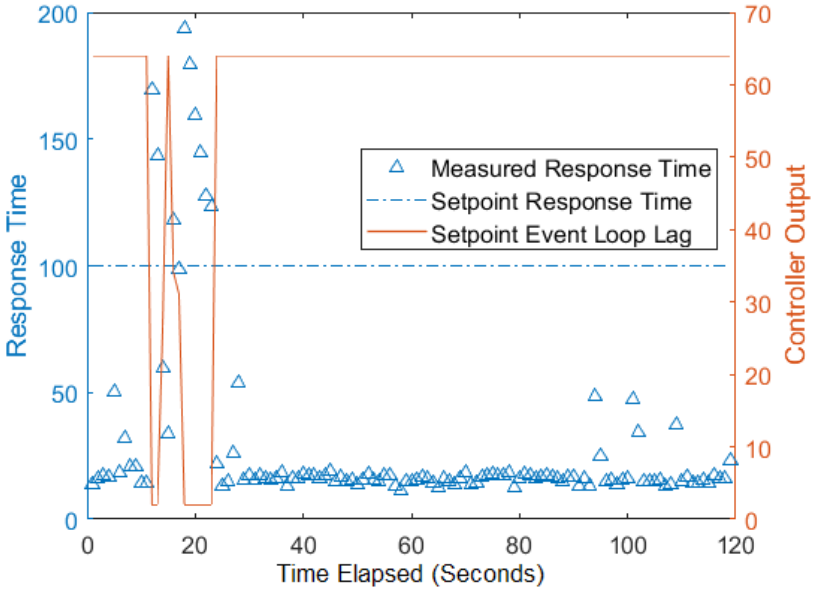
The custom workload contains a variety of tasks for each locust user to run. It contains a combination of CPU bound and I/O bound tasks that need to run both on the main thread and the threadpool.

For both a constant and fluctuating number of users, as can be seen from Figure 5.11, the controller acts to decrease the value of setpoint event loop lag to the HPA controller when the measured response time exceeds the desired (setpoint) response time. This has a direct effect on the number of pods in deployment. Whenever there is a discrepancy between the measured event loop lag and the setpoint event loop lag, which is the output of the controller, the HPA controller directs the deployment to increase or decrease the number of pods. This phenomenon is evident from the graphs in Figure 5.12. The result of the increase in the number of pods can again be seen in Figure 5.11, where the measured probe response time lies well below the setpoint response time.

However, the higher overall 95^{th} percentile response times and outliers observed for an increased number of users in the custom implementation, as seen in Figure 5.4, is due to the tasks a simulated user has to perform. In addition to database queries, the custom workload also contains CPU bound tasks. As a result, the CPU utilization-based autoscalers react and scale on time whereas the custom implementation, which is not dependent on the degree of CPU utilization and only on measured probe response time and event loop lag, scales at a later instance. Also, the large downscaling as seen in Figure 5.12 (b), due to the measured probe response time consistently being less than the setpoint, could also be impacting the overall benchmark response



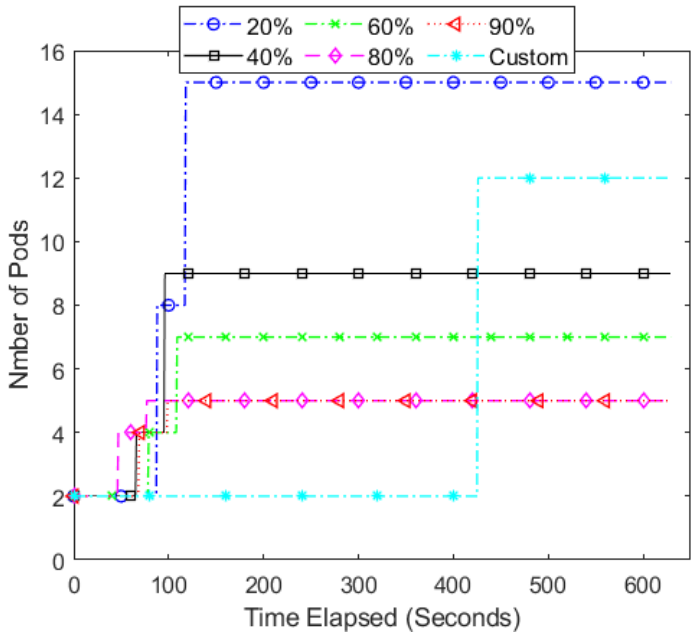
(a) Constant Users



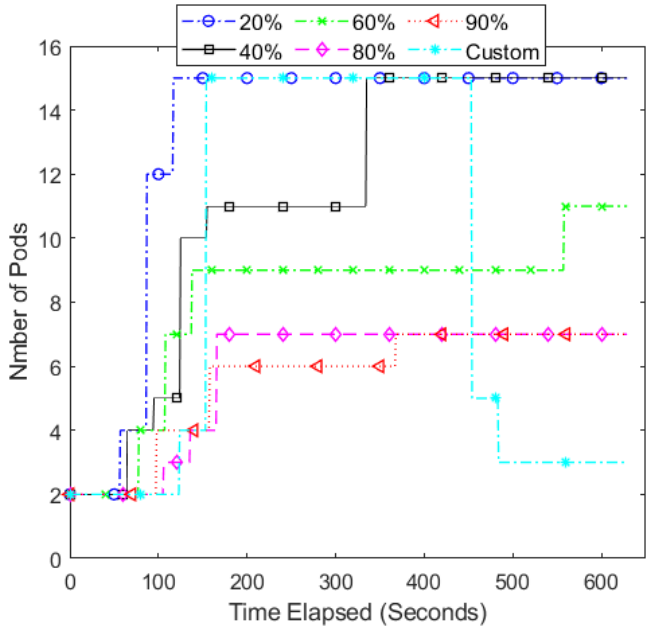
(b) Fluctuating Users

Figure 5.11: Response Time and Controller Output

time.



(a) Constant Users

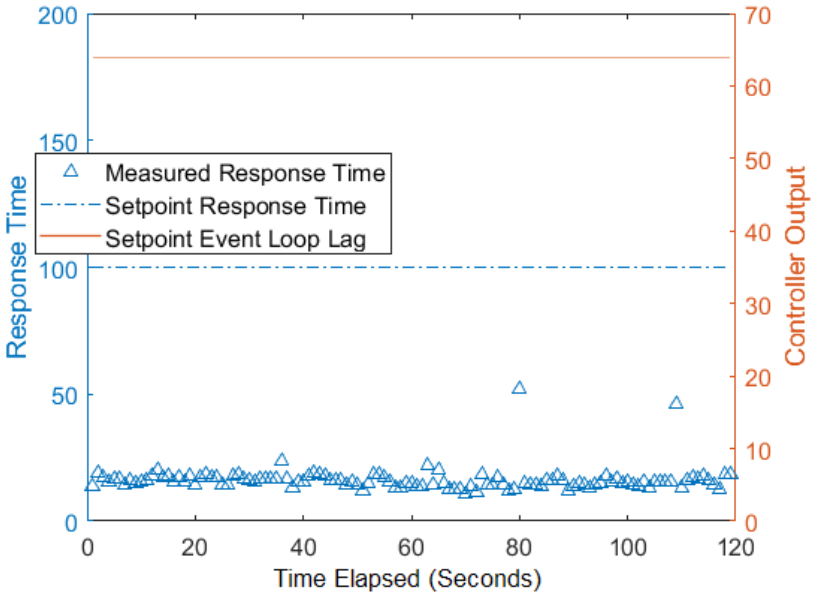


(b) Fluctuating Users

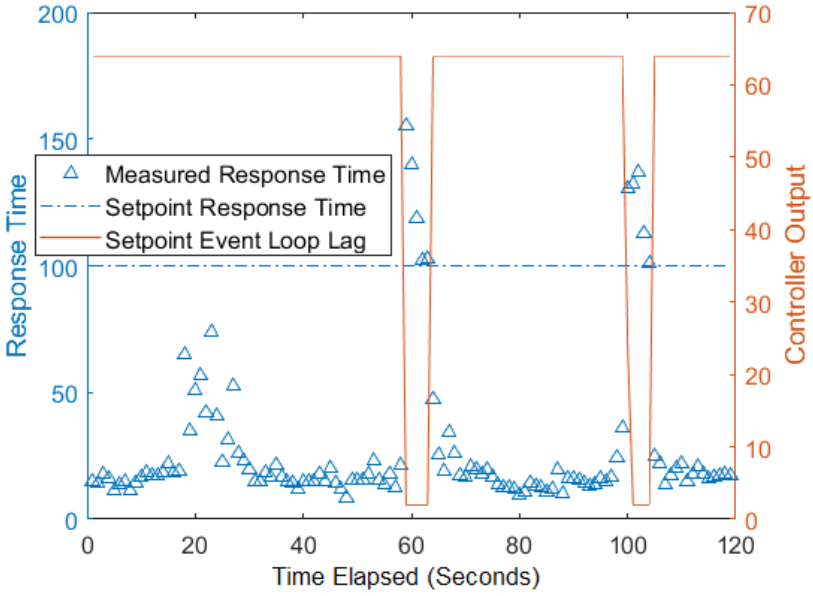
Figure 5.12: Number of Pods in Deployment

