Vrije Universiteit Amsterdam

Master Thesis

Controless: A serverless control plane for Kubernetes

Author: Debarghya Saha (2738094)

1st supervisor: Alexandru Iosup
daily supervisor: Matthijs Jansen
2nd reader: Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 20, 2024

Abstract

Cloud-native applications have gained massive popularity and adoption in the last decade due to their ease of scaling, development and deployment. Cloud native applications make use of containers to package applications, and container orchestration technologies are used to manage these containers. Kubernetes is the most popular container orchestration tool used by cloud native developers. The control plane is the brain of Kubernetes, which handles all the operations like receiving incoming requests, authentication and authorisation, scheduling requests to workers etc. It has a serverful architecture, meaning that control plane components, like the API server, datastore and scheduler, are created at the start of the lifetime of a cluster, and are always running throughout the lifetime of the cluster. If the workload on the cluster increases, and the CPU and memory resources allocated to the control plane cannot keep up with it, then the latency in the system increases. In contrast, in the past decade, there has been development of serverless computing, which replaces containers for lightweight functions, and handles resource allocation on the fly. Serverless functions are able to scale faster than containers, and without the need for infrastructure management which is offloaded to the cloud provider. In this work, we explore making the Kubernetes control plane using a serverless architecture, with the aim of enhancing scalability and elasticity of the control plane. We design, implement and evaluate a novel serverless control plane. We find that while it takes 400ms, compared to Kubernetes' 60ms, to serve a request, it introduces no additional latency when scaled up to 1000s of requests per second. In our tests, with 500 requests a second, Kubernetes reaches 100% CPU usage and starts to queue up requests, taking 19 minutes to complete a 3-minute test, while the serverless control plane maintains the same resource usage and latency even at 1000 requests per second.

Contents

1 Introduction **1**

1.1 Context . . . 2
1.2 Problem Statement . . . 4
1.3 Research Questions . . . 4
1.4 Research Methodology . . . 6
1.5 Thesis Contributions . . . 7
1.6 Plagiarism Declaration . . . 7

2 Background **9**

2.1 A Primer on Kubernetes . . . 9
2.2 The Kubernetes control plane . . . 10
2.3 A Primer on Serverless Computing . . . 11

3 Design of the Serverless Control Plane **13**

3.1 Requirements for a serverless control plane . . . 13
3.2 Overview of the design . . . 15
3.3 Design of the API server . . . 15
3.3.1 The Authentication function . . . 16
3.3.2 The Authorisation function . . . 17
3.3.3 The Write-Request-to-DB function . . . 18
3.3.4 The Send-To-Worker function . . . 18
3.4 Design of the datastore . . . 18
3.4.1 The Users table . . . 18
3.4.2 The Roles table . . . 19
3.4.3 The RoleBindings table . . . 19
3.4.4 The Requests table . . . 19
3.4.5 Access control for the datastore . . . 19

3.5 The Scheduler function 20

4 Implementation of our Serverless Control Plane 21

4.1 Implementation of the datastore 21

4.1.1 Users Table 22

4.1.2 Roles table 23

4.1.3 RoleBindings table 23

4.1.4 Requests table 24

4.2 Implementation of the API Server 24

4.2.1 Authentication Lambda 25

4.2.2 Authorisation Lambda 26

4.2.3 Initiate Request Lambda 27

4.2.4 Send Request to Worker Lambda 28

4.3 Implementation of the Scheduler 28

4.4 Alternatives Considered 28

4.4.1 Alternatives for the datastore 28

4.4.2 Alternatives for the architecture 29

5 Evaluation 31

5.1 Experimental Setup 32

5.2 Latency of requests 33

5.3 Throughput 35

5.4 Resource Usage 37

5.5 Cost 38

5.6 Limitations and Threat to Validity 38

5.7 Summary of Evaluation 39

6 Related Work 41

6.1 Scaling in Kubernetes 41

6.2 Kubernetes and Serverless Computing 42

7 Conclusion 43

7.1 Answering Research Questions 43

7.2 Limitations and Future Work 45

References 47

A Reproducibility **49**
A.1 Abstract 49
A.2 Artifact check-list (meta-information) 49
A.3 Description 49
A.3.1 How to access 49
A.3.2 Data sets 49
A.4 Installation 50
A.5 Evaluation and expected results 51
A.6 Experiment customization 51

B Kubernetes YAML definition examples **53**
B.1 Role definition YAML 53
B.2 Rolebinding definition YAML 53
B.3 Request definition YAML 54

1

Introduction

The advent of cloud-native technologies has revolutionized the way applications are developed, deployed, and managed. The Cloud Native Computing Foundation (CNCF) is an open source software foundation that promotes the adoption of cloud-native computing, and according to their 2023 annual survey, 76% of organisations are using cloud native applications in production (1). Cloud-native applications are software programs composed of multiple small, interdependent services known as microservices and designed specifically to run in cloud computing environments. Traditionally, developers built monolithic applications with a single block structure containing all the required functionalities. In contrast, the cloud-native approach involves breaking these functionalities into smaller microservices. This approach makes cloud-native applications more agile since these microservices operate independently and require minimal computing resources to run. The popularity of cloud-native applications is due to several advantages. Cloud-native applications are typically built using microservices that are packaged in containers and can easily scale up or down to handle varying levels of demand, ensuring responsiveness and availability. A good example of a cloud native application is Netflix, which has been able to handle a growth in viewership of three orders of magnitude since switching over to the cloud (2). Containers package an application and its dependencies together, ensuring that it runs reliably in different computing environments. For orchestrating containers, container orchestration technologies like Kubernetes are used. Of the 76% of organisations using cloud native applications, mentioned above, 84% use Kubernetes. Kubernetes is an open-source platform that automates the deployment, scaling, and operation of application containers. Kubernetes helps manage these containers by providing tools to easily deploy them across a cluster of computers, monitor their performance, and scale them up or down as needed. With Kubernetes, applications can scale rapidly to handle demand spikes. However, there

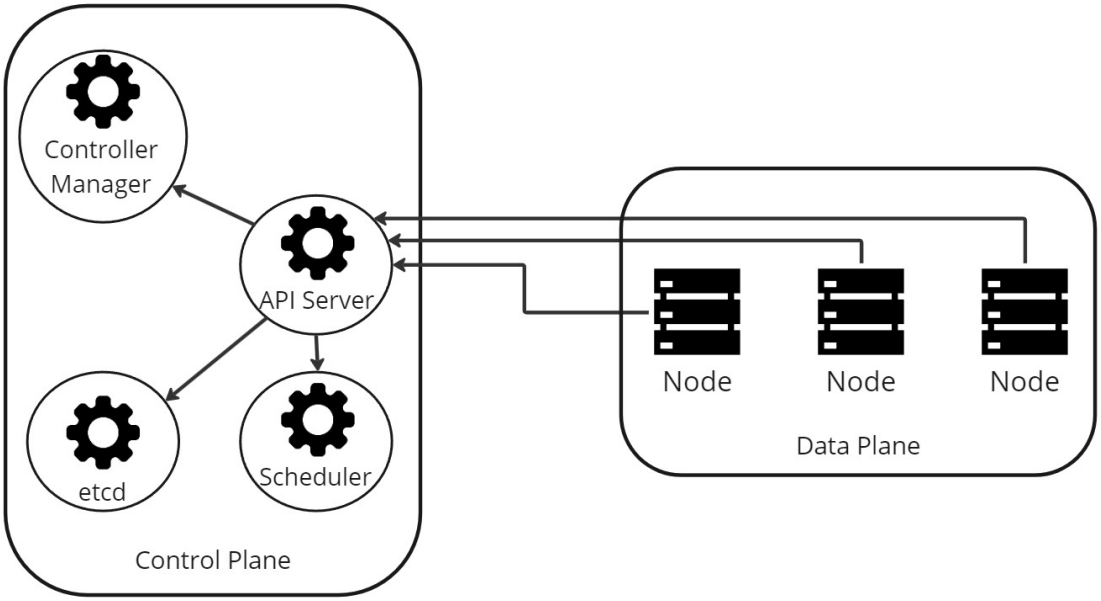

Figure 1.1: Kubernetes Architecture

is a bottleneck to this scaling ability, the control plane of Kubernetes. In a simple analogy, if the worker nodes of Kubernetes can be thought of as puppets, then the control plane is the puppet-master pulling the strings and ensuring each puppet does what it is supposed to. More and more puppets can be added, but at some point the puppet-master becomes overwhelmed. In this research, we propose making the Kubernetes control plane elastic and scalable in response to load.

1.1 Context

The architecture of Kubernetes can be broadly divided into two parts - the data plane and the control plane, or the puppets and the puppetmaster respectively, from the previous analogy. The architecture is illustrated in Figure 1.1. The data plane provides capacity such as CPU, memory, network, and storage so that the containers can run and connect to a network. The control plane is the brain of Kubernetes, which is responsible for managing the cluster and carries out essential tasks like authentication, scheduling workloads and managing the cluster. The control plane consists of four components: the API server, the database, the scheduler and the controller manager. In Kubernetes, each of these components are run in individual containers that start during the initialisation of the cluster and never shut down during the lifetime of the cluster. The data plane is able

to scale in response to demand, it can scale up in times of high demand, increasing its resources, and scale down, even to zero if there is no workload. However, the control plane is not so flexible because of the following reasons: it is unable to scale dynamically in response to demand. While it can be replicated for high availability, that is not the same as scaling dynamically in response to load. Control plane components are resource-intensive and require careful planning of CPU and memory resources. Dynamically scaling these components requires sophisticated resource management to ensure that new instances are provisioned with adequate resources without overloading existing nodes. Secondly, control plane components have elevated privileges and access to critical cluster data. Dynamically scaling these components requires robust security mechanisms to ensure that new instances are securely provisioned and configured without introducing vulnerabilities.

But is there a need to scale the control plane? A previous analysis of workload traces from production clusters in Microsoft Azure reveals peaks of up to 2000 VM creation requests per second (3). As the size of a cluster, or load on a cluster grows, the control plane needs to keep up with it. If the control plane cannot scale up fast enough, the latency in the cluster increases as new requests are queued up, in extreme cases leading to request timeouts. This is also evident by OpenAI's attempt to scale their clusters to a large number of nodes, wherein the load on the control plane was the main bottleneck (4, 5). Conversely, if the control plane cannot scale down efficiently in times of low demand, it leads to resource wastage. The aforementioned peaks in requests of 2000 VM creations per second drop to 0 almost instantly, which also highlights the importance of the speed of scaling, or elasticity, in the control plane. The current design of a somewhat rigid control plane was required at the time Kubernetes was developed, due to reasons highlighted above. However, since then there have been new developments in cloud computing, namely, serverless computing. Serverless computing allows for faster scaling, without worrying about the underlying servers and infrastructure. When using serverless services, the cloud provider automatically handles the provisioning, scaling, and management of the infrastructure needed to run the code. This approach not only simplifies the development process but also ensures that applications can scale seamlessly, even to zero, in response to demand. Users only pay for the actual computing resources consumed. Serverless computing is event-driven, meaning functions can be started up in response to specific events - like an incoming workload request. The architecture of Kubernetes is also event-driven, albeit with a serverful approach. The main benefit is that serverless functions are lightweight compared to VMs or containers, meaning that they can be invoked and shut down quicker. This intuitively

makes serverless functions an attractive option for fast scaling. This thesis explores the potential of integrating serverless principles with the Kubernetes control plane, transforming its components into serverless services that can scale up in milliseconds to handle spikes in workload, and also down to zero when idle to save resources and cost.

1.2 Problem Statement

The main problem that this thesis tackles is **how to increase the scalability and speed of scaling (elasticity) of the control plane.** Each of the control plane components have varying resource requirements as well as different security considerations. When the control plane is overloaded, for example due to many incoming requests, it could mean that either a single component is overloaded, or multiple components are overloaded. Simply scaling up all components is not a good solution. The interactions between the different components also needs to be taken into account, which components will write what data to the datastore, how will that data be modelled, what will be the access control mechanisms, and so on. Apart from the perspective of system design, there are also considerations to be made for the serverless platform. Which serverless platform to use, which database to use, and what security mechanisms are offered by the platform. Finally, there are performance considerations, using a serverless architecture will most likely add some latency compared to a serverful architecture, this is because a serverless system is not always running but rather constantly turning off and on with demand, unlike a serverful system which is always running.

