Vrije Universiteit Amsterdam

Bachelor Thesis

Kubeλess: A Novel Architecture for Kubernetes' Control Plane

Author: Davit Darbinyan (2739431)

1st supervisor:	Prof. dr. ir. Alexandru Iosup
daily supervisor:	Matthijs Jansen, MSc
2nd reader:	Dr. Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 21, 2024

Abstract

In the rapidly evolving landscape of cloud computing, container orchestration tools have become critical for managing large-scale applications. Among these tools, Kubernetes has established itself as the *de facto* standard due to its extensive features and community support. Central to Kubernetes' architecture is its control plane, responsible for managing the overall state of the cluster. In cases of large clusters, the control plane needs to scale to manage the increasing pressure exerted by the workers. However, the relatively large size of its containers can lead to slow startup times, making it challenging for the control plane to promptly respond to variations in workloads. Serverless computing, characterized by its ability to dynamically scale resources based on demand, presents a promising solution to this problem.

This thesis aims to address the problem of insufficient elasticity in Kubernetes' control plane, through the design, implementation and evaluation of *Kubeλess*, a novel serverless architecture for Kubernetes' control plane. Our results demonstrate that *Kubeλess* significantly reduces the startup time of control plane components and achieves more efficient pod deployment in high-demand scenarios. The code for this thesis work is openly available GitHub.

Contents

1

Introduction

With the growing popularity of containerization techniques, container orchestration tools gained significant attention and became essential for automating the deployment, scaling, and management of container-based applications in the cloud [1]. Among these tools, Kubernetes stands out as the most popular and widely adopted, establishing itself as the *de facto* standard in the industry [2, 3, 4]. While other orchestration tools like Docker Swarm [5] and Apache Mesos [6] are also popular, Kubernetes' extensive features and community support made it a favorite among end-users, with 66% using it in production [7]. Some authors have even referred to the last several years as a "Kubernetes Tsunami" in the IT world, highlighting the overwhelming impact and rapid adoption of the system [8].

This widespread adoption is evident as many large companies, including IBM, Pinterest, and OpenAI, have successfully migrated their infrastructure to Kubernetes for managing their containerized applications [9]. These companies have reported improved deployment speeds, enhanced scalability, and more efficient resource utilization, demonstrating the transformative impact of Kubernetes on their operations. The economic impact of Kubernetes is also substantial, with 44% of companies citing the need to increase revenue or profits as one of the primary reasons for using Kubernetes [10].

1.1 Context

Kubernetes, originally developed by Google and inspired by a decade of experience in deploying reliable container applications through systems such as Borg and Omega [11], has evolved into a comprehensive platform for managing containerized workloads. Currently, it is widely used for deploying large and scalable applications due to its robust auto-scaling

mechanisms that efficiently manage varying loads. Depending on workload fluctuations, these mechanisms determine how to scale the number of compute nodes in an existing cluster (i.e., horizontal scaling) and/or the quantity of assigned computing resources (CPU time, cores memory, etc.) for each node (i.e., vertical scaling) [12].

Central to Kubernetes' architecture is its control plane, consisting of a datastore and containerized API servers, schedulers, controller managers, and more [13]. Together, these components are responsible for managing the overall state of the cluster. The control plane follows a "hub-and-spoke" design pattern where each component reads and writes to the datastore via the API server and does not directly communicate with any other control plane component [11]. In cases of large clusters, the control plane components need to scale to manage the increasing pressure exerted by the worker nodes. However, the relatively large size of its containers can lead to slow startup times, making it challenging for the control plane to promptly respond to variations in the workload. Thus, Kubernetes fails to provide a sufficient degree of elasticity, which is a fundamental attribute of cloud computing.

Elasticity enables applications to quickly adapt to fluctuating workloads by dynamically scaling computational resources (such as CPU cores, memory, VM, and container instances) [12]. It ensures that the application can meet user demands efficiently while optimizing resource utilization and cost-effectiveness [14]. In production environments, the need for elasticity is crucial due to the nature of workloads that can vary drastically, often experiencing sudden and unpredictable spikes in demand. A study on the real-world workloads of Azure Functions reveals that functions can exhibit a range of 8 orders of magnitude in invocation frequencies [15]. Although the vast majority of these functions are invoked infrequently, the most popular ones can be triggered as much as 100 million times a day. Such variability highlights the necessity for an elastically scaling solution capable of adapting to these dynamic demands.