5.2.3.2 Static Server

Unlike the custom workload, the static server does not have a variety of tasks for the locust user to run. Therefore, the majority of the time for each request is spent



(a) Constant Users

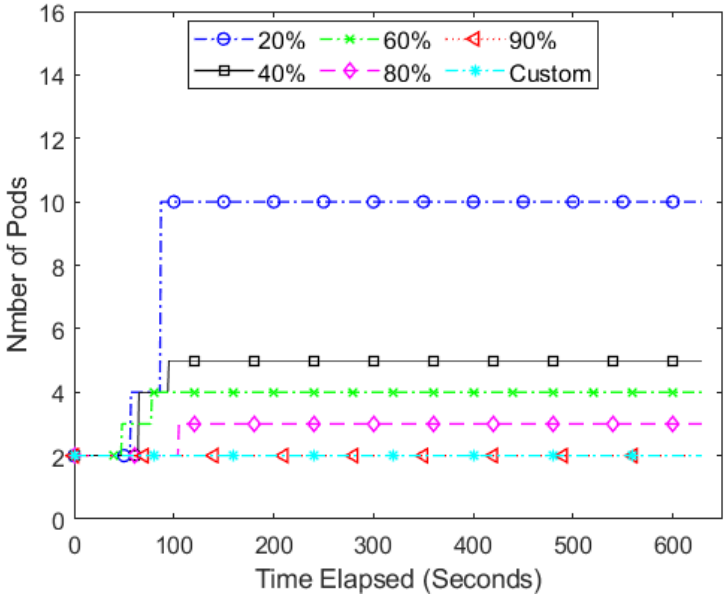


(b) Fluctuating Users

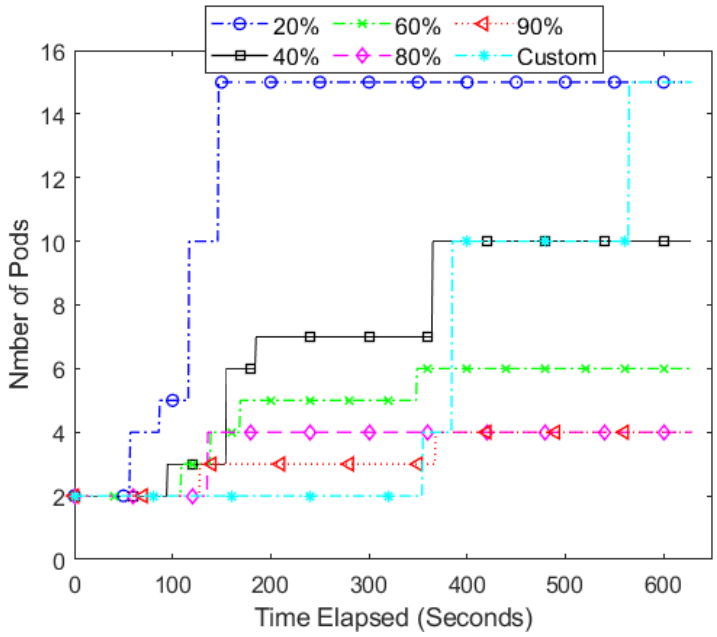
Figure 5.13: Response Time and Controller Output

on necessities such as setting appropriate response headers, response status, finding correct mime types, and calculating file sizes.

As can be seen from Figure 5.13 (a), for a constant number of users, the measured probe response time is always well below the setpoint response time. Therefore,



(a) Constant Users



(b) Fluctuating Users

Figure 5.14: Number of Pods in Deployment

the controller output (setpoint event loop lag) remains high and no autoscaling is triggered. The graph in Figure 5.14 (a) also verifies this as the number of pods do not increase from a minimum value of two.

For a fluctuating number of users, as seen from the graph in Figure 5.13 (b), there

are two instances where the measured probe response time is significantly greater than the setpoint response time. Only during these periods, the controller's output reaches the minimum value, which triggers a large autoscaling, as seen from the graph in Figure 5.14 (b). As a result of scaling, the measured probe response time falls below the setpoint response time.

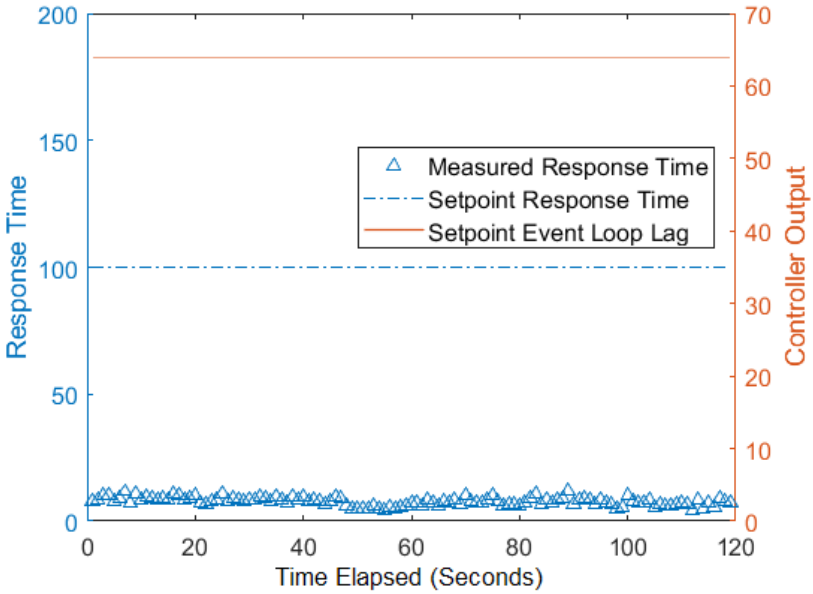
Compared to some CPU based thresholds, which scale aggressively from the beginning, the custom implementation takes a conservative approach. This lack of additional pods affects the overall benchmark response time, as reported in Figure 5.6. Furthermore, the *sendFile* method in express.js used to send files to requesting clients uses asynchronous streams and pipes and therefore, does not drastically affect the event loop lag nor cause the probe response time to be above the setpoint response time. However, the CPU utilization does seem to increase with the increase in the number users, thus triggering autoscaling early in lower CPU utilization thresholds as seen from Figure 5.14 (b).

5.2.3.3 Node.js Todo

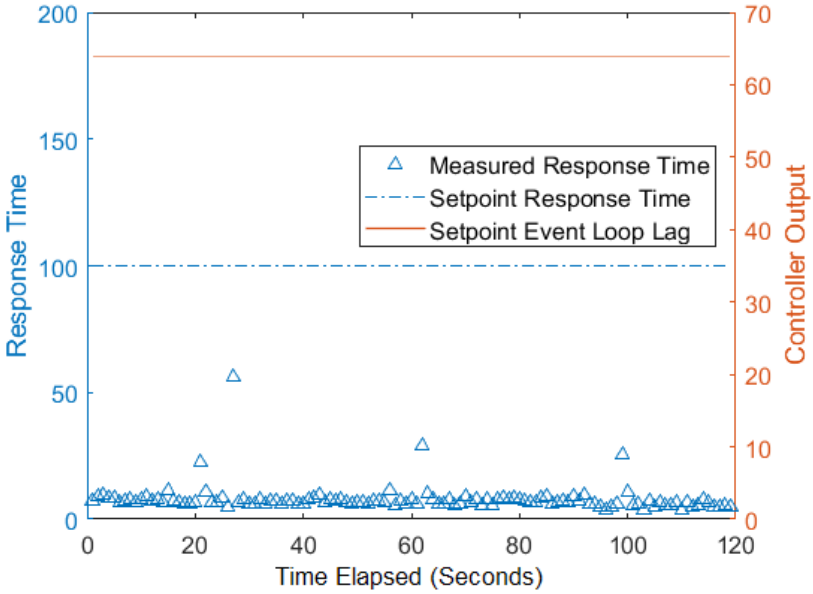
The Node Todo application has two task methods for the locust users to invoke: one to create an item and the other to delete an item. The server persists the items in Redis, which is an in-memory data store (database).

As can be seen from Figure 5.15 (a), for a constant number of users, the measured probe response remains constant with time. As a result, as seen from Figure 5.16 (a), no additional pods are provisioned during the entirety of the benchmark run. This is in accordance with the results we have seen in Section 5.2.2.3, where the 95^{th} percentile response time is $10ms$.