1.3 Research Questions

The research questions are related to the design decisions of our system because we design and prototype a system in this work.

RQ1: How do we design the Kubernetes control plane as a serverless system? As described earlier, the control plane has many components, and to implement these as serverless functions some considerations are needed. For example, how will the functions interact with one another, what data needs to be passed between them, can each component be replaced with one function or is further granularity needed. Since there are components with varying functionalities, we split this question into three sub-questions, each dealing with a specific component of the control plane:

RQ1.1: How can the API server be implemented in a serverless system? The API server has several functions including managing requests, authentication and authorisation. It is also the only component that communicates directly with the database. Answering this research question will provide us with the main design considerations for a serverless API server, including: *how will a serverless API server receive user requests? What will be the authentication process? How will authorisation take place and what will be the scope of authorisation? How will the API server interact with the database?* To answer these questions, we take a look at how the API server is implemented in Kubernetes, with the aim of modelling our system as close to it as possible.

RQ1.2: How can the database be implemented in a serverless system? Answering this question will lead to design choices for the database, including: *what database to use? How will the data be modelled? What will be the access control mechanism for the database, and how will it interact with the API server?* To answer this, we look at how the data is stored and retrieved by etcd, which is the default database in a Kubernetes installation, and consider the cloud database options available for a serverless implementation.

RQ1.3: How can the scheduler be implemented in a serverless system? The scheduler in Kubernetes is always running and listening for new requests. In a serverless system, we cannot have an always running scheduler. Answering this question will help us in implementing a scheduler in our serverless system.

RQ2: How do we deploy and configure a serverless control plane in the cloud? This research question deals with design choices relating to the serverless platform, wherein we look at how the system designed in RQ1 will translate to a cloud ecosystem. We look at various services offered by the cloud platforms and consider how they fit into our system. This leads to considerations for security and authentication mechanisms and which cloud database service to use, for instance. Answering this question will lead to the system design from RQ1 becoming actual components and services running in the cloud.

RQ3: What is the latency impact when the control plane is serverless? When using a serverless system, containers for the serverless functions will be created and destroyed, sometimes we will have warm starts and other times cold starts, all this will affect the total latency of the system. It is therefore important to benchmark how much the latency is affected and what components take up the most latency. This will also enable

us to get some insight into possible areas for optimisation of latency. To answer this, we benchmark the latency at various stages in the lifecycle of a request in our system.

RQ4: What is the impact on cost of a serverless control plane?

Since the serverless functions making up the control plane in our system can scale to zero and have per millisecond billing granularity, we can expect the cost of running a serverless control plane to be less when compared to the traditional approach. The aim of this question is to measure the impact on cost.

1.4 Research Methodology

In this thesis, we address the research questions through a combination of conceptual, technical, and experimental approaches.

M1 (Quantitative, surveys): To answer **RQ1.1**, we go through the Kubernetes documentation and source code to find out exactly how the API server operates, how is the authentication and authorisation carried out, and how does it interact with the database. These are detailed in Section 3.3. To answer **RQ1.2**, we look at the database. There are various options available when choosing a cloud database - object stores, key-value stores, SQL and NoSQL databases to name a few. Since the goal of our system is to mimic Kubernetes as closely as possible, we go through the Kubernetes and etcd documentation to figure out the best option for a cloud database. To implement a database in our system, we need to know how is the Kubernetes data in etcd modelled, what are the tables and the keys used to carry out different functionalities of the API server. To figure this out, we run a Kubernetes installation in Continuum (6) and go into the etcd container to manually inspect the data that gets added to etcd by Kubernetes. These are detailed in section 3.4.

M2 (Design, abstraction, prototyping): After conceptualising our system, we look into how this system can be ported over to a serverless setting and implement a prototype system that has a serverless API server that takes requests from a user, authenticates the user and authorizes the request by querying the database, stores the request in the database, and then passes the request to a worker. This system is detailed in Chapter 4.

M3 (Experimental research, benchmarking): We benchmark the latency and resource usage of this system, to answer RQ3. This is detailed in Chapter 5.

Finally, for RQ4, we also evaluate the cost of running this serverless control plane.

1.5 Thesis Contributions

This thesis has the following contributions:

1. **(Conceptual)** Dissemination of system design, findings, and experiences:
   (a) **Design:** We design a serverless control plane and document our design choices.
2. **(Technical)** Development and evaluation:
   (a) **Implementation and validation** of the serverless control plane, running in AWS.
   (b) **Evaluating and benchmarking** the implemented system.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

2

Background

As mentioned previously, Kubernetes already has an event driven architecture where serverful components listen for events and react accordingly. Serverless computing is also event-driven, where serverless functions are deployed to carry out tasks in response to events. Intuitively, serverless is a good fit for Kubernetes as it can preserve the event driven approach with the benefit of having to only use compute resources when specific events occur, instead of always running and listening for events. In this chapter, we provide a comprehensive overview of topics related to the Kubernetes control plane and serverless computing, establishing a foundation for the subsequent content of this thesis.

2.1 A Primer on Kubernetes

Kubernetes, often abbreviated as K8s, is an open-source platform designed to automate the deployment, scaling, and operation of containerized applications. To understand Kubernetes, it's helpful to first understand a few basic concepts: containers, container orchestration, and the role Kubernetes plays. These are explained in the following paragraphs.

What Are Containers? Containers are a lightweight, portable way to package software. They bundle an application and its dependencies together, ensuring that the application runs reliably regardless of where it's deployed. Containers are isolated from one another and from the host system, providing a consistent environment across different stages of development and deployment.

Container Orchestration. When running a few containers, managing them manually is feasible. However, as the number of containers grows, managing them becomes increasingly complex and challenging. Tasks such as deployment, scaling, and monitoring require significant effort and resources, making manual management impractical for large-scale

environments. This is where container orchestration comes into play. Container orchestration automates the deployment, scaling, and management of containerized applications, ensuring that containers are efficiently allocated across nodes, services are seamlessly discovered and accessed, and resource usage is optimized.
Kubernetes is the most popular container orchestration platform. Originally developed by Google, it is now maintained by the Cloud Native Computing Foundation (CNCF). Some basic concepts of Kubernetes that are used in this thesis:

Cluster: The fundamental unit in Kubernetes, consisting of a set of worker machines called nodes. A cluster runs containerized applications.

Node: A single machine in the Kubernetes cluster, which can be a physical or virtual machine. Each node runs containerized applications managed by Kubernetes.

Pod: The smallest and simplest Kubernetes object. A pod represents a single instance of a running process in the cluster. Pods can contain one or more containers that share the same network namespace.

Namespace: A way to divide cluster resources between multiple users. Namespaces are intended for use in environments with many users spread across multiple teams or projects.

2.2 The Kubernetes control plane

The Kubernetes control plane is the central management entity responsible for maintaining the desired state of the cluster, orchestrating the containerized applications, and ensuring the system's health and stability. It comprises four key components:

The API Server. It exposes the Kubernetes API, acting as the central hub for communication between all components within the control plane and the cluster nodes. The API Server handles all requests, including operations such as creating, updating, deleting, and querying resources. In addition to that, the API server is also responsible for authentication and authorisation, ensuring that requests are from authenticated users that are authorized to perform specific actions. It also validates and modifies requests as they are processed, enforcing policies and ensuring consistency.

Datastore. Kubernetes manages all the cluster information via a datastore. While users are free to implement their own datastores, etcd is used in Kubernetes as default. etcd is a distributed key-value store used for storing all cluster data in a consistent and highly available manner. It holds the configuration data, state information, and metadata of the Kubernetes cluster. All other components interact with etcd to read or write necessary data, ensuring the desired state of the cluster is consistently maintained.

Scheduler. The Scheduler is responsible for assigning newly created pods to suitable nodes in the cluster based on resource availability and specific scheduling policies. It evaluates various factors such as CPU and memory to make informed decisions.

Controller Manager. The Controller Manager runs various controllers that regulate the state of the cluster by managing different aspects of the system. Controllers are control loops that watch the state of the cluster. Each controller is responsible for a specific function and continuously works to ensure that the actual state matches the desired state as defined in the configuration.

In conclusion, the control plane consists of 4 key components, the API Server handles communication and requests, etcd stores critical cluster data, the Scheduler assigns pods to nodes based on resource availability, and the Controller Manager ensures the cluster's state matches the desired configuration. Together, these components automate deployment, scaling, and operations, providing a robust and reliable foundation for cloud-native applications. This integration allows Kubernetes to efficiently manage both small and large-scale environments, making it a powerful tool for modern application development and operations.

2.3 A Primer on Serverless Computing

Serverless computing is a cloud computing execution model where the cloud provider dynamically manages the infrastructure, allowing developers to focus solely on writing code. Contrary to the name, serverless computing does involve servers, but the key distinction is that developers do not need to manage or provision these servers. This model abstracts away the complexities of server management, as shown in Figure 2.1, providing a highly scalable and cost-effective environment for running applications. Serverless platforms automatically scale applications up or down based on demand. Serverless applications are typically event-driven. They are triggered by specific events, called triggers, such as HTTP

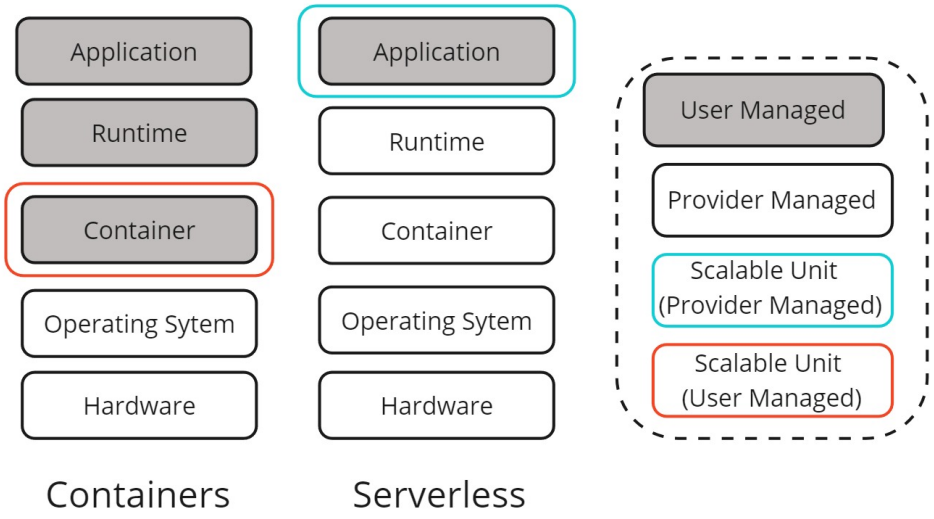

Figure 2.1: The difference between container and serverless deployments.

requests, database changes, or message queue events, allowing for highly responsive applications that execute only when needed. With serverless computing, you pay only for the compute resources you actually use. Billing is based on the number of requests and the execution time of your code, per millisecond, rather than on pre-allocated server instances. Containers take longer to set up initially than serverless functions because it is necessary to configure system settings, libraries, and so on. Once configured, containers take only a few seconds to deploy. But because serverless functions are smaller than container microservices and do not come bundled with system dependencies, they only take milliseconds to deploy. In addition to being quick to deploy, serverless functions are short-lived, typically executing in under a second as evidenced in (7). All these properties, in combination, make serverless a good fit for our objective of a load-based rapidly scalable control plane.

So far in this chapter we have presented background information on containers, the role of container orchestration technologies, some basic concepts of Kubernetes and a detailed look into its control plane. We also covered scaling mechanisms offered by Kubernetes and why said mechanisms are only applicable to the data plane and not the control plane. Lastly, we gave a background on serverless computing, its benefits and why it is a good fit for scaling the Kubernetes control plane. In the next chapter, we will present (our system) that makes the control plane serverless by implementing the control plane components as serverless functions in order to make an elastic control plane that can respond quickly to fluctuations in demand.