One solution to this need is serverless computing, popular for its ability to scale operations elastically. It has emerged as a relatively recent paradigm in cloud computing, transforming the way programs are built and scaled [16]. At its core, serverless computing abstracts the underlying infrastructure, allowing users to run event-driven and granularly billed applications without having to address the operational logic [17]. By taking responsibility for the setup, management, and maintenance of the physical infrastructure, serverless technologies enable rapid development and deployment, as developers can focus primarily on the business logic of their applications.

1.2 Problem Statement

As illustrated in the aforementioned study of Azure Functions, workloads in a production environment can vary frequently and rather drastically. While Kubernetes' control plane has auto-scaling mechanisms to handle these fluctuating workloads, those are not fast enough to provide a sufficient degree of elasticity in large clusters. This poses a serious problem, especially when 50% of Azure workloads execute in less than one second on average [15]. As a result, large delays in the control plane are extremely obstructive, and elastic mechanisms are necessary to ensure its responsiveness to workload variations. A serverless architecture designed for rapid scaling operations presents a promising solution. Kubernetes' control plane follows an event-driven architecture which aligns well with this serverless design, allowing it to scale quickly in response to changing demands.

In addition to ensuring elasticity, a serverless control plane can offer significant advantages in terms of resource utilization and cost efficiency. Traditional Kubernetes environments often pre-allocate more control plane instances than required, to handle peak workloads. However, leasing costly cloud-based resources can be unaffordable in the long run, as those would be underutilized during periods of low demand. In contrast, a serverless control plane can dynamically allocate and deallocate resources based on workload demands, thus optimizing resource usage and reducing operational costs. This aligns with the economic impact that Kubernetes has had on companies, as it can further contribute to cost savings and increase profits.

Lastly, optimized resource utilization, provided by a serverless control plane, also addresses energy efficiency and climate responsibility, tackling one of the four grand societal challenges highlighted in the CompSys Manifesto for the Netherlands [18]. This manifesto also identifies Resource Management and Scheduling as a foundational research theme required to tackle those challenges and urges researchers to study new approaches to its operational efficiency. This study's aims align with these calls, as it attempts to enhance elasticity in one of the most popular resource management platforms. By improving Kubernetes' responsiveness to dynamic workloads, this research can lead to more sustainable and efficient cloud computing, benefiting both industry and society through smarter resource utilization.

1.3 Research Questions

In this work, we attempt to address the challenge of elasticity present in Kubernetes' control plane using a serverless architecture. To tackle the aforementioned challenge, we identify the following research questions.

RQ1 How can the Kubernetes control plane be redesigned to operate entirely within a serverless architecture?

To identify a suitable approach for adapting Kubernetes' control plane to a serverless architecture, it is essential to develop a design that takes into account the specific requirements and constraints of both Kubernetes and serverless computing.

RQ2 What are the practical steps required to implement a serverless Kubernetes control plane?

Once the appropriate design has been identified, we must understand the integration process of the new control plane within the Kubernetes framework to ensure that the overall system's functionality and reliability are maintained.

RQ3 What are the performance characteristics of a serverless Kubernetes control plane, and how do they compare to the traditional Kubernetes?

Having designed and implemented the new architecture, it is essential to evaluate its performance and analyze its behavior under regular operations. This will allow us to quantify the potential improvements brought about by this solution.

1.4 Research Methodology

A combination of various research methodologies is used to address the aforementioned research questions. To answer **RQ1**, we carry out a systematic study of Kubernetes' architecture, focusing primarily on its control plane. We also conduct a shortened literature survey investigating existing systems with serverless control planes (**M1**; quantitative research [19, 20]). This helps us get an in-depth understanding of Kubernetes and gather insights and best practices in serverless design, forming the background of this research.

To build on our answer for **RQ1** and tackle **RQ2**, we design and implement a serverless control plane, following a process inspired by the AtLarge vision on the design of distributed systems. This process involves an iterative cycle with various stages, including requirements formulation, design, implementation, and analysis (**M2**; design, abstraction,

prototyping [21, 22, 23]). The implementation of the serverless control plane focuses primarily on control loops, which are the backbone of most of control plane components, and are responsible for continuously monitoring and reconciling the state of resources.

Finally, to address **RQ3**, we design a series of experiments to evaluate the performance characteristics of the serverless control plane (**M3**; experimental research, designing appropriate micro and workload-level benchmarks, quantifying a running system prototype [24, 25, 26]). These experiments are conducted using the *Continuum* framework [27], which was designed to automate infrastructure deployment and benchmarking for the cloud.

Throughout all phases of our research, we prioritize and follow the principles of open and reproducible science. Thereby, our implementation and experiment results are open-sourced and adhere to standard practices (**M4**; open-science, open-source software, community building, peer-reviewed scientific publications, reproducible experiments [28, 29, 30, 31]).