Similarly, for a fluctuating number of users, as seen from the graph in Figure 5.15 (b), the measured probe response time also does not increase with time and the increasing number of users. This is in accordance with our findings for a constant number of



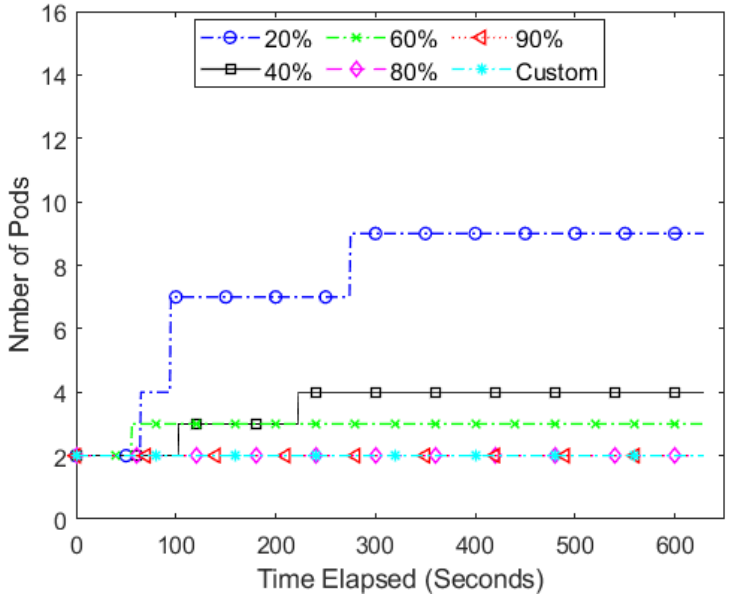
(a) Constant Users



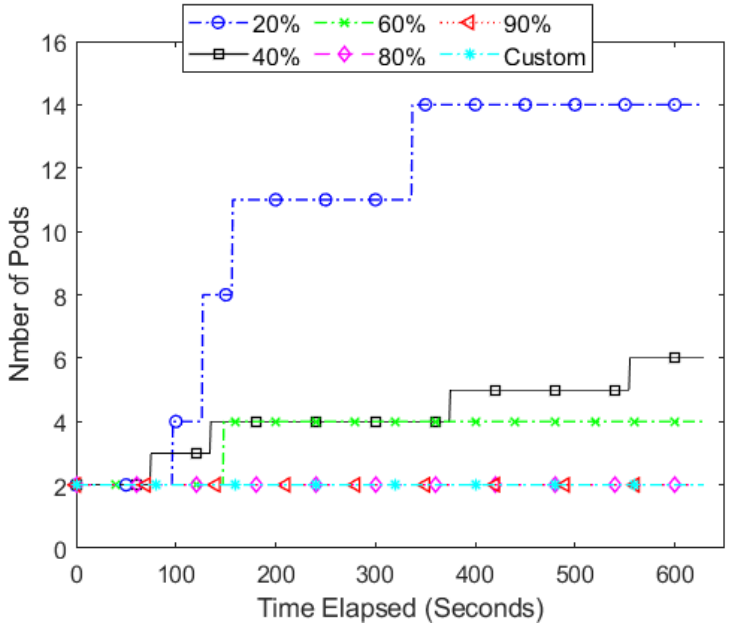
(b) Fluctuating Users

Figure 5.15: Response Time and Controller Output

users: the increase in the number of users does not seem to affect the probe response time. This also aligns with the findings of Lei et al., where the authors compare the performance of PHP, Node.js, and Python and conclude that Node.js performs the best for I/O intensive workloads [56], even for thousands of requests per second.



(a) Constant Users

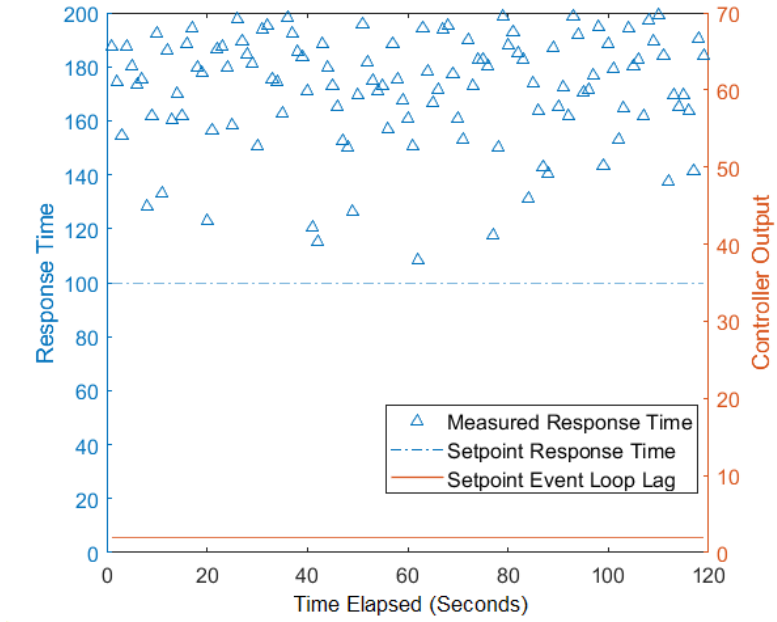


(b) Fluctuating Users

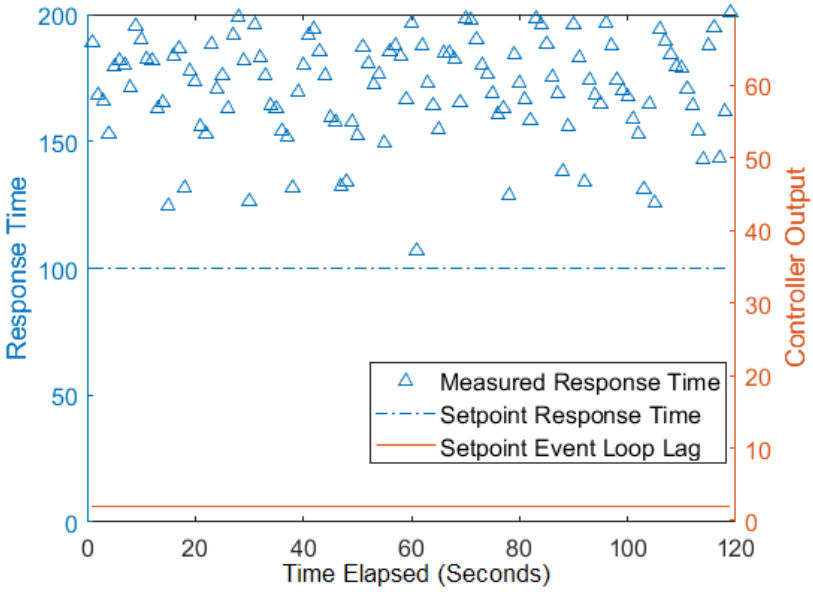
Figure 5.16: Number of Pods in Deployment

5.2.3.4 Word Finder

Like Static Server and Node Todo, Word Finder also consists a set of non-varying tasks for the locust users to run. It has one HTTP endpoint "/search", which accepts



(a) Constant Users

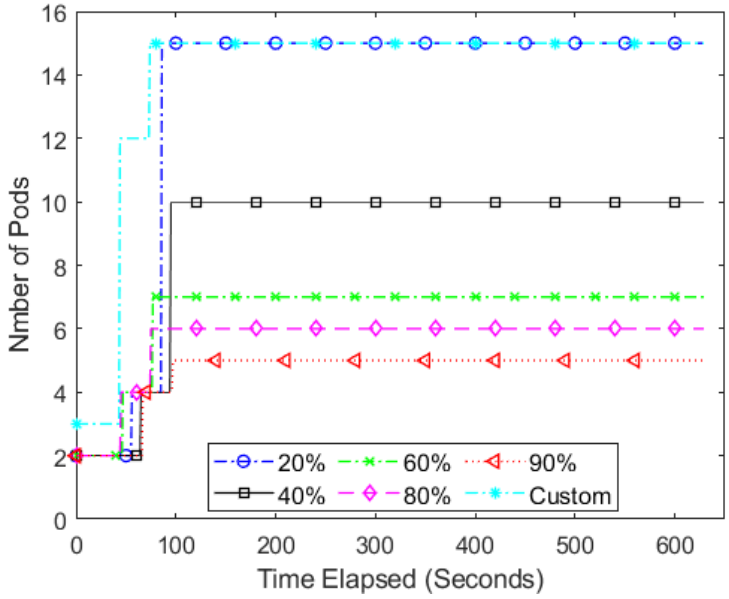


(b) Fluctuating Users

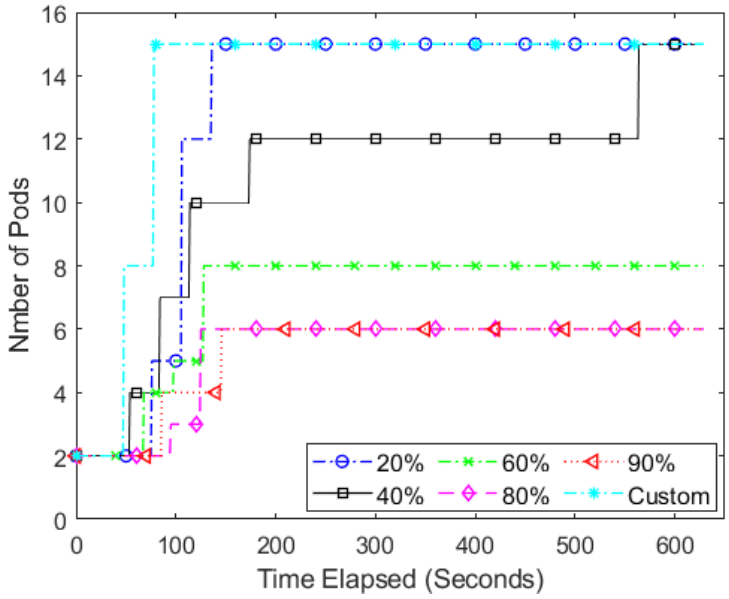
Figure 5.17: Response Time and Controller Output

a word pattern as input and matches the input pattern with 200,000 words from an English dictionary using a regular expression.