3

Design of the Serverless Control Plane

In this chapter, we address **RQ1.1**:*How can the API server be implemented in a serverless system?*, **RQ1.2**: *How can the database be implemented in a serverless system?* and **RQ1.3**: *How can the scheduler be implemented in a serverless system?* Together they cover the design decisions for the serverless control plane. To arrive at a design, we must first have a set of requirements, which will help guide our design decisions. In the following section, we formulate the requirements of our serverless system.

3.1 Requirements for a serverless control plane

To formulate requirements, we take a look at the design decisions of Kubernetes, with the aim of modelling our serverless system as close to Kubernetes as possible. This suggests three non-functional requirements (**NFR**) for our system:

> **NFR-1: Follow the design of the Kubernetes control plane as closely as possible.** Since with our system we are aiming to replace the serverful control plane of Kubernetes with a serverless one, following the design of the Kubernetes control plane will be helpful in insuring that our new system integrates seamlessly with Kubernetes. Even so, we are going from a serverful system to a serverless system, and thus there are bound to be some scenarios where we cannot completely follow the same design as the serverful counterpart. Due to this, we set our requirement to following the Kubernetes design as closely as possible, allowing us to come up with new design choices when required by the serverless approach.

NFR-2: Follow serverless design principles[1]. Going from a serverful design to a serverless design, we will have a different set of challenges, and so we follow the serverless design principles to help in designing our system.

NFR-3: Minimize latency overhead. The serverless nature of our system adds some latency that the serverful system does not have to deal with, in view of this we try to make design decisions keeping latency in mind.

Next we define the functional requirements of our system. To define the functional requirements, we focus on the serverless and cloud-native nature of our system. We model our requirements such that it favours a serverless design while also aligning with the Kubernetes design principles of security, extendability and ease of use. The functional requirements of our system are:

FR-1: Each serverless function should only be responsible for one specific functionality. The control plane has components of varying functionalities, as detailed earlier in 2.2, following **NFR-2** we split up the components of the control plane into functions such that each function is responsible for one distinct functionality.

FR-2: Allow requests to be submitted from outside the cloud environment. Following the Kubernetes approach, we want to keep our system open and loosely coupled, meaning that the data plane and control plane are separate entities. It should be possibl to integrate our control plane into any cluster. In view of this, we must allow the requests to the control plane to submitted from outside the cloud environment where the control plane is deployed. For example, if someone wishes to submit a request from a local machine to our system, it should be able to receive such a request, whether it is processed or not is subject to authentication.

FR-3: Only the API server can directly write to the datastore. This follows from the same design choice in Kubernetes. By centralizing write access to the datastore through the API server, Kubernetes ensures a consistent, secure, and reliable mechanism for managing the cluster state. This design choice simplifies validation, auditing, and recovery, while providing a robust foundation for scalable and extensible cluster operations.

Now that we have a set of functional and non-functional requirements to base our design on, the next sections outline the design of our system.

[1]AWS Serverless design principles: `https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/general-design-principles.html`

3.2 Overview of the design

In this section, we give a high level overview of the components of the system and their interactions, illustrated in Figure 3.1. We not include controllers in our design as in our prototype system we are only concerned about latency and scaling of requests in the control plane, and not the actual work done in the data plane. For this, we can use pod requests, and pods do not require a controller.

The API server and the scheduler are serverless functions that form a chain, similar to the flow of a request. First the request needs authentication, so first the authentication function is called, after authentication the request needs to have proper authorisation, so second the authorisation function is called. Next, on successful authorisation, the request needs to be logged in the database, which is handled by the third serverless function in the chain. After logging the request has to be scheduled, which is handled by the fourth serverless function and finally the request needs to arrive at the worker to which it is scheduled by the final fifth function in the chain. To carry out their respective tasks, these functions need to interact with the datastore by either reading from it or writing to it. The triggers to these functions are an API call, for the first function, and for the rest the next function is triggered from within the previous function, this is explained in the following subsections. We highlight that how the functions are triggered have no impact on the function of the system. They could all be triggered by APIs, by database events or from other functions, the only difference is in the latency of the system. Different triggers have different latencies. This design, and the design of each component, is discussed in detail in the following sections.

3.3 Design of the API server

This section deals with the design choices for the API server. Going through the Kubernetes documentation and source code, we found that the API server has 4 distinct functions that it needs to carry out upon receiving a request:

Authentication. The first step for the API server upon receiving a request is to authenticate that the request comes from a valid user in the system.

Authorisation. After successful authentication, the API server determines whether the user sending the request is allowed to carry out the functions in that request, for example, creating pods, deleting pods, listing information about nodes, and more.

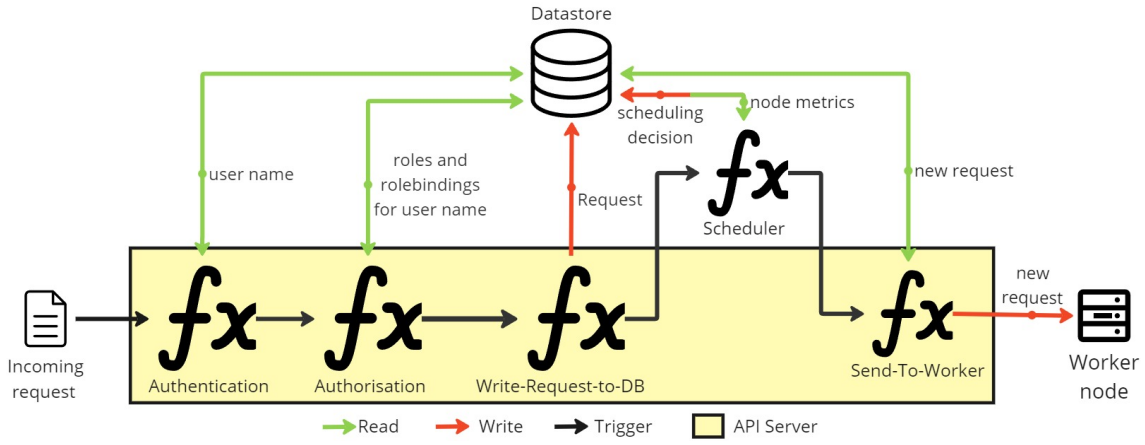

Figure 3.1: The overview of the design, with the components and their interactions.

Write request to the datastore. After successful authorisation, the request is stored in the datastore for logging and monitoring purposes.

Carry out the request. Depending on the request, this involves communicating with other components to carry out the requested function.

In a serverless system, each of these functions should be handled by a unique serverless function, as per the serverless design principles (**NFR-2, FR-1**). For this reason, we split the design for the API server into four parts, described below, each concerning one of the four functions mentioned above.

3.3.1 The Authentication function

This function is solely responsible for authentication of an incoming request. It should be the first function that is triggered upon receiving a request. Since a request can come from anywhere (**FR-2**), we make our first design choice:
DC-1: The authentication function must be trigged by a public-facing API call. This ensures that a request can be made to the API server from anywhere, from a local or a cloud environment. Whether this request is allowed into the system for further processing will depend on successful authentication. Aligning with our goal of designing our system as close to the Kubernetes design as possible, we look at the authentication methods offered by Kubernetes. Kubernetes offers the following methods for authenticating users:

Client Certificates. These are certificates signed by a certificate authority (CA). Any user that presents a valid certificate signed by the cluster's CA is considered authenticated.

Authenticating Proxy. This method is useful if you want to authenticate based on data that is stored in an external service, like Google for example, by configuring the service to export authentication information in HTTP headers.

Bearer Tokens. The API server reads bearer tokens from a file, or Authorization header with a value of *Bearer <token>* in an HTTP request.

The Authentication function must support at least one of these methods to authenticate the user submitting a request. When using Bearer Tokens, in Kubernetes, a static file stored on the API server is used to validate the supplied token. However, as our API server is serverless, we cannot store a file on it, leading us to our second design choice.
DC-2: Store the static token file on the datastore instead of the API server. This aligns with serverless design principles (**NFR-2**) and has the benefit of decoupling the token file from the API server, so that the token file can be modified without restarting the API server, unlike Kubernetes where the API server needs to restart for the changes to take effect. This decision does add some latency in the form of communication to the datastore, but that is negligible (detailed in Section **??**).

In Kubernetes, the control plane components must also authenticate themselves. This is required due to the open nature of Kubernetes, because anyone can add their own components and have them masquerade as legitimate components. However, when we implement the serverless control plane in the cloud, it comes with the cloud provider's security mechanisms. These could be, for instance, registered accounts or IAM roles, so it is guaranteed that only users with the proper credentials can modify our components. For example, if the serverless control plane runs in the AWS ecosystem, only valid AWS accounts and IAM roles can make changes to it. This leads us to our third design choice.
DC-3: The components of the serverless control plane need not authenticate themselves, because they are already covered under the cloud provider's security mechanisms. This also aligns with **NFR-3** as bypassing additional authentication will reduce latency. Upon successful authentication, the Authentication function must trigger the authorization function.

3.3.2 The Authorisation function

The Authorisation function is responsible for checking if the user that submitted the request is allowed to access the resources listed in the request. For example, if a user submits a request to create pods, then the Authorisation function must check if the user in question has the required permissions to create pods in the cluster. In Kubernetes, this is done

by querying the datastore to check if the appropriate Roles and RoleBindings exist (the datastore is detailed later in Section 3.4). If the required Roles and RoleBindings for the user in question are present, then the authorisation is successful. Upon successful authorisation, the Authorisation function triggers the WriteRequest function.

3.3.3 The Write-Request-to-DB function

Now that the incoming request is authenticated and authorised, it can actually be worked on. The next step is to write the request to the datastore. The WriteRequest function is responsible for this action. It simply takes the request and writes it to the datastore. After the write is completed, the Write-Request-to-DB function triggers the Scheduler function (detailed in Section 3.5).

3.3.4 The Send-To-Worker function

This is the last step for an incoming request. So far, the request has been authenticated, authorised, stored in the datastore and scheduled to a worker. The last step is to send the request to the appropriate worker. The Send-To-Worker function takes a scheduled request and sends it to the worker responsible for it.

This concludes the design for the serverless API server. In summary, the serverless API server consists of four serverless functions, each responsible for one of the tasks of authentication, authorisation, writing the request to the datastore, scheduling the new request and sending the request to a worker.

3.4 Design of the datastore

Kubernetes requires a persistent key-value store, leading to the third design choice:
DC-4: The datastore used must be a persistent, key-value store. Depending on the serverless provider, various options for a cloud key-value datastore are available, for example, DynamoDB in AWS, Firestore in Google Cloud and CosmosDB in Azure. The choice of datastore is trivial, however the data model is of importance. Below, we detail the data model of the datastore that facilitates the various functions of the API server described in the previous section, this is also summarised in Table 3.4.5.

3.4.1 The Users table

The datastore has a table that contains the users' data. This table serves also as the static token file as required by **DC-1**, outlined previously in Section 3.3.1, and is used

for authentication and authorisation. In Kubernetes, the token file is a csv file containing at minimum these 3 fields: *token, user name, user uid*. The *token* field holds the unique token value that is sent to the API server for authentication, the *user* field holds the user name, used for authorization, and the *uid* field contains a unique ID for each user. The users table must also contain these three fields.

3.4.2 The Roles table

This table in the datastore contains all the roles defined for the cluster.

3.4.3 The RoleBindings table

As per the Kubernetes design, Roles are not useful by themselves, they also need corresponding RoleBindings, and so we define the RoleBindings table in the datastore. Whenever the API server needs to authorise a request, it first queries the Users table to get the *user name* and then queries the RoleBindings table to check if appropriate RoleBindings exist for the user, finally it queries the Roles table to check if the user has the required permissions to carry out the request. Together, the Users, Roles and RoleBindings tables facilitate the authorisation process for the API server.

3.4.4 The Requests table

This table stores the requests that are authorised by the API server. Additionally, it also stores the scheduling decision for the request along with the request itself.

3.4.5 Access control for the datastore

Following **FR-3**, only the API server functions should communicate with the datastore. Out of the four functions that make up the API server in our system, only two write to the datastore, the rest of them only read from the datastore. And so, the datastore should only have the following interactions:

READ Allowed for Authentication, Authorisation, Scheduler and Send-To-Worker functions.

WRITE Allowed for Write-Request-to-DB, Scheduler functions.

DENY Everything else.