1.5 Thesis Contributions

In answering our research questions, we provide the following contributions, mapped to the research question that they answer:

1. *(Design, **RQ1**)* Design of a *Kubeλess*, a novel architecture for Kubernetes with a serverless control plane.
2. *(Artifact, **RQ2**)* Implementation of a prototype for *Kubeλess*, focusing on implementing a serverless job controller.
3. *(Experimental, **RQ3**)* Quantitative results and analysis of the serverless control plane's performance compared to the traditional architecture.

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.

1.7 Thesis Structure

We briefly introduce the structure of the rest of the paper. In Chapter 2, we cover the relevant background information. In Chapter 3, we address **RQ1** by presenting the design

of *Kubeλess*. Chapter 4 discusses the steps required to implement a prototype for *Kubeλess*, addressing **RQ2**. This prototype is then evaluated in Chapter 5, answering **RQ3**.

2

Background

In this section, we present the core concepts required to understand our work. We begin by providing an architectural overview of a Kubernetes cluster, looking at its main components and their functionalities. We then discuss control loops, one of the fundamental mechanisms used in Kubernetes, in more detail and explore its internal components.

2.1 Overview of Kubernetes' Architecture

A Kubernetes cluster consists of various components, that together form the control plane, responsible for managing the lifecycle of containers, and the data plane, responsible for running the containerized applications. The main architectural components of both planes are depcited in Figure 2.1.

The control plane is composed of several vital components, that can possibly be replicated to manage increasing workloads. At the heart of the control plane is the API server, which serves as the sole interface for reading and writing workload objects and their states to a persistent datastore. The remaining control plane components regulate the state of those workloads, reconciling their current state to the desired state through control loops. A control loop (i) watches the datastore through the API server, (ii) reads newly created or updated objects, (iii) manages the state of the read objects, and (iv) writes them back to the datastore [32]. For example, the controller manager reads submitted workload objects, creates pod objects (the smallest deployable unit of Kubernetes housing a group of containers) described in the workload, and writes them to the datastore. Afterwards, the scheduler watches for newly created pods with no assigned node, selects a node for them to run on, and writes the decision back.

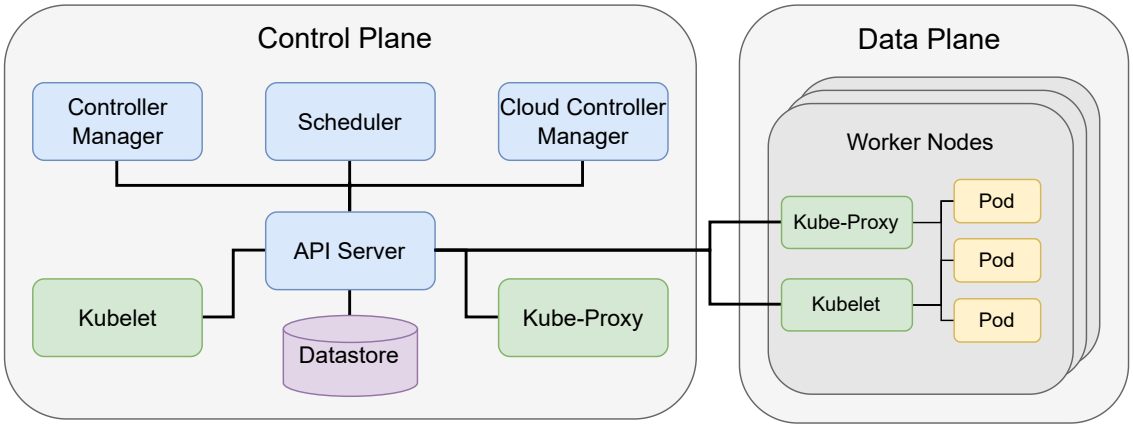


Figure 2.1: Main architectural components of a Kubernetes cluster

The data plane is composed of one or more worker nodes (physical and/or virtual) that execute the actual workloads. On each node runs the *kubelet* and the *kube-proxy*. The former is an agent which makes sure that the containers described in the specifications of pods scheduled on its node are running and healthy. The latter handles the networking layer, allowing communication to pods from inside or outside the cluster.

2.2 Internal Components of a Control Loop

As highlighted in the previous section, the control loop is a key mechanism used by many of the control plane components. It is central to how Kubernetes ensures the cluster's current state consistently aligns with the desired state defined by users. Therefore, it is crucial to understand the intricate workings of the control loops and its internal components, as depicted in Figure 2.2.