As can be seen from Figure 5.17 (a), for a constant number of users, the measured probe response time is always well above the setpoint response time. Therefore, the



(a) Constant Users



(b) Fluctuating Users

Figure 5.18: Number of Pods in Deployment

controller output (setpoint event loop lag) remains low and autoscaling is triggered. The number of pods quickly reaches up to 15, which is the maximum value set in the deployment manifest. As is evident from Figure 5.18 (a), the number of pods in the custom autoscaling solution grows as quickly as the one in CPU utilization-based

autoscaling when the threshold is set to 20%.

For a fluctuating number of users as well, as seen from the graph in Figure 5.17 (b), the measured probe response time is almost twice the setpoint response time in some cases. Similar to the case of a constant number of users, this causes the deployment to scale aggressively to 15 pods. However, the higher number of pods as seen in Figure 5.18 (b) does not translate into a better response time.

5.2.3.5 Controller Performance Summary

Overall, the control action taken by the controller, in response to the divergence from a desired SLO, has led to the appropriate autoscaling action. This is especially evident in the case of the custom workload and the static server, where the controller appropriately updates the setpoint event loop lag when the measured probe response time exceeds the setpoint, thereby triggering autoscaling.

In the custom workload, since it contains a combination of CPU bound tasks for both the event loop and the threadpool as well as I/O bound tasks in the form of queries to the Mongo database, the benchmarks produced varying results. When the number of users is less than 100, the custom implementation is on par with CPU utilization-based autoscalers, in spite of having a smaller number of pods. However, as the number of users increase, the CPU utilization-based autoscalers scale faster and consistently have a higher number of pods, thus producing better overall response times. Since the custom implementation is dependent on the measured probe response time and event loop lag, it scales less aggressively. Furthermore, when the `--horizontal-pod-autoscaler-downscale-stabilization` period of 5 minutes expires, the number of pods in deployment decreases, as the measured probe response is less than the setpoint response time, thereby affecting the overall response time.

In case of Static Server and Node Todo, the measured probe response times sel-

dom exceed the setpoint. Therefore, there are only a few instances where the PID controller needs to update the setpoint to the HPA controller.

In the case of Word Finder, using regular expressions impacts the performance in terms of the response times, as seen from Table 5.3. Since each user runs a computationally intensive task on the main thread, additional pods do not seem to make a significant difference in the measured probe response time, as seen from Figure 5.17. Although the number of pods in deployment is similar in both CPU based autoscaling as well as the custom implementation, the probe requests, which are also equally computationally intensive, exacerbate the problem as seen from the significantly-higher response times for custom implementation in Figure 5.9 and 5.10.

5.2.4 Rise Time, Settling Time, Overshoot and Stability

The rise time, settling time and overshoot observed in the closed-loop system built using the system's transfer function and the PID controller in MATLAB and Simulink are as follows:

- Rise Time: 7 seconds
- Settling Time: 21 seconds
- Overshoot: 0%

The aforementioned characteristics, however, were not found to be consistent across the various workloads and benchmark runs. This can be attributed to multiple factors:

- The discrepancy between the pod scheduling time taken by the Kubernetes scheduler to schedule a pod after autoscaling has been triggered.
- The difference between the period of the HPA controller (15 seconds) and the PID controller (5 seconds), which is the same as the Prometheus metric re-list

interval. This could cause a delay as the HPA controller does not immediately compare the new setpoint with the current values of the target metric.

In the case of the CPU-utilization based autoscaler, accurately instrumenting the rise time, settling time and the overshoot is challenging. Unlike the case of the custom implementation, a user does not possess the timestamp of the control action to calculate the aforementioned metrics. This would require a custom build of the Kubernetes codebase, which is not within the scope of this work. Furthermore, since the custom implementation uses an external supervisory controller to change the setpoint on the HPA controller, the characteristics of the two autoscaling implementations are not directly comparable.

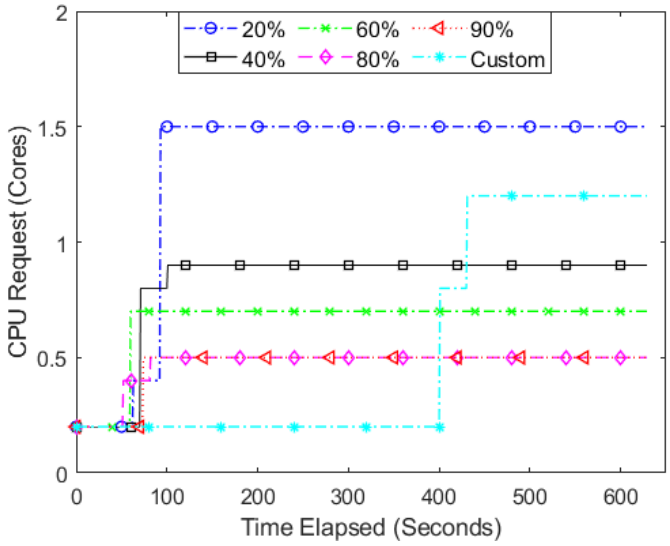
The stability of the closed-loop system is, however, the same as the default, CPU utilization-based autoscalers. This can be explained by the fact that the `--horizontal-pod-autoscaler-downscale-stabilization` flag is not modified in either case, which means that once a deployment is scaled up, it does not scale down for at least 5 minutes. This is a default behaviour in the HPA and is set this way to prevent the "flapping" number of pods in a deployment due to varying measured metrics [2].

5.2.5 CPU Requests

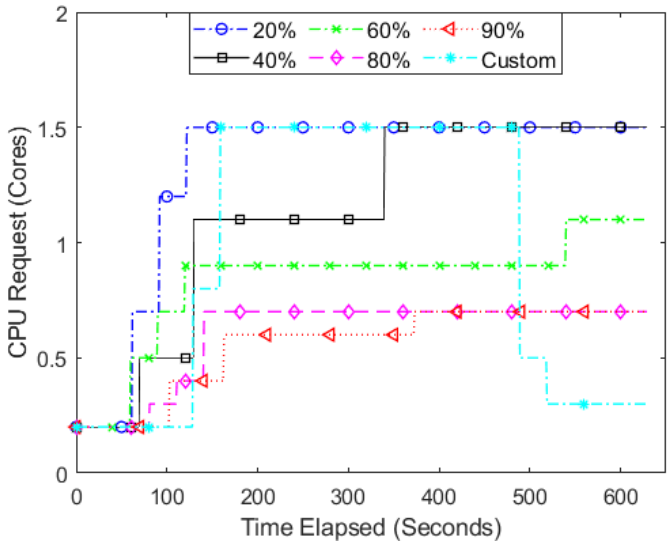
To evaluate the feasibility of the custom autoscaling implementation, we compare the CPU requests, under the same load, for both CPU utilization-based autoscalers, using different thresholds, and the custom implementation.

5.2.5.1 Custom Workload

As can be seen from Figure 5.19, for a constant number of users, the CPU requests for the custom implementation grows more slowly than that for the CPU utilization-based autoscalers. When compared to the best performing CPU utilization threshold, 40%, the custom implementation is a lot more conservative with respect to the CPU



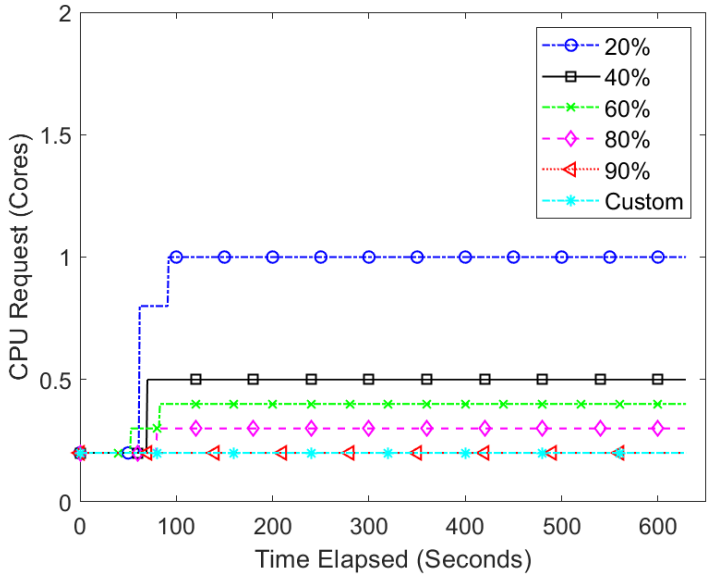
(a) Constant Users



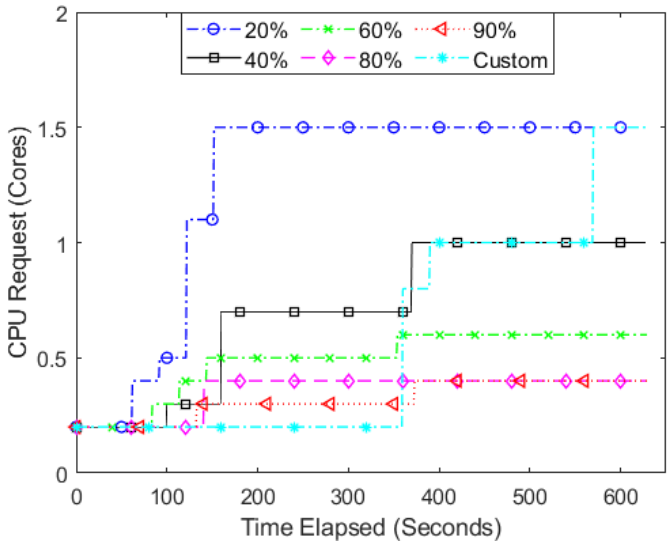
(b) Fluctuating Users

Figure 5.19: CPU Request for Custom Workload

request. It also has better 95^{th} percentile response times compared to other CPU thresholds with a higher CPU request. This conservative approach, however, is not replicated in the case of a fluctuating number of users. Although the CPU requests grows more slowly than that of CPU utilization-based autoscalers, it does reach the maximum allocatable value. However, once the downscaling is triggered due to the measured probe response time consistently being less than the setpoint response



(a) Constant Users



(b) Fluctuating Users

Figure 5.20: CPU Request for Static Server

time, the CPU requests plummets as well.