This concludes the design for the datastore, next we detail the design of the scheduler.

Table	Accessed by function	Type of access
Users	Authentication, Authorisation	Read
Roles	Authorisation	Read
RoleBindings	Authorisation	Read
Requests	Write-to-DB, Scheduler	Write

Table 3.1: Access scope of the tables in the datastore.

3.5 The Scheduler function

The scheduler function is responsible for scheduling newly received requests. In Kubernetes, the scheduler is always running and listening for new requests, however we cannot have a always running scheduler in a serverless system. In serverless, the scheduler has to be triggered by an event. If we look at Kubernetes, the scheduler is already triggered by an event, i.e a new request, the only difference is that in Kubernetes the scheduler is actively running and listening for this event. Following this, we can intuitively design a serverless scheduler to be triggered by an incoming request. We do not go into detail about the scheduler in this section because any form of scheduler can be implemented in this function following the basic design of receiving a new request, getting node metrics, running a scheduling algorithm based on the received metrics and writing the scheduling decision. The actual scheduler is not of substance here, apart from the fact that it is implemented in a serverless function that gets triggered by new requests.

In this chapter, we have outlined the design of a serverless control plane for Kubernetes. The components of the design include the API server (formed by a group of serverless functions), a persistent key-value data store and the scheduler (also a serverless function). The serverless functions interact with the datastore to carry out authentication, authorisation, logging and scheduling of an incoming request.

4

Implementation of our Serverless Control Plane

In this section, we answer **RQ2: How do we deploy and configure a serverless control plane in the cloud?** This section deals with the implementation of the design proposed in Chapter 3 by running the components of the serverless control plane as services in the cloud. We note that the design proposed previously is agnostic to cloud providers, and for our implementation we have chosen AWS. The components of the system and their interactions are shown in Figure 4.1. We begin with the implementation of the datastore (Section 4.1), followed by the implementation of the API server (Section 4.2) and then the scheduler (Section 4.3). At the end, we also briefly talk about the alternatives considered during implementation and why we reject the alternatives (Section 4.4).

4.1 Implementation of the datastore

For our datastore, we use AWS DynamoDB. As outlined in the design, the datastore must have the following tables:

1. Users table
2. Roles table
3. RoleBindings table
4. Requests table

These tables are created in DynamoDB. The serverless functions can read and write to these tables. Next, we describe the implementation of these tables, also shown in Figure 4.2.

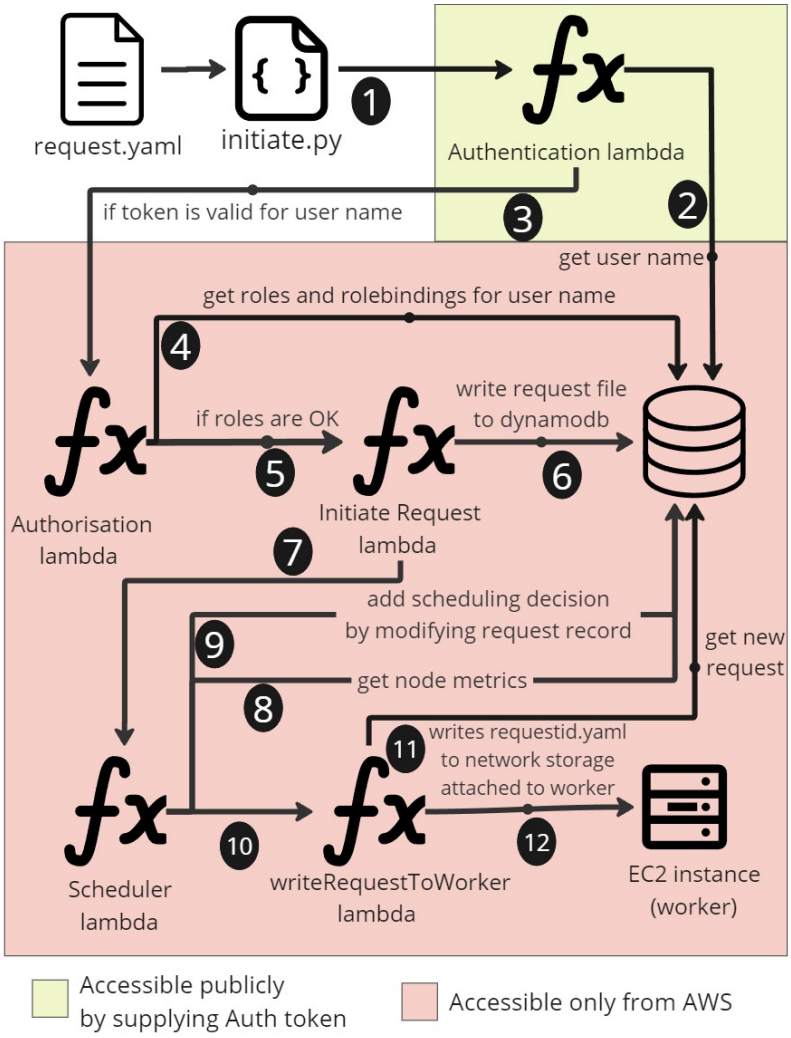

Figure 4.1: The components in our system and their interactions.

4.1.1 Users Table

Users table contains three fields *token, user name* and *user uid*, with the *token* field serving as the primary key for this table. We choose *token* as primary key, instead of *user uid* which could also act as primary key. This decision was made because the *token* is supplied with a request to the API server and the API server can then perform a look-up for the user name during authorisation using *token* easily, instead of having to use the *user uid* to find the matching *token* and then the user name.

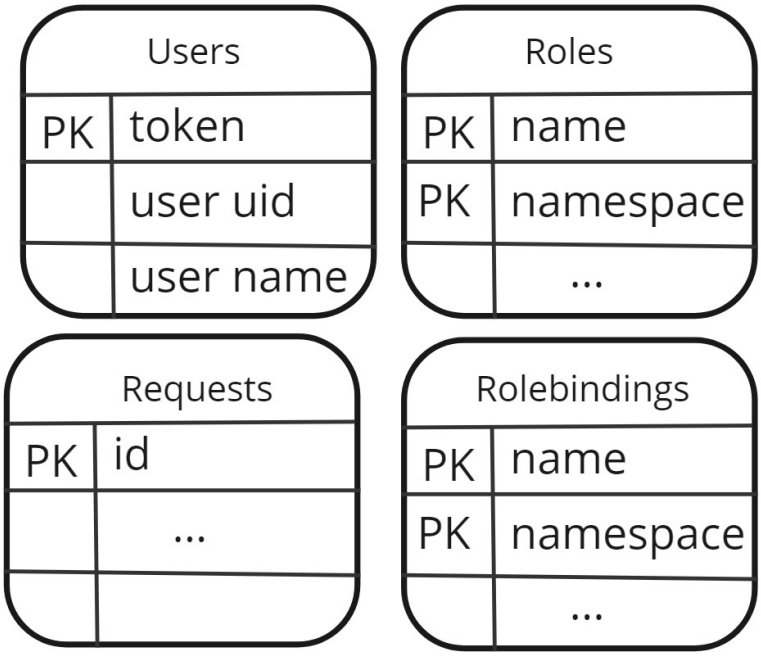

Figure 4.2: The structure of the tables in DynamoDB. PK denotes the primary key. A composite key (primary key made of multiple fields) is represented by multiple PKs in the table. Only the compulsory fields are shown, and '...' denotes that it can have any other fields, depending on the requirement.

4.1.2 Roles table

The Roles table contains all the roles defined for the cluster. They are exactly the same as Kubernetes YAML role definitions, an example of which can be found in Appendix B.1.

All the keys in the YAML act as keys in DynamoDB, this enables copy-pasting a standard Kubernetes role definition into the datastore, without any modifications. The data model for this table is slightly more complex than that of the User table. In Kubernetes, Role names are not unique, so *name* cannot serve as the primary key by itself. However, Role names are unique per namespace, and so both the fields *name* and *namespace* together make the composite key for this table.

4.1.3 RoleBindings table

Roles are not useful by themselves, they also need corresponding RoleBindings, and so we define the RoleBindings table to store the rolebindings. Again following Kubernetes convention, the structure of the RoleBindings table exactly matches that of a RoleBindings

definition YAML file, an example of which can be found in Appendix B.2.

Just as the Roles table, all the keys in this YAML are keys in the datastore for the RoleBindings table, and *namespace* together make the composite key for this table.

4.1.4 Requests table

The requests table is used to store the authorised requests. The structure of this table follows the structure of a request definition YAML in Kubernetes, an example of which can be found in Appendix B.3.

We make a slight change to the data when storing requests in the datastore. First, to uniquely identify each request, an *id* field is added to each request. This ID is generated from the Write-Request-to-DB function, which is responsible for writing the request to the datastore. The ID is the request ID of that particular invocation of the function. Secondly, we add the scheduling decision to the request in the *worker* field. So, after writing the request to the datastore, it looks like this:

```
id: 1cf0b12f-220a-4fcf-b672-716857a2668e
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:1.14.2
    name: nginx
    ports:
    - containerPort: 80
worker: 1
```

The *id* field serves as the primary key for this table.

This concludes the implementation of the datastore in DynamoDB. In the next section, we detail the implementation of the API server and how the different functions interact with the data in the datastore.

4.2 Implementation of the API Server

For the API server, AWS Lambda is used. The API server consists of four lambdas, one each for authentication, authorisation, initiating request and writing request to a worker.

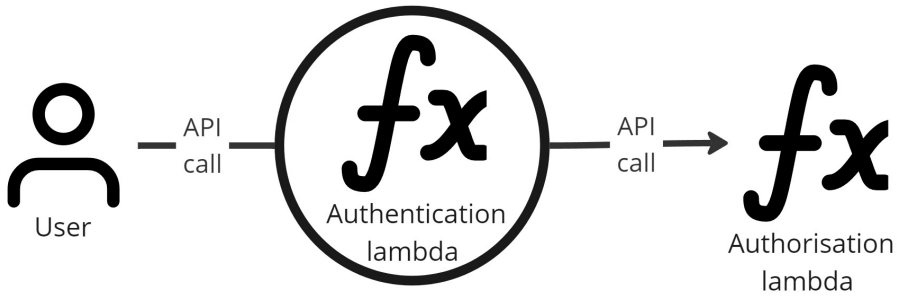

Figure 4.3: The Authentication function implemented as a lambda authoriser for the API call to the Authorisation function. The user makes an API call to the Authorisation function, and AWS calls the authoriser to authenticate the API call. If the authentication is successful, the API call is allowed.

These are described in detail in the following sections. The lambdas are triggered directly from the previous lambda in the chain, unless otherwise stated, such as in the case of the Authentication lambda, described below. This choice of trigger arises from **NFR-3:** *Minimize latency overhead.* because calling the next lambdas directly from the previous lambdas has the lowest latency as it does not have to rely on external events and can be triggered as soon as the previous lambda is finished.

4.2.1 Authentication Lambda

The basic function of the Authentication lambda is to take a post request, verify the user and pass the request to the Authorisation lambda on successful authentication. For our implementation, we have chosen to authenticate the users by using bearer tokens, which means a token has to be supplied in the HTTP header, along with the request. Following design choice **DC-1:** *The authentication function must be trigged by a public-facing API call*, the lambda needs to be triggered by an API call. AWS API Gateway is a service that allows interaction with other AWS services via API calls. We use the API Gateway to make the API call to our lambda, but we also need to authenticate calls to the API to make sure only legitimate users are allowed access. AWS offers many mechanisms to authenticate API calls to the API Gateway service, like RBAC policies and IAM Roles. It also offers authentication by Lambda Authorisers, which is a perfect fit for our use case. Lambda authorizers are Lambda functions that control access to REST API methods using bearer token authentication. Lambda authorizers are used to control who can invoke REST API methods. Since we are already using token based access control for authentication, making the Authentication function into a Lambda authoriser is best.