One of the key components of a control loop is the Informer, which provides a higher-level abstraction over the resource tracking mechanisms. It starts its operation by requesting the API server to list all instances of a specific resource (such as job, pod or deployment). Following this, the Informer initiates a watch against the API server, using the resource version returned as a response to the previous request. This mechanism allows the controllers to fetch the current state and then subscribe to changes that occur after the specified resource version, without missing any events. When such a change is detected, the Informer updates its internal cache holding the last known state of the watched resources. This local caching is vital as it reduces the load on the API server and ensures that controllers can access the current state of resources efficiently.

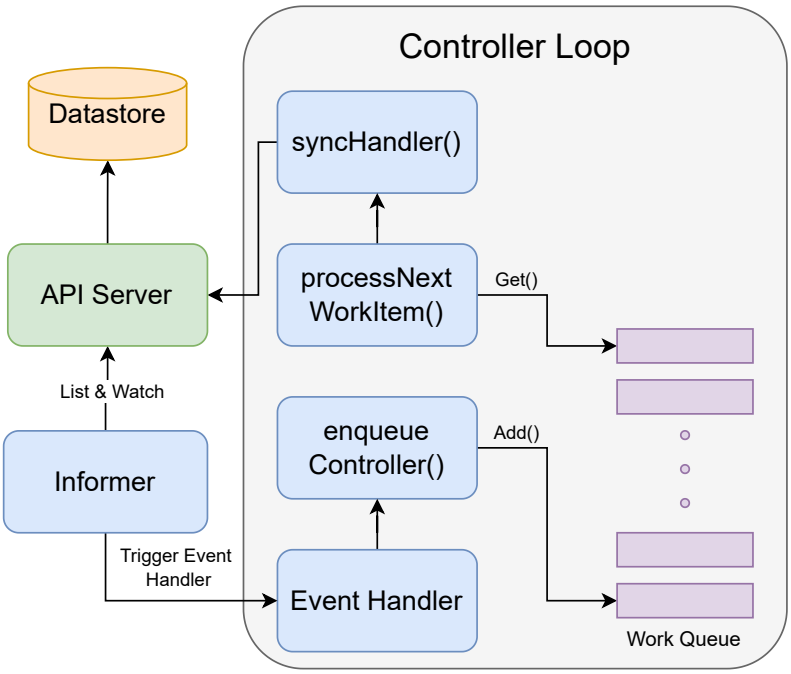


Figure 2.2: Internal components of a control loop

After updating its cache, the Informer triggers relevant event handlers, which are callback functions registered by the controllers during initialization. These functions are designed to respond to specific events, such as the addition or deletion of a resource. Once they receive the modified resource object, they process it and enqueue the object key into a rate limiting work queue. This work queue is operated on by multiple workers started by the controller, which run periodically in separate goroutines, fetch the next item to process and invoke the *syncHandler*, a function that synchronizes the state of the resource. It starts off by reading the current state of the resource, performs the necessary operations to align it with the desired state, and writes any changes back to the datastore. For instance, when a new job is created, the *syncHandler* of the job controller chechs the job's specifications, such as the number of desired completions and paralellism, creates the necessary amount of pod objects from the mentioned template, and writes them to the datastore.

This reconciliation process is iterative and ongoing. For instance, if the desired state specifies that a certain number of pods should be running, the controller ensures that this number is maintained by creating or terminating pods as needed. By continuously comparing the current state with the desired state and making adjustments, controllers ensure that the system remains consistent and operates as intended.

3

Design of *Kubeλess*

In the traditional Kubernetes architecture, control plane components, such as the API server, controller manager, and scheduler are constantly running to manage the state of the cluster. This always-on nature poses significant challenges in terms of elasticity and resource utilization. When there is a sudden spike in workloads, scaling up the control plane requires the allocation of new nodes, a process that is inherently slow and inefficient. Alternatively, pre-allocating multiple control plane instances can ensure availability, but results in wasted resources and higher costs during periods of low workload.

To address this lack of rapid elasticity and poor resource utilization in the Kubernetes control plane, we formulated **RQ1**, asking *how can the Kubernetes control plane be re-designed to operate entirely within a serverless architecture?*. In this chapter, we answer **RQ1** by following the AtLarge design framework [21] to propose the design of *Kubeλess*, a novel Kubernetes architecture with a serverless and elastic control plane.

3.1 Requirements Analysis

We started the design process by eliciting the requirements for the new Kubernetes architecture, based on its current architecture and the problems found in it. The list of the core requirements is presented below.

R1 Serverless Control Loops: Monolithic containerized components shall be split into separate control loops, each of which running as a separate serverless function in the control plane, and capable of maintaining and reconciling the desired state for its necessary resources.