5.2.5.2 Static Server

For a constant number of users, as seen from Figure 5.20, the CPU requests remains at 200 millicores, the same as 90% CPU utilization threshold. Despite the lower number of pods and CPU request, the 95th percentile response time is $20ms$, which is

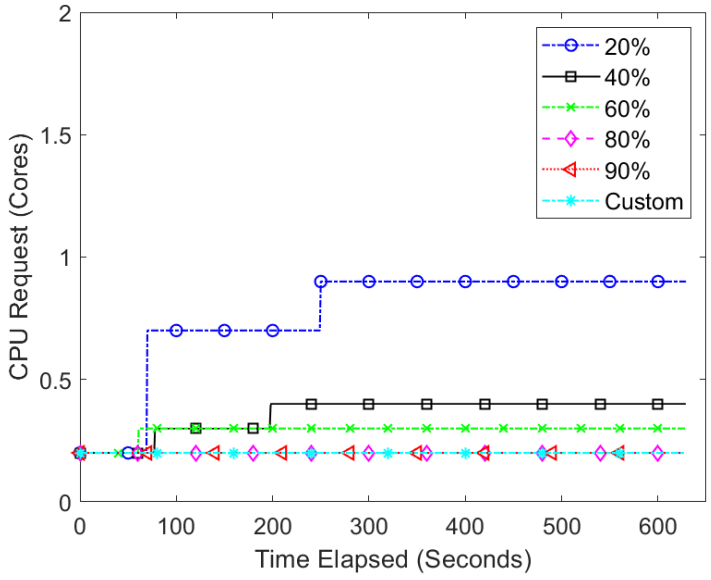
the lowest of all other baselines with high CPU requests. For a fluctuating number of users, the CPU requests remains low for large periods of the benchmark, but there are two instances where the number of pods and the CPU request grow drastically, which is comparable to that of lower CPU utilization thresholds. This conservative scaling behavior demonstrated by the custom implementation affects the overall benchmark result, with a 95^{th} percentile response time of $60ms$, which is greater than all the CPU utilization-based autoscalers.

5.2.5.3 Node Todo

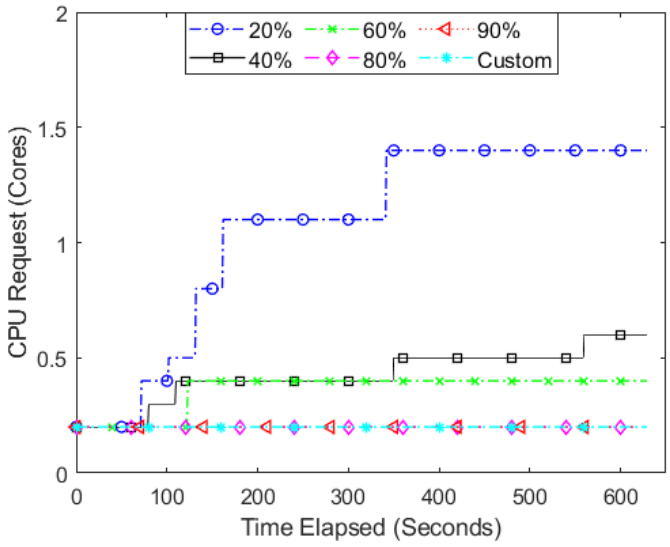
As can be seen from Figure 5.21, for Node Todo, the CPU requests remains low, at the minimum value of 200 millicores, for both a constant and fluctuating number of users. For a constant number of users, the CPU request for CPU utilization-based autoscalers with thresholds of 20% grows the fastest, followed by 40% and 60%. The CPU request for 80% and 90% thresholds is the same as the custom implementation. However, the higher number of pods and CPU requests for the lower CPU utilization thresholds do not translate into better response times, as elaborated in Section 5.2.3. The trend is similar for a fluctuating number of users. The CPU requests for lower CPU utilization thresholds is higher due to the increased number of pods. Therefore, given the overall response times achieved for both custom implementation and the CPU utilization-based autoscalers, it is safe to assume that the lower CPU utilization thresholds are more resource intensive than higher thresholds and the custom implementation.

5.2.5.4 Word Finder

As can be seen from Figure 5.22, for a constant number of users, the CPU requests for the custom implementation grows faster than CPU utilization-based autoscalers. Due to the high running time of each task, the measured probe response time is much



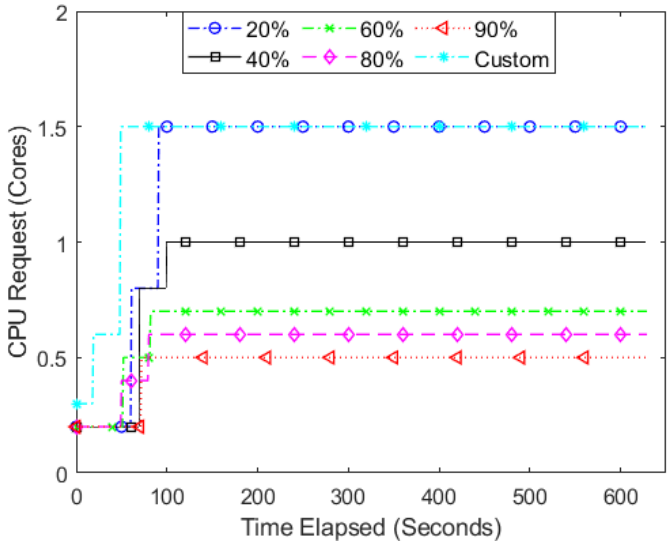
(a) Constant Users



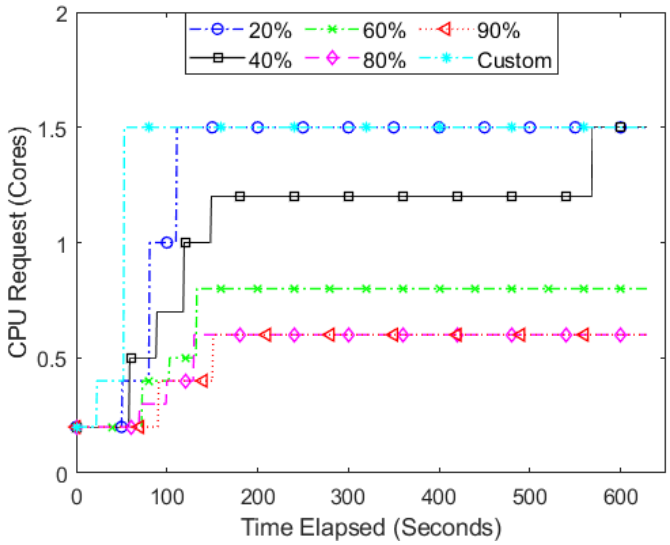
(b) Fluctuating Users

Figure 5.21: CPU Request for Node Todo

higher than the setpoint, thus forcing the PID controller to decrease the setpoint event loop lag on the HPA controller to a minimum value. This increases the number of pods drastically and thus, the CPU request as well. The CPU requests for other lower CPU utilization thresholds also increases eventually, but not as drastically as the custom implementation. This trend is replicated for a fluctuating number of users as well. The CPU requests for the custom implementation also grows faster than



(a) Constant Users



(b) Fluctuating Users

Figure 5.22: CPU Request for Word Finder

CPU utilization-based autoscalers, reaching the maximum value of 1500 millicores. The CPU requests for custom implementation is comparable to that of the CPU utilization-based autoscaler, when the threshold is set to 20%. The CPU requests for other higher CPU utilization thresholds, such as 60%, 80%, and 90% does not reach 1500 millicores, but still results in better overall response times, as described in Section 5.2.3

5.2.5.5 CPU Request Summary

From the findings described above and in Section 5.2.3, the CPU requests and the number of pods in deployment go hand in hand. A higher number of pods means a high request for CPU and other resources. Therefore, it is critical to have an autoscaling mechanism in place that does not cause unwarranted scaling actions. For instance, in case of Node Todo, under both a constant and fluctuating number of users, the CPU request for lower CPU utilization thresholds, such as 20%, 40%, and 60%, is higher than the custom implementation. However, the overall response time, which is a crucial service level metric, does not improve with the addition of resources. In this case, the custom autoscaling implementation performs better, in terms of both resource usage and fulfillment of SLOs, than the existing CPU-based one. However, a conservative scaling approach can also impact the performance, as seen in case of the Static Server and custom workload, for a fluctuating number of users.