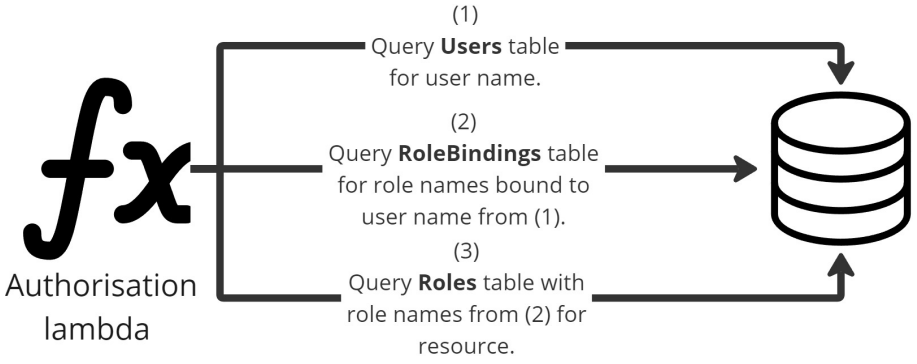

Figure 4.4: The interactions of the Authorisation function with the DB.

For that we need to invoke the next function, i.e the function that would be invoked by the Authentication function, with a public API call. In this case, the next function is the Authorisation function. So in our implementation, the user actually sends an API request which triggers the Authorisation function, but because the Authentication function is acting as the authoriser for this API call, AWS actually calls the Authentication function first, and if authentication is successful, the API call is allowed and the Authorisation function is triggered. This process is illustrated in Figure 4.3.

The Authentication lambda checks token in the HTTP header supplied with the request, if it matches the value stored in the datastore, it allows the API call to be made. Upon a successful API call, the Authorisation lambda is triggered.

4.2.2 Authorisation Lambda

The trigger for this lambda is an API call with a POST request. The POST request contains the Kubernetes request YAML. Upon being triggered, this lambda first gets the user name of the user making the request. It can do so by querying the Users table in DynamoDB with the token supplied in the request. After getting the user, it then queries the RoleBindings table to check if any rolebindings exist for the user: If rolebindings exist, then it needs to check if the roles in the rolebindings allow access to the resource specified in the request. For example, if a user requests to create pods, the Authorisation lambda needs to check if roles relating to the resource type 'pods' exist for the user, and because in Kubernetes roles alone mean nothing without a corresponding rolebinding, the lambda also has to check for appropriate rolebindings. So, after it gets rolebindings for a user, it needs to check the corresponsing roles from the Roles table: If the roles for the resource type specified in the request exist, the request is allowed. These interactions of this lambda

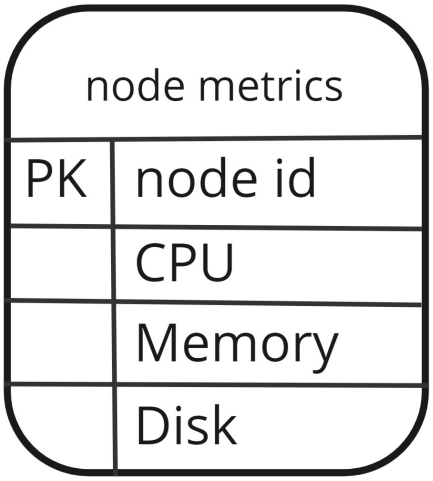

Figure 4.5: The node metrics table that is queried by the scheduler to get information about the nodes in the cluster.

and the datastore are depicted in Figure 4.4. If the request is allowed, this lambda will then trigger the next lambda, which is the Initiate Request lambda. If not allowed, then it prints a message saying the specified user is not allowed to access this resource. After the authorisation process is completed successfully, this lambda triggers the Initiate Request lambda and passes the request file to it.

4.2.3 Initiate Request Lambda

This lambda writes the authorised request to the datastore and after the write is complete, triggers the scheduler lambda. The input to this lambda is the request file. It adds an *id* to the request, which is equal to the request ID for this particular invocation of this lambda. And also adds a *worker* field with a value of '-1' to denote an unscheduled request. After modifying the request with the *id* and *worker* fields, it writes it to the Requests table in DynamoDB. After the write, it triggers the Scheduler lambda. Here we make the decision to trigger the scheduler from this lambda, instead of from the database write event, which was our choice initially. This is because, in AWS, DynamoDB triggers to lambda functions are actually event streams which are polled at given intervals of time, and this leads to additional latency that is avoided when invoking the lambda directly from the previous lambda in the chain. This also follows the non function design requirement **NFR-3:** *Minimize latency overhead.* This lambda triggers the Scheduler lambda, detailed in Section 4.3, however, we next describe the Write-Request-to-DB lambda which is triggered after

the scheduler lambda, since it is a part of the API server lambdas and fits better in this section.

4.2.4 Send Request to Worker Lambda

This lambda reads the newly written and scheduled request from DynamoDB, which was written to the DB by the Initiate Request lambda previously, and sends the request over to the concerned worker node. Since the request has the ID of the worker that it is scheduled to, this lambda can simply read that and figure out where to send the request.

4.3 Implementation of the Scheduler

In our implementation of the scheduler lambda, it pulls metrics (CPU, memory and disk usage) for the nodes in the cluster from a table in the datastore. The structure of this table is illustrated in Figure 4.5. The scheduling algorithm checks for the node with the lowest CPU usage and schedules the request on to that node. After making the scheduling decision, the scheduler lambda updates the *worker* field in the request with *node-id* and writes it to the requests table in the datastore. In a real world scenario, the node metrics would be pulled from a monitoring service running in the cluster. However, as we are not concerned with the data plane in our system, we have substituted the node metrics server with a table in the datastore.

4.4 Alternatives Considered

In this section we discuss some alternatives for system components like the datastore, and the architecture of the system that we considered during implementation. We also weigh into the pros and cons of each alternative and why we decided to not go with the alternatives for our implementation.

4.4.1 Alternatives for the datastore

Since we implemented our system on AWS we had two options for a persistent key-value store: DynamoDB and S3. S3 is mainly designed as an object store, to store large volumes of unstructured data. It is useful for high-throughput data, but unsuitable for low latency targets, unlike DynamoDB which was made for low latency. Unlike S3, DynamoDB is able to scale on demand, which is a major requirement of our system. DynamoDB also has the ability to create indexes on items, which we used for the Roles and RoleBindings table.

In S3, instead of creating indexes on the namespace, we could have used multiple buckets, each bucket acting as a namespace, but it adds unnecessary complexity. Lastly, we have to factor in cost. S3 is significantly more expensive compared to DynamoDB, and in S3 the majority of the cost comes from accessing the data and not storing the data. Since in our system, every request produces multiple accesses to the data, S3 is not a good choice. Overall, due to the unpredictable scaling, high number of accesses to the data and the model of the data, S3 was not a suitable datastore for our system.

4.4.2 Alternatives for the architecture

While implementing the system, we considered alternatives to triggering the lambdas compared to the direct from lambda trigger that we have in our final implementation. Mainly, we considered triggering the Scheduler and Send-to-Worker lambdas from WRITE and MODIFY events in DynamoDB. This design is depicted in Figure 4.6, this is a simplified figure, but the main point of interest is triggers to the Scheduler and Send Request to Worker lambdas. Intuitively, this design makes sense, as the lambdas only need to be triggered by a database event. However, compared to triggering directly from the previous lambda, this method added a latency between 200ms to 500ms. This seems to be mainly because the DynamoDB triggers are not kicked into action immediately on changes to the database, but at set intervals of time by batching all events in that time interval together. Due to this reason, we did not go with this architecture.

With this we conclude the implementation section. In this section, we discussed our implementation of the system proposed in Chapter 3 within the AWS ecosystem. In summary, we implemented a serverless control plane within AWS. The API server was implemented as a group of four lambdas, each handling one of the following functions of authentication, authorisation, writing requests to the datastore and sending request to the concerned worker. The lambdas are executed one after the other, forming a chain, where the next lambda is triggered by the previous lambda in the chain, except for the authentication lambda which is the first lambda in the chain and trigged by the AWS API Gateway as an authoriser to the API call that triggers the authorisation lambda. The datastore was implemented in DynamoDB consisting of four tables - users table, requests table, roles table and rolebindings tables. Finally, the scheduler is implemented as another lambda that retrieves cluster state from a table in DynamoDB and makes a scheduling decision.

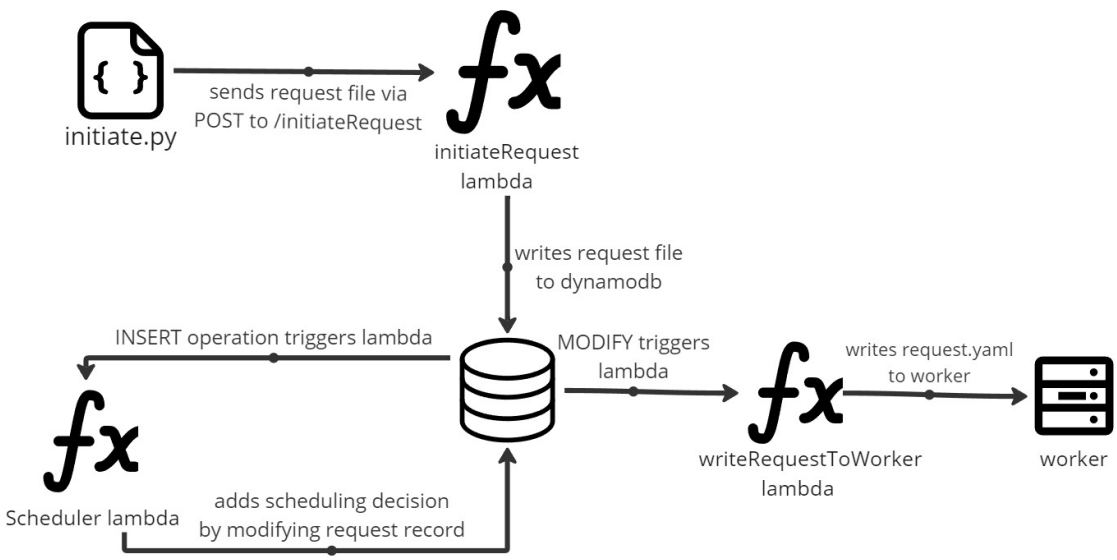

Figure 4.6: The alternative implementation of the system, where database events trigger lambdas.

5

Evaluation

In this chapter, we evaluate the serverless control plane that was designed in Chapter 3 and implemented in Chapter 4. For our evaluation, we focus on four dimensions:

1. **Latency of the system.** We measure how long it takes for a workload request to arrive at a worker. In a serverless architecture, we can expect the latency to increase compared to the serverful counterpart, this experiment will help us gauge how much the latency increases in our system compared to standard Kubernetes.

2. **Throughput of the system.** For our evaluation, we define throughput as the number of requests to our system at the same time, in other words the number of concurrent requests. We measure how the latency of the system changes based on the number of concurrent requests received. This will aid us in measuring the scaling and elasticity of the serverless control plane.

3. **Resource usage of the system.** Here we measure the resource usage (CPU and memory) of our system.

4. **Cost to run the system.** We measure the cost to run our system on AWS. Since serverless functions can scale to zero, we expect it to be cheaper than its serverful counterpart. Measuring this allows us to gauge the impact on cost.

We describe the experimental setup in Section 5.1 and report the results of our experiments in Sections 5.2 through 5.5. We also highlight the limitations of our experiments in Section **??** and close the chapter with a summary and discussion of the evaluation in Section 5.7.

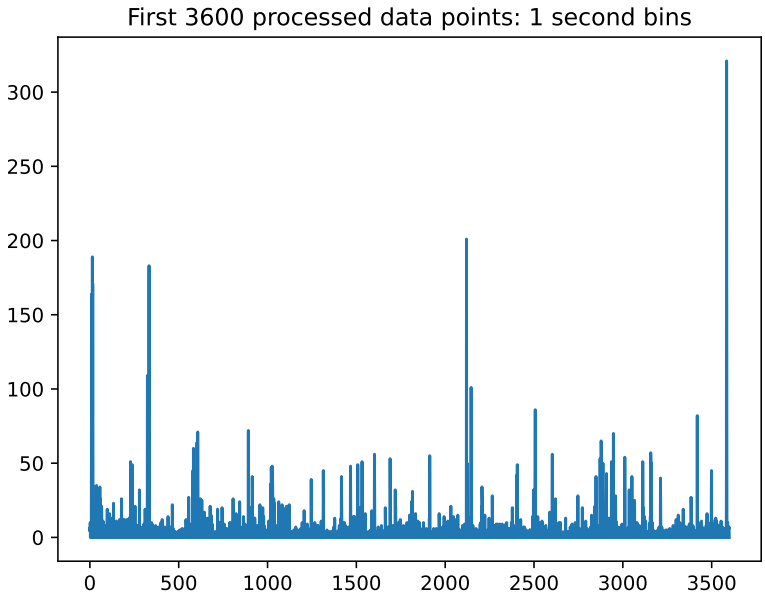

Figure 5.1: Request frequency, per second, for the first hour (3600 seconds), from the Azure VM allocation trace. The Y-axis denotes number of requests, while the X-axis denotes seconds.