5.3 Findings

To sum up, given the nature of the workloads tested in this research, and the results discussed in the sections above, we are able to make the following conclusions:
For a workload containing a mixture of both CPU bound and I/O bound tasks, such as the custom workload, the custom implementation is on par with the CPU utilization based autoscalers with different thresholds when the number of users is less than 100. However, as the number of users increases above 100, the CPU utilization-based autoscalers perform better than the custom implementation. Therefore, for a higher number of users, as explained in Section 5.2.3.1, CPU utilization-based autoscalers with lower thresholds yield better response times, as they are able to scale early and maintain higher number of pods for a longer period of time.

For workloads containing network I/O only, such as the Node Todo, both CPU utilization-based autoscalers, with higher thresholds, and the custom implementation offer similar performance, in terms of response times and resource usage. However, further stress testing such workloads could offer better insight as to which strategy delivers the best performance for systems that require an even higher throughput than the test cases in this research.

For workloads that involve serving static resources, the custom implementation is on par with the CPU utilization-based autoscalers when the number of users is less than 100. However, when the number of users increases to 200, CPU utilization-based autoscalers demonstrate better response times. Therefore, for systems that are intended to be used as a file server for a large number of concurrent users, CPU utilization-based autoscalers perform better, albeit with a higher resource usage.

For compute-intensive workloads, such as the word finder application, both the CPU utilization-based autoscalers and the custom implementation show unsatisfactory performance. This is in accordance with the design goals and the documentation of Node.js, which state that "for maximum scalability, the run time of tasks associated with each client should be small" [19]. Since for each request, there is a compute-intensive search involved in the event loop itself, using either scaling strategy does not ensure the satisfaction of response time SLOs. Therefore, such workloads should be avoided in Node.js. Leveraging multi-process or multi-thread techniques such as Child process or Worker threads could help improve the performance in this case; however, as elaborated above, having compute intensive tasks for each request is generally considered an anti-pattern in Node.js.

5.4 Threats to Validity

As stated in Section 5.2.4, the rise time and settling time varied across different experiment runs. Since we are implementing an external supervisory controller on top of the HPA controller, the synchronization issues with the period of the HPA controller and the PID controller meant that we were unable to consistently achieve the desired characteristics with the tuned gain parameters.

Furthermore, the evaluation of the custom implementation is done with a constant setpoint response time of 100ms. Therefore, we are unable to make claims about the performance when the setpoint response time is varied. Any future enhancements to this work can involve testing the custom implementation under varying setpoint response times.

Reliance of this work on MATLAB products, such as the system identification toolbox, Simulink, and the PID tuner application could pose further threat to its validity. Since accurately modeling a system and synthesizing a controller is a non-trivial task, we use the aforementioned applications to generate the transfer function model and the controller gain parameters. These tools may not be readily available for free, for other users to verify. In cases where the system identification toolbox does not generate an accurate model, the user might have to find alternate ways to generate the model parameters to use the PID tuner and Simulink.

Chapter 6

Conclusion and Future Work

Various methodologies have been explored for autoscaling computing instances to satisfy performance requirements. The existing literature focuses on either reactive scaling, which scales the number of computing instances when a certain threshold metric is reached, or predictive scaling, which is able to predict the future resource usage and scale ahead of time. We, however, focus on scaling a Kubernetes deployment based on a runtime-specific metric of Node.js. Additionally, instead of using a static setpoint, such as 70% CPU utilization for scaling, our implementation utilizes a dynamic setpoint for the HPA controller, based on the difference between the desired and measured SLO. To this end, the cluster is modeled using a state-of-the-art System Identification tool and a PID controller is designed to react to any divergence from a desired service level metric, by adaptively changing the setpoint on the HPA controller. We tested our approach on a Kubernetes cluster with the specification listed in Chapter 4 and containerized pods with the constraints listed in Section 5.1

6.1 Conclusion

As listed in the research questions in Section 4.1, using the custom metrics API in Kubernetes, we successfully introduced event loop lag to scale Node.js applications

in Kubernetes. Furthermore, we leveraged control-theoretic concepts to model the cluster and adaptively changed the setpoint event loop lag value in response to the divergence between the desired and the measured service level metric. As illustrated in the previous chapter, the response time metrics achieved using our custom autoscaling implementation differs with the type of workload, when compared against a CPU utilization-based autoscaler. Overall, for the four different workloads that we analyzed, the custom autoscaling implementation is on par with a CPU utilization-based autoscaler for a number of test cases, when the number of users is limited. The three instances where the custom autoscaling implementation performs significantly worse are: custom workload for a fluctuating number of users, Static Server for a fluctuating number of users, and the Word Finder for a fluctuating number of users, as explained in Section 5.2.3.5. In contrast, the instances where the custom autoscaling implementation performs on par with CPU utilization-based autoscaler, in terms of resource usage as well as service level metrics, in our test cases are: custom workload under a constant number of users, Node Todo for both custom and fluctuating users, and Static Server under a constant number of users. However, as seen from the results in the Word Finder application, selection of the probe request endpoint must be made properly so that the probe request does not interfere with the actual benchmarking process.

6.2 Possible Enhancements

This work serves as a "proof of concept" for leveraging control-theoretic concepts in autoscaling Kubernetes deployments. The methodology used in this research can be extended to other language runtimes as well. In particular, the process of modeling the Kubernetes cluster as a black-box and adaptively changing a setpoint metric, in the HPA controller, in response to the change in an SLO of interest, can be applicable

to several use cases. Among a multitude of improvements that are possible to this research, some notable ones include:

- Using a non-linear or state-space modeling technique to better capture the relationship between the process and control variables.
- Automating the re-synthesis of the system model and controller parameter estimation process in response to a model drift, e.g., when the relationship between the process and control variables changes.
- Tuning several other parameters in Kubernetes itself such as the downscale stabilization period, horizontal pod autoscaler sync period, and metrics collection interval in the prometheus adapter, for application specific use cases.
- Exploring the use of other Node.js specific metrics, such as the event loop utilization, number of active handles, and GC duration.

The overall planned architecture for the future is illustrated in Figure 6.1. It includes a three-tier adaptive supervisory architecture, in which blocks one and two exist within the current implementation, whereas block three is the proposed addition. It includes a model drift detecting mechanism that detects potential model drift based on a previous model and current input-output characteristics of the system. Furthermore, another component triggers the system identification phase again, in response to the model drift and updates the gain parameters of the controller based on the new model as well as the adaptation goals such as rise time, settling time, overshoot, etc., which can be changed at runtime. The possible implementation strategies are discussed below:

6.2.1 Detecting Model Drifts

Concept drift, in which the statistical properties of the response variable changes over time, is a common problem in machine learning. In this case, the goal is to revise

the model parameters in case of concept drift, so that the model remains valid. A general framework for concept drift detection includes sampling historical data and present data, measuring the dissimilarity between the two datasets and finally using a hypothesis test to calculate the statistical significance of the difference in the two datasets. For future enhancement, we propose to leverage an error-rate-based concept drift detection algorithm—the Drift Detection Method (DDM) [49]. Essentially, a landmark time window is set for drift detection, in which the starting point is the same for both historical data and present data, but the ending point expands to cover new data points for the present dataset. The DDM then estimates the model, including new data points, within the landmark window and calculates the online error rate. In the hypothesis test phase, the algorithm estimates the statistical significance of the error rate to indicate model drift.

6.2.2 System Identification and Controller Parameter Re-tuning

Since this research leverages the system identification toolbox available in MATLAB to generate the system model and SIMULINK to run simulations, updating the model and tuning the controller parameters requires manual intervention. Therefore, an automated mechanism could be built to trigger the system identification phase in response to the detected model drift and, subsequently, the controller tuning phase using a set of formally verifiable methods. While the Python Control Systems Library [30] does provide some functionalities for feedback control design, it lags behind the feature-rich MATLAB System Identification Toolbox and PID Controller Tuner App. Therefore, a system can be designed to programmatically implement some of the features available in MATLAB to fit this use case. Having a mechanism to adaptively re-synthesize the system model as well as tune controller parameters could help to increase the robustness of the presented autoscaling pipeline.

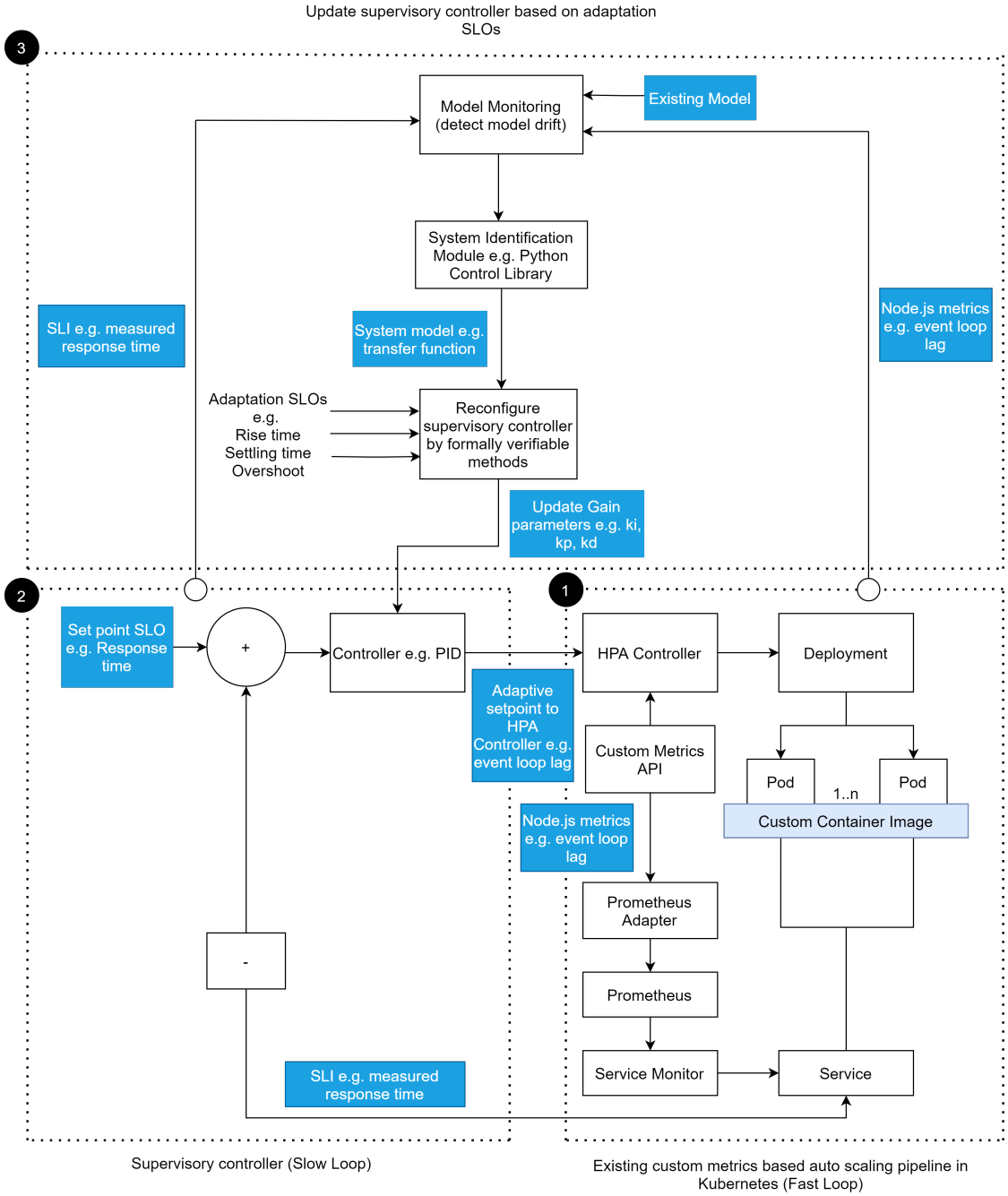


Figure 6.1: Possible Enhancement to the current architecture

References

[1] *Kubernetes overview*, https://kubernetes.io/docs/concepts/overview/components, Accessed: June, 2021.

[2] *Kubernetes horizontal pod autoscaler*, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, Accessed: June, 2020.

[3] *Prometheus adapter for Kubernetes metrics APIs*, https://github.com/kubernetes-sigs/prometheus-adapter, Accessed: August, 2020.

[4] *What is Kubernetes?*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes, Accessed: June, 2020.

[5] *Identify linear models using system identification app*, https://www.mathworks.com/help/ident/gs/identify-linear-models-using-the-gui.html, Accessed: October, 2020.

[6] *Locust documentation*, https://docs.locust.io/en/stable, Accessed: August, 2020.

[7] *Mongodb packaged by Bitnami*, https://github.com/bitnami/charts/tree/master/bitnami/mongodb, Accessed: June, 2021.

[8] *c-ares: library for asynchronous name resolves*, https://c-ares.org, Accessed: June, 2021.

[9] *About Calico*, https://docs.projectcalico.org/about/about-calico, Accessed: August, 2020.

[10] *Node.js dependencies*, https://nodejs.org/en/docs/meta/topics/dependencies, Accessed: January, 2020.

[11] *Matlab mathworks matlab simulink*, https://www.mathworks.com/products/matlab.html, 2021, Accessed: October, 2020.

[12] *Worker threads*, https://nodejs.org/api/worker_threads.html, Accessed: June, 2020.

[13] *Perf hooks*, https://nodejs.org/api/perf_hooks.html, Accessed: October, 2020.

[14] *Vertical pod autoscaling*, https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler, Accessed: November, 2021.

[15] *Basics of libuv*, http://docs.libuv.org/en/v1.x/guide/basics.html, 2021, Accessed: January, 2021.

[16] *Design overview*, http://docs.libuv.org/en/v1.x/design.html, 2021, Accessed: January, 2021.

[17] *About Node.js*, https://nodejs.org/en/about, Accessed: January, 2020.

[18] *The Node.js event loop, timers and process.nexttick*, https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick, Accessed: June, 2020.

[19] *Do not block the event loop (or the worker pool)*, https://nodejs.org/en/docs/guides/dont-block-the-event-loop, Accessed: June, 2020.

[20] *Express - Node.js web application framework*, https://expressjs.com, Accessed: October, 2020.

[21] *Docker overview*, https://docs.docker.com/get-started/overview, Accessed: July, 2021.

[22] *Understanding kubernetes objects*, https://kubernetes.io/docs/concepts/working-with-objectskubernetes-objects, Accessed: June, 2021.

[23] *Identify linear models using system identification app*, https://www.mathworks.com/help/ident/gs/identify-linear-models-using-the-gui.html, Accessed: July, 2021.

[24] *Transfer function models*, https://www.mathworks.com/help/ident/transfer-function-models.html, Accessed: July, 2021.

[25] *Simulink: Simulation and model-based design*, https://www.mathworks.com/help/simulink/index.html, Accessed: August, 2021.

[26] *Matlab: The language of technical computing*, https://www.mathworks.com/help/matlab/index.html, Accessed: August, 2021.

[27] *System identification overview*, https://www.mathworks.com/help/ident/gs/about-system-identification.html, Accessed: October, 2020.

[28] *Creating a cluster with Kubeadm*, https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm, Accessed: June, 2021.

[29] *Root locus design*, https://www.mathworks.com/help/control/ug/root-locus-design.html, Accessed: June, 2021.

[30] *Python control systems library*, https://python-control.readthedocs.io/en/0.9.0/index.html, Accessed: June, 2021.

[31] *llhttp: Port of http parser to llparse*, https://llhttp.org, Accessed: June, 2021.

[32] *OpenSSL*, https://www.openssl.org, Accessed: June, 2021.

[33] *REST APIs*, https://www.ibm.com/cloud/learn/rest-apis, Accessed: October, 2020.

[34] *Prometheus - monitoring system and time-series database*, https://prometheus.io, Accessed: August, 2020.

[35] *Microservices*, https://martinfowler.com/articles/microservices.html, Accessed: October, 2020.

[36] *Orchestration*, https://docs.docker.com/get-started/orchestration, Accessed: August, 2020.

[37] A. Andrieux, K. Czajkowski, Asit Dan, Kate Keahey, Hans Ludwig, Jim Pruyne, John Rofrano, Steven Tuecke, and Mantao Xu, *Web services agreement specification (ws-agreement)*, Global Grid Forum **2** (2004), 1–81.

[38] Kiam Heong Ang, G. Chong, and Yun Li, *PID control system analysis, design, and technology*, IEEE Transactions on Control Systems Technology **13** (2005), no. 4, 559–576.

[39] Cornel Barna, Marios Fokaefs, Marin Litoiu, Mark Shtern, and Joe Wigglesworth, *Cloud adaptation with control theory in industrial clouds*, 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW), IEEE, 2016, pp. 231–238.

[40] Stephen Burroughs, Helge Dickel, Matrin Van Zijl, Vladimir Podolskiy, Michael Gerndt, Robi Malik, and Panos Patros, *Towards autoscaling with guarantees on Kubernetes clusters*, IEEE International Conference on Autonomic Computing and Self-Organizing Systems, 2021.

[41] Oracle International Corporation, *System and method for automatically scaling a cluster based on metrics being monitored*, U.S. Patent US20200319935A1, Oct 2020.

[42] Wenzhi Cui, Daniel Richins, Yuhao Zhu, and Vijay Janapa Reddi, *Tail latency in Node.js: Energy efficient turbo boosting for long latency requests in event-driven web services*, Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE 2019, ACM Press, 2019, pp. 152–164.

[43] James C Davis, Eric R Williamson, and Dongyoon Lee, *A sense of time for Javascript and Node.js: First-class timeouts as a cure for event handler poisoning*, 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 343–359.

[44] Helge Dickel, *Control-theoretic approach to horizontal autoscaling of services orchestrated by Kubernetes*, Master's Thesis, Technical University Of Munich, 2020.

[45] Helge Dickel, Vladimir Podolskiy, and Michael Gerndt, *Evaluation of autoscaling metrics for (stateful) IoT gateways*, 2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA), IEEE, 2019, pp. 17–24.

[46] Brian Douglas, *Engineering media*, `https://engineeringmedia.com`, Accessed: October, 2020.

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

[48] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel, *Software engineering meets control theory*, 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE, pp. 71–82.

[49] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues, *Learning with drift detection*, Brazilian symposium on artificial intelligence, Springer, 2004, pp. 286–295.