5.1 Experimental Setup

In this section, we describe the setup and the process that we used to run and benchmark our system. For our Kubernetes cluster, we use Continuum with the control plane having 10 CPU cores and 50GB of memory. For our serverless functions, we use lambdas with memory of 512 MB, storage of 512 MB and a maximum execution time of 3 seconds. The runtime for the lambdas is Python 3.12. For DynamoDB tables, we set the read and write capacity to on-demand, so that our requests won't be throttled if we hit a certain ceiling. To simulate incoming workloads, we use Azure VM allocation traces[1]. This allows us to simulate part of the workload on Azure clusters, providing a real world workload allocation scenario.

We first convert the trace into 1 second bins and use a bash script to send X parallel curl requests to our serverless API server, where X is the number of requests in that 1 second bin. The distribution of the request frequency, per 1 second bins, for the first hour (3600 seconds) is shown in Figure 5.1. In the figure, the X-axis denotes time, in seconds, and the Y-axis denotes the number of requests per second. For our evaluation we simulate the first

[1]Azure traces: `https://github.com/Azure/AzurePublicDataset/blob/master/AzureTracesForPacking2020.md`

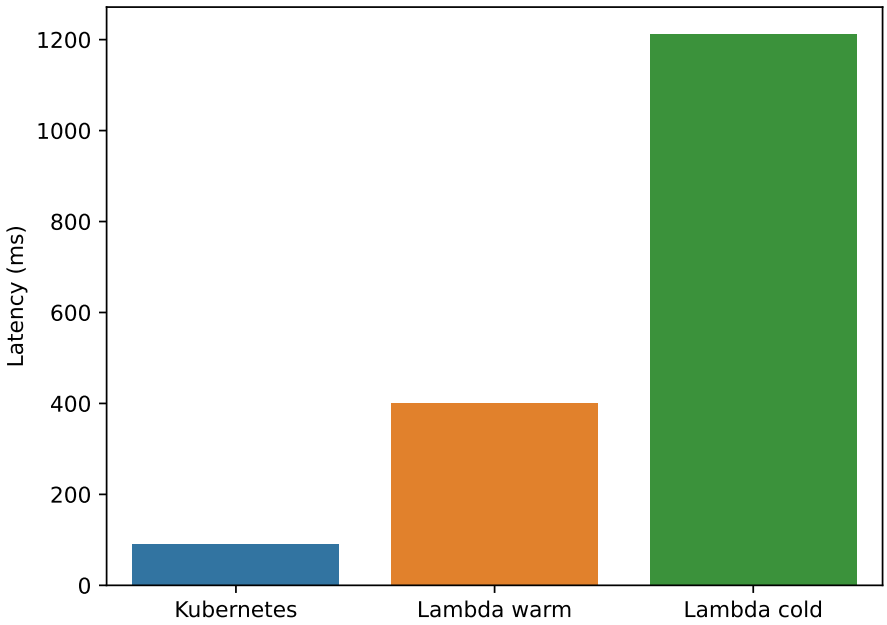

Figure 5.2: Latency of 1 request for Kubernetes and our system. X-axis denotes the control plane and Y-axis denotes the time in ms. Lower is better.

1 hour from the trace, which has a total of 15122 requests. To measure the throughput, we send 500 and 1000 concurrent requests per second to our system for 60 seconds and compare their latencies. We stop at 1000 concurrent requests as it was the highest number of requests in 1 second from all bins in the trace. Each experiment is run 5 times, and the average of the 5 runs in reported, to minimize bias.

To obtain the metrics like resource usage and latency after running these experiments, we make use of AWS CloudWatch. We use CloudWatch because it offers detailed metrics for all the dimensions we are interested in. It is also the metrics tool used by AWS to calculate billing, meaning that the metrics reported from CloudWatch are accurate.

5.2 Latency of requests

The comparison of latencies of our system and the Kubernetes control plane, for a single request, is shown in Figure 5.2, with the X-axis denoting the type of control plane, and the Y-axis denoting latency to serve those requests. Lower latency is better. Comparing latency for a single request, our system takes 1.2s per request for a cold start, and 0.4s for warm starts. For Kubernetes, the latency for a single request was around 80ms.

Next, we run 1 hour of the trace on both systems, the completion time was similar

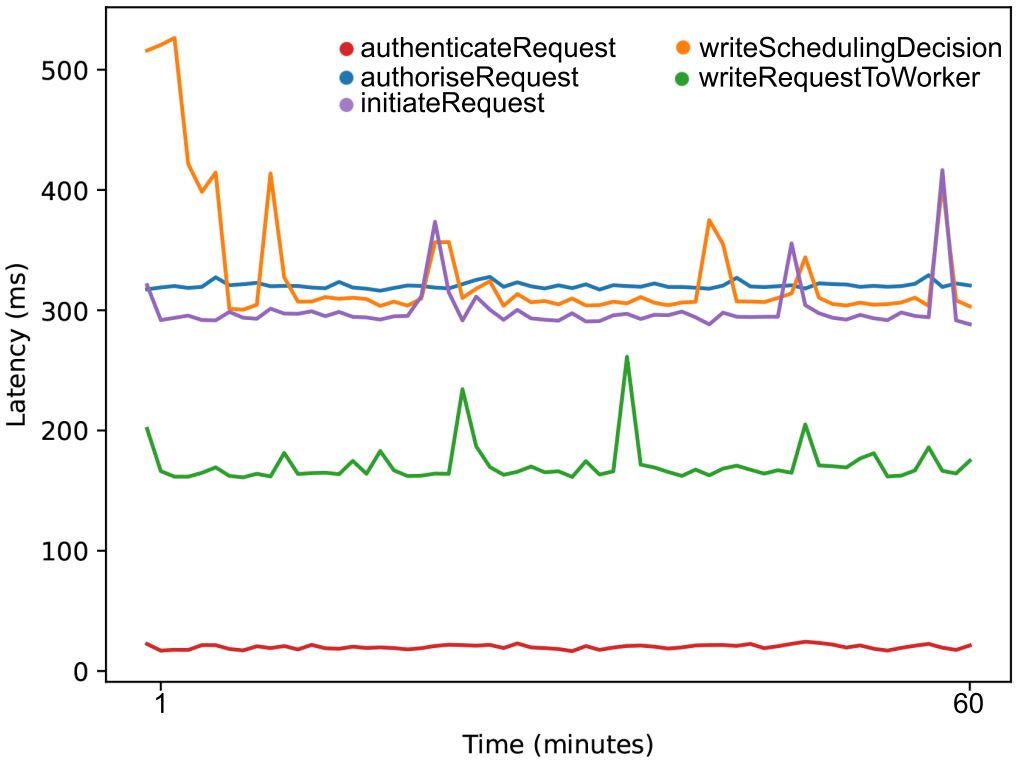

Figure 5.3: Execution latency of the lambdas for the first hour of the trace, shown per minute.

(around 3600 seconds) on both. The breakdown of 90th percentile latency per lambda can be seen in Figure 5.3. In this figure, the X-axis denotes the different lambda functions that make up our control plane, and the Y-axis denotes the 90th percentile latency of those functions, with lower latency being better. On average, each request completed (i.e. got sent to a worker) in 0.5 seconds. This measurement also includes the read and write latencies from DynamoDB. The read and write latency from DynamoDB is around 10ms and 15ms respectively, as such we consider them to be negligible.

Noticeably, the latency of the Authentication function is minimal. This is most likely because this function only has to read one value from a table in DynamoDB, which it can directly query for using the primary key. In comparison, the other functions have to carry out reads as well as writes. In the case of the Authorisation function, the read is actually a scan operation instead of a direct query with a primary key, which takes longer. From this we conclude that although the individual read and write operations in DynamoDB are around 10ms, the time that the lambdas take to process the DynamoDB operations are significant, and constitute the bulk of the latency.

Overall, for a single request, the latency of our serverless control plane is about 4 times more than the standard Kubernetes control plane. The serverless control plane takes 400ms while Kubernetes takes 80ms.

5.3 Throughput

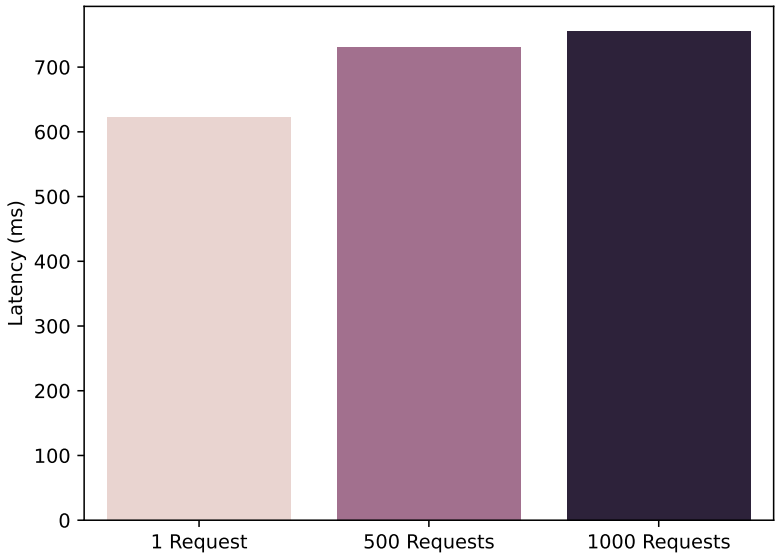

Figure 5.4: 90th percentile latency to complete 1 request, 500 concurrent requests and 1000 concurrent requests per second, for 60 seconds.

To test throughput, we send 500 and 1000 concurrent requests, per second for 60 seconds, to our serverless control plane.

An interesting observation is the latency for completion of concurrent requests, shown in Figure 5.4, with the X-axis denoting the number of concurrent requests sent to our control plane and the Y-axis denoting the 90th percentile latency to serve those requests. It takes about the same amount of time for our control plane to complete 1000 concurrent requests, per second for 60 seconds, and 1 request per second for 60 seconds. These measurements were carried out in intervals of 30 minutes to ensure there are no warm starts. This proves that our system scales well to handle large number of requests in a short time. The comparison of running this test on our system and Kubernetes is shown in Figure 5.5. The total run time of the test is 3 minutes, and from Figure 5.5 we can see that our serverless control plane completes the test in around 3 minutes, while Kubernetes takes 19 minutes to complete. This is because at 500 requests per second and above, Kubernetes is not able to keep up with the incoming requests and starts to queue requests, also evidenced

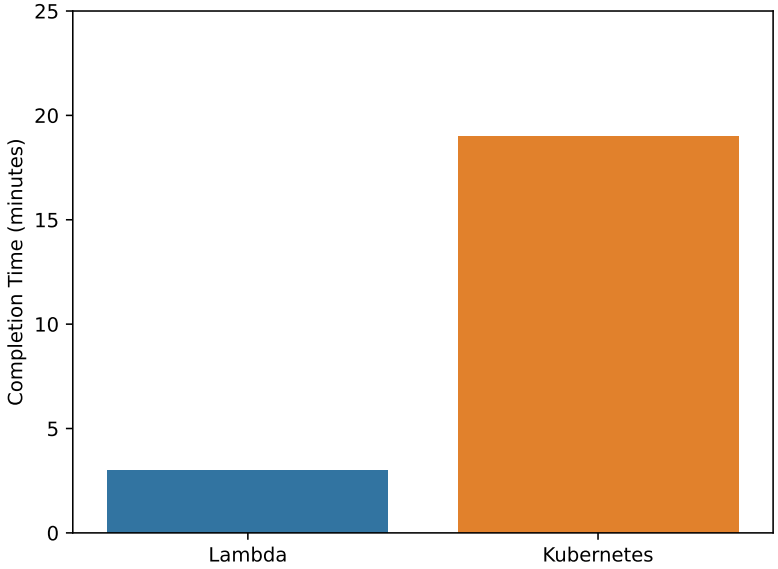

Figure 5.5: Total time taken to complete 1, 500 and 1000 concurrent requests per second, each for 60 seconds, for a total test time of 3 minutes.

by the CPU usage of Kubernetes reaching 100% while handling 500 requests per second as detailed in Section 5.4.