[50] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu, *Serverless computing: One step forward, two steps back*, arXiv e-prints (2018), arXiv–1812.

[51] Philipp K. Janert, *Feedback control for computer systems*, O'Reilly Media, Inc., 2013.

[52] J.O. Kephart and D.M. Chess, *The vision of autonomic computing*, Computer **36** (2003), no. 1, 41–50.

[53] Abeer Abdel Khaleq and Ilkyeun Ra, *Agnostic approach for microservices autoscaling in cloud applications*, 2019 International Conference on Computational Science and Computational Intelligence (CSCI), IEEE, 2019, pp. 1411–1415.

[54] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez, *Brownout: building more robust cloud applications*, Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 700–711.

[55] Dhrub Kumar and Naveen Kumar Gondhi, *A QoS-based reactive autoscaler for cloud environment*, 2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS), IEEE, 2017, pp. 19–23.

[56] Kai Lei, Yining Ma, and Zhi Tan, *Performance comparison and evaluation of web development technologies in PHP, Python, and Node.js*, 2014 IEEE 17th International Conference on Computational Science and Engineering, 2014, pp. 661–668.

[57] Martina Maggio, Cristian Klein, and Karl-Erik Årzén, *Control strategies for predictable brownouts in cloud computing*, IFAC proceedings volumes **47** (2014), 689–694.

[58] Peter Mell and Tim Grance, *The nist definition of cloud computing*, `https://csrc.nist.gov/publications/detail/sp/800-145/final`, 2011.

[59] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B. Kent, *Insights into webassembly: Compilation performance and shared code caching in node.js*, Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (USA), CASCON '20, IBM Corp., 2020, p. 163–172.

[60] Trevor Norris and Roderick Vagg, *Utilization and load metrics for an event loop*, U.S. Patent US20200142802A1, May 2020.

[61] Panagiotis Patros, *Modeling and improving the performance of cloud systems*, Ph.D. thesis, University of New Brunswick., 2018.

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

[63] Panagiotis Patros, Kenneth B Kent, and Michael Dawson, *Why is garbage collection causing my service level objectives to fail?*, International Journal of Cloud Computing **7** (2018), no. 3-4, 282–322.

[64] Panos Patros, Kenneth B. Kent, and Michael Dawson, *SLO request modeling, reordering and scaling*, Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (USA), CASCON '17, IBM Corp., 2017, p. 180–191.

[65] Maria Patrou, Jacob M. Baird, Kenneth B. Kent, and Michael Dawson, *Software evaluation methodology of Node.js parallelism under variabilities in scalable systems*, 30th Annual International Conference on Computer Science and Software Engineering (EVOKE CASCON), 2020, p. 103–112.

[66] Vladimir Podolskiy, Michael Mayo, Abigail Koay, Michael Gerndt, and Panos Patros, *Maintaining SLOs of cloud-native applications via self-adaptive resource sharing*, 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE, 2019, pp. 72–81.

[67] Meina Song, Chengcheng Zhang, and E Haihong, *An auto scaling system for API gateway based on Kubernetes*, 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS), IEEE, 2018, pp. 109–112.

[68] Tian Ye, Xue Guangtao, Qian Shiyou, and Li Minglu, *An auto-scaling framework for containerized elastic applications*, 2017 3rd International Conference on Big Data Computing and Communications (BIGCOM), IEEE, 2017, pp. 422–430.

[69] Hanqing Zhao, Hyunwoo Lim, Muhammad Hanif, and Choonhwa Lee, *Predictive container auto-scaling for cloud-native applications*, 2019 International Con-

ference on Information and Communication Technology Convergence (ICTC), IEEE, 2019, pp. 1280–1282.

[70] Jiapeng Zhu, Panagiotis Patros, Kenneth B. Kent, and Michael Dawson, *Node.js scalability investigation in the cloud*, Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON), 2018, p. 201–212.

[71] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi, *Microarchitectural implications of event-driven server-side web applications*, 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015, pp. 762–774.

Appendix A

Appendix A

A.1 Dockerfile

```
FROM node:14.15.4-alpine as debug
WORKDIR /work/
COPY ./src/package.json /work/package.json
RUN npm install
RUN npm install -g nodemon
COPY ./src/ /work/src/
ENTRYPOINT [ "nodemon","--inspect=0.0.0.0","
./src/server.js" ]

FROM node:14.15.4-alpine as prod
WORKDIR /work/
COPY ./src/package.json /work/package.json
ENV NODE_ENV=production
RUN npm install --production
COPY ./src/ /work/
```

```
CMD node .
```

A.2 Deployment Configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-deploy
  labels:
    app: node-app
  annotations:
spec:
  selector:
    matchLabels:
      app: node-app
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: node-app
    spec:
      containers:
```

```
    - name: node-app
      image: casa/node:prom
      imagePullPolicy: Always
      ports:
      - containerPort: 5000
        name: web
      resources:
        requests:
          memory: "128Mi"
          cpu: "100m"
        limits:
          memory: "256Mi"
          cpu: "250m"
```

A.3 Service Configuration

```
apiVersion: v1
kind: Service
metadata:
    name: node-app
    labels:
    app: node-app
spec:
    type: LoadBalancer
    externalIPs:
    - xx.xx.xx.xx
    selector:
```

```
    app: node-app
    ports:
    - name: web
        port: 80
        targetPort: 5000
        protocol: TCP
```

A.4 Service Monitor

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: node-app
  namespace: monitoring
  labels:
    app: node-app
spec:
  selector:
    matchLabels:
      app: node-app
  endpoints:
  - port: web
    path: /metrics
    interval: 5s
  namespaceSelector:
    matchNames:
      - default
```

A.5 HPA Configuration

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
    name: node-app
spec:
    scaleTargetRef:
    # point the HPA at the sample application
    apiVersion: apps/v1
    kind: Deployment
    name: node-deploy
    # autoscale between 2 and 15 replicas
    minReplicas: 2
    maxReplicas: 15
    metrics:
    - type: Pods
    pods:
        metricName: nodejs_eventloop_lag_mean_seconds
        targetAverageValue: 64m
```

A.6 Load Generation

```
from locust.user.users import HttpUser
from locust import User, TaskSet, task, constant,
SequentialTaskSet
from locust.contrib.fasthttp import FastHttpUser
```

```
from locust import LoadTestShape
from locust import events
from locust.runners import MasterRunner
import time

@events.test_start.add_listener
def on_test_start(**kwargs):
    start_time = int(time.time())
    print("test start at time: ", start_time)

@events.test_stop.add_listener
def on_test_stop(**kwargs):
    end_time = int(time.time())
    print("test end at time: ", end_time)

class UserTasks(SequentialTaskSet):
  @task
  def io(self):
      self.client.get("/")
  @task
  def cpu(self):
      self.client.get("/cpu?limit=25")
  @task
  def threadpool(self):
      self.client.get("/threadpool?rounds=4000")

class WebsiteUser(HttpUser):
```

```
tasks = [UserTasks]
class StagesShape(LoadTestShape):
stages = [
  {"duration": 30, "users": 10, "spawn_rate": 10},
  {"duration": 60, "users": 15, "spawn_rate": 10},
  {"duration": 90, "users": 20, "spawn_rate": 10},
  {"duration": 120, "users": 25, "spawn_rate": 10},
  {"duration": 150, "users": 20, "spawn_rate": 10},
  {"duration": 180, "users": 15, "spawn_rate": 10},
  {"duration": 210, "users": 10, "spawn_rate": 10},

  {"duration": 240, "users": 10, "spawn_rate": 10},
  {"duration": 270, "users": 15, "spawn_rate": 10},
  {"duration": 300, "users": 20, "spawn_rate": 10},
  {"duration": 330, "users": 25, "spawn_rate": 10},
  {"duration": 360, "users": 20, "spawn_rate": 10},
  {"duration": 390, "users": 15, "spawn_rate": 10},
  {"duration": 420, "users": 10, "spawn_rate": 10},

  {"duration": 450, "users": 10, "spawn_rate": 10},
  {"duration": 480, "users": 15, "spawn_rate": 10},
  {"duration": 510, "users": 20, "spawn_rate": 10},
  {"duration": 540, "users": 25, "spawn_rate": 10},
  {"duration": 570, "users": 20, "spawn_rate": 10},
  {"duration": 600, "users": 15, "spawn_rate": 10},
  {"duration": 630, "users": 10, "spawn_rate": 10},
]
```

```
def tick(self):
    run_time = self.get_run_time()

    for stage in self.stages:
        if run_time < stage["duration"]:
            tick_data = (stage["users"], stage["spawn_rate"])
            return tick_data

    return None
```

Vita

Candidate's full name: Sujit Bhandari

University attended: Bachelor in Computer Engineering, Tribhuvan University, 2016

Publications:

1. S. Bhandari, M. Patrou, K.B. Kent, P. Patros, M. Dawson, J. Siu. "Node.js Event Loop Metric Based auto-scaling in Kubernetes", Poster, 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020.

2. S. Bhandari, M. Patrou, P. Patros, K. B. Kent, M. Dawson and J. Siu "Control theoretic auto scaling of Kubernetes cluster based on event loop metrics of Node.js", submitted to IBM for disclosure (P202102130US01), 12 pages, April 14, 2021; Decision: Search-2, May 2021; Filed on August 12, 2021.