Next, we look at the concurrent executions of our lambdas during this test. Figure 5.6 shows the concurrent lambda executions for the respective number of requests, with the X-axis denoting the lambdas and the Y-axis denoting the maximum number of concurrent executions of those lambdas. Higher concurrent executions are better, with the ideal scenario being the number of concurrent executions of each lambda matching the number of requests.

We see that even though we sent 500 and 1000 requests in parallel, the concurrent execution of the lambdas is below that. We suspect this may be because of some internal load balancing done by AWS as the concurrent executions get closer to the account limit of 1000 concurrent executions. To test this, we launched 100 requests in parallel, which means a total of 500 concurrent executions (100 executions for 5 lambdas). This is well below the account limit of 1000 concurrent executions, and we see that all lambdas are able to reach 100 concurrent executions (Figure 5.7a). The Authentication lambda, which has an execution duration of about 14ms, is always below the ideal concurrency. This most likely is due to its short execution time, due to which it is not making use of the maximum concurrency. Next, we send 200 requests in parallel, amounting to a total of 1000 concurrent lambda executions. This is equal to the account limit, and we see that 200

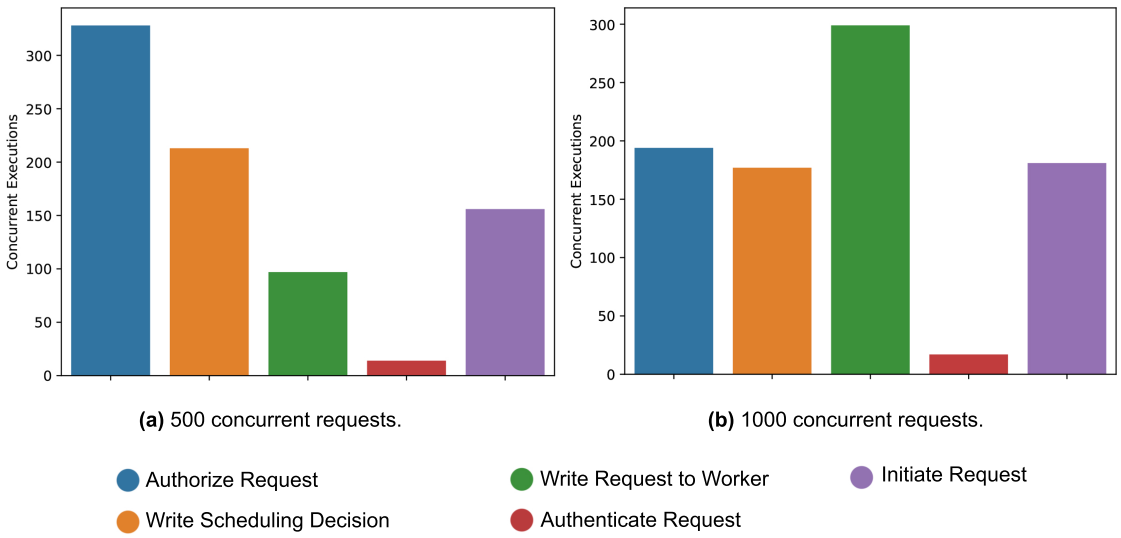

Figure 5.6: Maximum number of concurrent lambda executions for 500 and 1000 concurrent requests.

concurrent requests per lambda isn't reached, the maximum is 195 concurrent executions (Figure 5.7b). This leads us to believe that higher concurrency can be achieved if we have a higher account concurrency limit from AWS.

5.4 Resource Usage

We measured the resource usage during the throughput test, where we sent 1 request per second, 500 requests per second, and 1000 requests per second, each for 60 seconds. The CPU and memory usage for Kubernetes is shown in Figure 5.8, with Figure 5.8a showing the CPU usage and Figure 5.8b showing the memory usage. As mentioned in Section 5.1, the control plane has a 10 core CPU with 50GB of memory. CPU and memory usage per lambda for every request is shown in Figure 5.9a and 5.9b, respectively.

Kubernetes often touched 100% CPU usage during the 500 requests per second and above. The memory usage for Kubernetes went from 2GB at the start of the test, to 4GB by the end, increasing over time. In contrast, the CPU time for every lambda to process requests is about 110ms, except around 15ms for the Authentication lambda. The total memory for every lambda is 512 MB, which means for every request, the lambdas use about 16%, or 96 MB of memory.

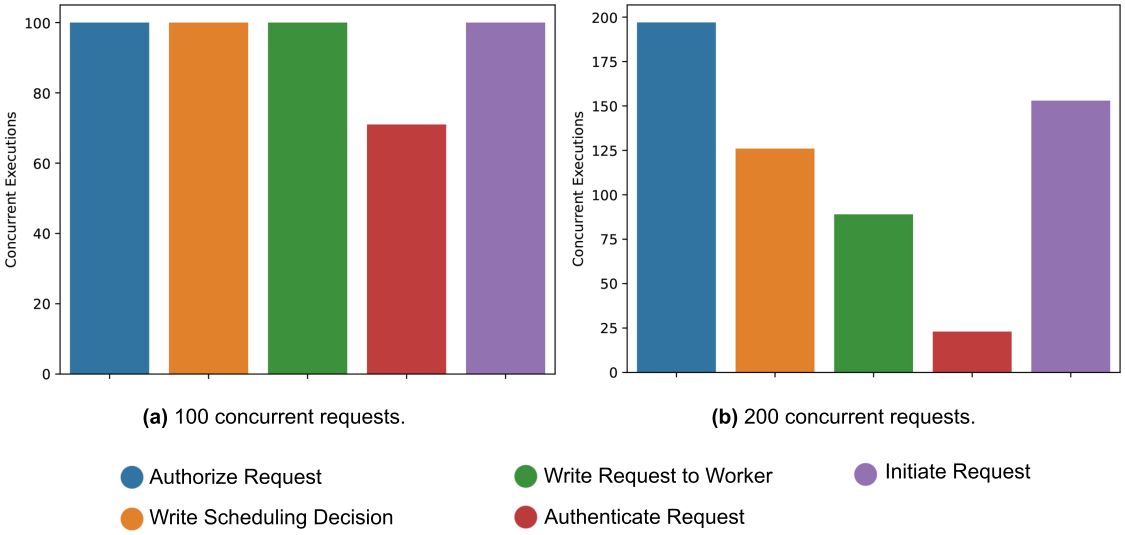

Figure 5.7: Maximum number of concurrent lambda executions for 100 and 200 concurrent requests.

5.5 Cost

In our experiments, we were within the AWS free tier limit for all resources. So in order to estimate cost for our system, we use the AWS Pricing Calculator. AWS Pricing Calculator is a service offered by AWS to measure the cost of running AWS services based on specific usage criteria. For every request, our system has 5 lambda invocations, 5 DynamoDB reads and 2 DynamoDB writes. Using this information in the AWS pricing calculator, we find that for 2 million requests a month, the cost of running the serverless control plane comes to 65.24 USD. In comparison, running the control plane in AWS EKS, costs 0.10 USD per hour per cluster, making it 74 USD a month to run the control plane for a single cluster, even if there is no workload.

5.6 Limitations and Threat to Validity

For our tests, we used lambdas with 512MB of memory and no provisioned concurrency, due to cost. Provisioned concurrency is a feature for lambdas where AWS will keep a specified number of instances of a function pre-warmed to reduce latency. Increasing the memory of lambdas and configuring provisioned concurrency will allow for lower latencies than observed in our tests. Other possible optimisations include using DynamoDB accelerator (DAX), which caches frequently accessed items in memory. This could improve latencies

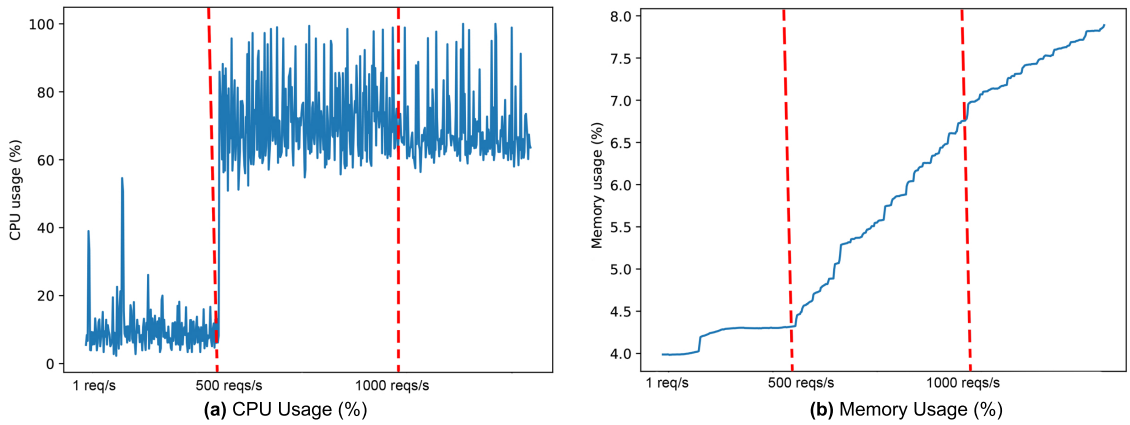

Figure 5.8: CPU and memory usage, in percent, for Kubernetes control plane. The Y-axis denotes the respective percentages, and the X-axis denotes the time, with the start time of 1 request/s, 500 requests/s and 1000 requests/s shown.

of Authorisation and Scheduler lambdas as they read from DynamoDB frequently. We chose to not implement this because this is specific to AWS and we focused on keeping our design and implementation cloud provider agnostic.

For our throughput tests in Section 5.3, we were unable to get the per-request latency for Kubernetes, which would have been useful to further visualise how the latency increased with increase in requests.

For testing resource usage, in Section 5.4, we measured CPU usage percent from Kubernetes but CPU time for lambdas. This is because AWS only shows CPU usage in terms of CPU time and otherwise has no other information regarding CPUs of lambda functions. Due to this, we could not make a direct comparison between them, but we can still see that CPU usage is constant in lambdas and increases with number of requests in Kubernetes.

5.7 Summary of Evaluation

In this section, we give a brief summary of the findings from our experiments in the previous section. **The latency of our serverless control plane is, on average, 4x higher than the standard Kubernetes control plane for a single request.** The average time to complete one request by our system was 1.2 seconds when cold started, and 0.4 seconds when warm started. A point of note here is that we used 512 MB of memory for our lambdas, and since AWS scales the CPU allocated to the lambdas in proportion with the allocated memory, opting for 1024 MB or higher memory for the lambdas may

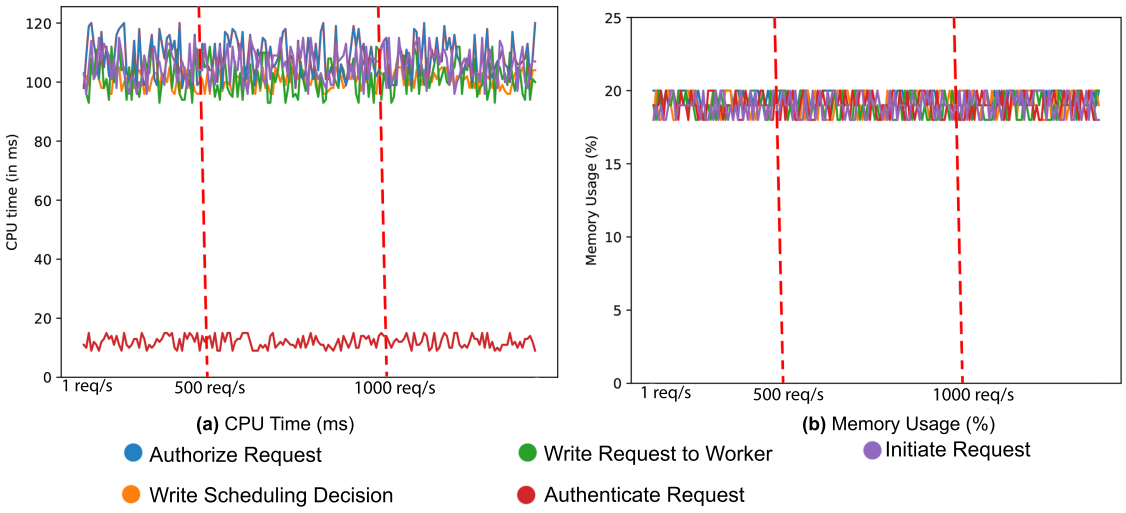

Figure 5.9: CPU time and memory usage per lambda for every request. The Y-axis denotes the respective percentages, and the X-axis denotes the time, with the start time of 1 request/s, 500 requests/s and 1000 requests/s shown.

see a performance increase.

For throughput, we sent 1, 500 and 1000 parallel requests to our control plane per second, for 60 seconds each. We notice that for our serverless control plane, the **time to serve 500 concurrent requests and 1000 concurrent requests is equal to the time it takes to serve one request. The serverless control plane completes the test in 3 minutes, while Kubernetes takes 19 minutes.** This highlights the scalability of our system, showing that it is able to handle sudden increases in concurrent requests without increasing the latency, unlike the standard Kubernetes control plane which starts to buffer requests when under load.

The lambdas in **our serverless control plane use, on average, 105ms of CPU time and 96MB of memory per request.** Lastly, the cost to run the serverless control plane is based on actual usage, unlike the managed Kubernetes services offered by cloud providers like AWS, where the control plane has a fixed price regardless of workload. **For 2 million requests a month, the serverless control plane costs 65.25 USD to run on AWS** compared to the 74 USD per month for running the control plane in EKS, even under zero workload.

6

Related Work

In this section, we briefly discuss some of the previous papers and surveys that deal with scaling Kubernetes or combining serverless computing with Kubernetes. To the best of our knowledge, our work is the first that tries to dynamically scale the control plane with load.

6.1 Scaling in Kubernetes

Cloud Computing With Kubernetes Cluster Elastic Scaling (8). They develop an open source solution for autoscaling Kubernetes worker nodes within a cluster to support dynamic workloads, particularly for ubiquitous computing and AI applications. This differs from our work in two ways. Firstly, they scale the data plane instead of the control plane, and secondly, they scale up using VMs due to which latencies of up to 90 seconds are observed.

Decentralized Kubernetes Federation Control Plane (9) proposes a decentralized control plane for Kubernetes federations, designed to address the scalability and reliability challenges of managing thousands of clusters in edge computing scenarios. While the vision of this paper is somewhat similar to ours, in the sense that it proposes scaling the control plane, it is limited to federations of clusters and not applicable to smaller scale clusters. The authors present this as a conceptual framework and starting point for future work, rather than a fully realized and tested system.

A Survey of Autoscaling in Kubernetes (10) presents a comprehensive survey of autoscaling techniques in Kubernetes. In contrast to our work, they only cover scaling in the data plane.

6.2 Kubernetes and Serverless Computing

Benchmarking Serverless Workloads on Kubernetes (11) runs serverless workloads on Kubernetes and benchmarks the latency and throughput. Differing from our work, they use serverless in the data plane of Kubernetes, while using the traditional Kubernetes control plane to manage the serverless workloads.

An evaluation of open source serverless computing frameworks (12) evaluates various serverless frameworks, including some based on Kubernetes by extending the Kubernetes API to support serverless functions as a custom resource type.

A scheduler for serverless framework base on kubernetes (13) proposes a new scheduling algorithm that enables rapid scheduling of pods for serverless workloads. In similarity to our work, they also modify the control plane, but with the goal of supporting serverless workloads in the data plane.

7

Conclusion

In this thesis we designed, implemented and evaluated a serverless control plane for Kubernetes. The main goal of this was to improve the elasticity of the control plane wherein it can respond to sudden large spikes in workload, and also scale to down in times of low demand to save cost and resources. We implemented and evaluated this system in AWS and found that our serverless control plane takes has 6x more latency compared to the standard Kubernetes control plane for a single request (400ms vs 60ms), but has good performance in high throughput situations where it takes the same time to serve 1000 requests a second as it does for 1 request a second.

7.1 Answering Research Questions

In this section we present our answers to the research questions proposed in Section 1.3.

RQ1.1 : How can the API server be implemented in a serverless system?
Typically, each serverless function should be responsible for only one particular functionality of the system. Therefore to design a serverless API server we split the basic functionalities of the Kubernetes API server into separate serverless functions, leading to four separate functions that together make up the API server. These functions handle authentication of a request, authorisation of the request, processing the request and sending the request to the appropriate worker. Due to the serverless nature of our system, our design choices deviated from Kubernetes in some areas. We chose to store the token file for authentication in the datastore because it cannot be stored on the serverless functions, as storage there is ephemeral. Second, we chose to not authenticate the control plane components to each other, unlike Kubernetes. This is because authentication of the components, in our case serverless functions and cloud services like datastore, is already handled by the

cloud provider and only authorised parties (for example with the proper IAM roles) can modify these components. We found that, at least in AWS, triggering the functions from datastore events introduced significant latency because the events are batched together and executed, instead of being immediately executed as the event occurs. For this reason, in our design we trigger the functions directly from the previous functions in the chain, which reduces latency.

RQ1.2 : How can the datastore be implemented in a serverless system?
The data store needs to a persistent key-value store, following the same from Kubernetes design. Not all serverless functions should have write access to the datastore, only the functions responsible for scheduling the requests and processing the request should be able to write to the datastore, while the others should have only read access.

RQ1.3: How can the scheduler be implemented in a serverless system?
After completion of authentication and authorisation of a new request, the function responsible for processing the request triggers the scheduler function, which in our implementation, pulls metrics for the nodes in the cluster from a table in the datastore. In a realistic scenario, the scheduler function will query the metrics endpoint to get the node metrics, and then schedule the request to a worker.

RQ2: How do we deploy and configure a serverless control plane in the cloud?
For deploying the serverless control plane, we chose AWS, but our design is agnostic of cloud provider. We use DynamoDB as the datastore, due to it being a key-value store as well as focused on low latency execution of requests.

RQ3: What is the latency impact when the control plane is serverless?
In our evaluation, we find that for a single request, our serverless control plane has 6x more latency than Kubernetes. The average time to serve one request is 1.8 seconds when cold started, and 0.4 seconds when warm started. The serverless control plane is good in high throughput situations, where it was able to handle 1000 requests per second, for 60 seconds with the same latency as it handled 1 request per second.

RQ4: What is the impact on cost when the control plane is serverless?
Using the AWS Pricing Calculator, we find that it costs 65.25 USD to serve 2 million requests a month, and the cost decreases if there is less workload. In comparison, the AWS managed Kubernetes service, EKS, costs 74 USD per month per cluster to run. This is even when no workload is carried out.

7.2 Limitations and Future Work

Our focus in this work was to design a proof of concept serverless control plane that can scale quickly with increase in workload. As such, we have only implemented the bare functionality required to send a request to a worker, in order to measure latency and scalability. We have left out some components of the Kubernetes control plane from our system. Notably, we did not implement controllers, the controller manager, and validation and translation of API requests.

References

[1] CNCF. **Cloud Native 2023: The Undisputed Infrastructure of Global Technology**, 2023. 1

[2] Ruslan Meshenberg Yury Izrailevsky, Stevan Vlaovic. **Completing the Netflix Cloud Migration**, 2016. 1

[3] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. **Protean:{VM} allocation service at scale**. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020. 3

[4] Christopher Berner. **Scaling Kubernetes to 2,500 nodes**, 2018. 3

[5] Benjamin Chess Eric Sigler. **Scaling Kubernetes to 7,500 nodes**, 2021. 3

[6] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: automate infrastructure deployment and benchmarking in the compute continuum**. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 181–188, 2023. 6

[7] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. **Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider**. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020. 12

[8] Brandon Thurgood and Ruth G Lennon. **Cloud computing with kubernetes cluster elastic scaling**. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, pages 1–7, 2019. 41

[9] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. **Decentralized kubernetes federation control plane**. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 354–359. IEEE, 2020. 41

[10] Minh-Ngoc Tran, Dinh-Dai Vu, and Younghan Kim. **A survey of autoscaling in kubernetes**. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 263–265. IEEE, 2022. 41

[11] Hima Govind and Horacio González-Vélez. **Benchmarking serverless workloads on kubernetes**. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 704–712. IEEE, 2021. 42

[12] Sunil Kumar Mohanty, Gopika Premsankar, and Mario Di Francesco. **An evaluation of open source serverless computing frameworks**. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 115–120. IEEE, 2018. 42

[13] Dayong Fan and Dongzhi He. **A scheduler for serverless framework base on kubernetes**. In *Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference & 2020 3rd International Conference on Big Data and Artificial Intelligence*, pages 229–232, 2020. 42

Appendix A

Reproducibility

A.1 Abstract

The serverless control plane was tested on AWS, the comparison with Kubernetes was done using Continuum Framework on an Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz with 10 cores and 50GB of memory. The code is publicly available.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** AWS Lambda with Python 3.12
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes
- **How much time is needed to complete experiments (approximately)?:** 1.5 hours
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to access

`https://github.com/minoritydev/msc-thesis/tree/main`

A.3.2 Data sets

Azure VM allocation trace for packing : `https://github.com/Azure/AzurePublicDataset/blob/master/AzureTracesForPacking2020.md`

A.4 Installation

The installation involves setting up the infrastructure on AWS, with the following steps: Set up DynamoDB:

1. Make 'Requests' table, with 'id' (string) as partition key.
2. Make 'Roles' table, with 'name' (string) as partition key and 'namepspace'(string) as sort key.
3. Make 'Rolebindings' table, with 'name' (string) as partition key and 'namepspace'(string) as sort key.
4. Make 'node-metrics' table with 'node-id' (string) as partition key.

Set up EFS:

1. Create a new EFS filesystem in the default VPC.
2. Create an access point for the EFS by going to EFS > Access Points > Create access point.

Set up Lambdas:

1. Create 5 lambdas with 512MB memory (or more) and runtime of Python 3.12 with Arcchitecture x86_64. Name them '*authenticateRequest*', '*authorizeRequest*', '*initiateRequest*', '*writeSchedulingDecision*' and '*writeRequestToWorker*'. The code for these lambdas is available in the GitHub repo.
2. Attach EFS to *writeRequestToWorker* lambda by going to Lambda > Configuration > File Systems and attaching the EFS created earlier.
3. To make sure the *writeRequestToWorker* lambda can access the EFS, you need to add it to the same VPC as the EFS. Do this by going to Lambda > VPC.
4. You may need to add this lambda to a security group that allows all inbound traffic.

Set up API Gateway trigger:

1. Create a new API endpoint to trigger the *authorizeRequest* lambda. The easiest way to do this is to go to the lambda in AWS console > Add Trigger > Set source to API Gateway > Create New API > HTTP API > Set security to Open > Add.

2. Now set up a lambda authorizer for the create API endpoint. To do this, go to the API route in AWS console > Click Configure under Authorizarion > Create and attach an authorizer > Set authorizer type to Lambda > Give it a name > select the *authenticateRequest* lambda under Lambda Function > Set response mode to IAM Policy > under identity sources enter $request.header.Authorization > Enable 'Automatically grant API Gateway permission to invoke your Lambda function' > Create and attach.

A.5 Evaluation and expected results

Numbers similar to evaluation section

A.6 Experiment customization

The tests are available on GitHub and you are free to customize them.

Appendix B

Kubernetes YAML definition examples

B.1 Role definition YAML

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: pod-manager
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "create", "delete"]
```

B.2 Rolebinding definition YAML

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-pod-manager
  namespace: development
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
```

```
  kind: Role
  name: pod-manager
  apiGroup: rbac.authorization.k8s.io
```

B.3 Request definition YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```