Running each control loop in a serverless function ensures that the system is modular and can scale independently based on the specific demands of each resource type.

R2 Event-Driven Triggers: The new architecture shall support an event-driven model where these serverless functions are triggered by changes in the datastore.
This requirement ensures that the control plane operates efficiently, only taking up resources when necessary, rather than continuously polling or running processes.

R3 Elastic Scalability: Each control plane component shall be able to scale dynamically in response to fluctuating workloads.
This requirement ensures that the system can handle sudden spikes in demand without performance degradation, while minimizing idle resource usage during periods of low activity.

R4 Security and Access Controls: The system shall enforce strong authentication and authorization controls, ensuring that only authorized entities can access and modify resources.
This requirement is crucial to protect the integrity and confidentiality of the cluster, ensuring that only authorized users and services can interact with the control plane and underlying resources.

3.2 Design of the *Kubeλess* Architecture

Figure 3.1 portrays the conceptual design of a *Kubeλless* cluster. The first thing to note in this new architecture is that the datastore is now at the center of the cluster, as opposed to the API server. This fundamental shift in the design reflects the decision to move the control plane to the cloud, a choice driven by the inherent advantages of cloud-native services that allow rapid scalability and better resource utilization. This move allows us to offload responsibilities traditionally managed by the API server to the cloud provider. For instance, authentication and authorization can be offloaded to the cloud provider's identity and access management (IAM) services, which are designed to scale effortlessly and provide robust security features. Similarly, access controls and API version translation can be managed by the cloud provider's datastore. Therefore, the API server becomes no longer necessary in this new architecture. Moreover, the integrity and confidentiality of the cluster is protected, addressing **R4**.

In place of the API server, the datastore now acts as the central hub for all control plane operations. This datastore is not just a repository for the states of the objects but also a critical component that triggers control plane actions. The serverless functions, which replace traditional monolithic components like the controller manager and scheduler, (**R1**)

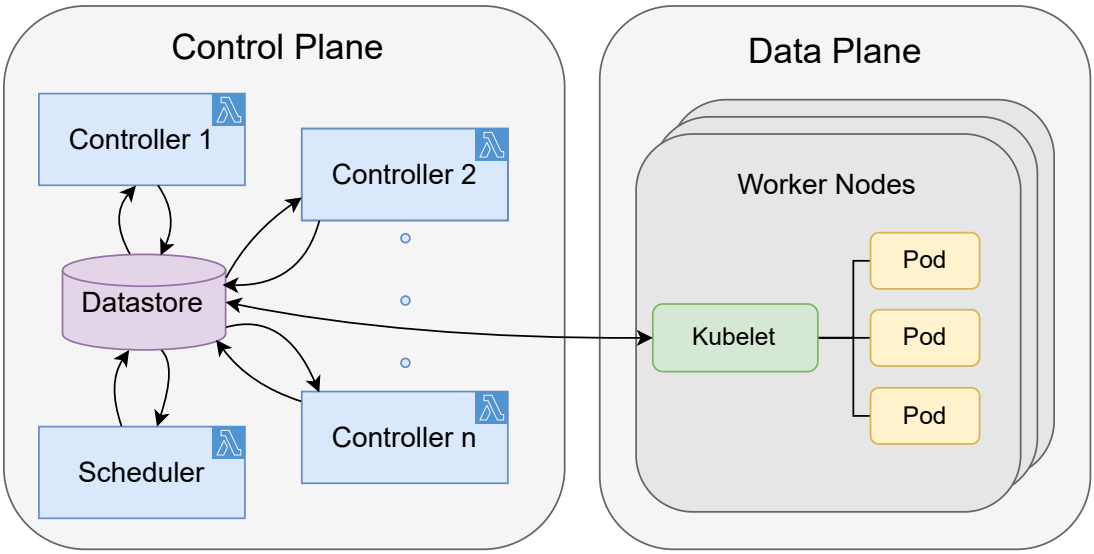


Figure 3.1: Main architectural components of a *Kubeλless* cluster

are event-driven and are triggered by changes in the datastore, addressing **R2**. This event-driven model is at the core of the *Kubeλless* architecture, ensuring that control plane operations are executed only when necessary, thereby optimizing resource usage.

Among the most important components of the *Kubeλless* architecture are the serverless controller functions, implementing the control loops of the built-in Kubernetes controllers. These functions are responsible for managing the lifecycle of a particular resource type within the cluster. When a change is detected in the datastore, the corresponding serverless function is triggered to perform the necessary actions for reconciling the actual state of the resource with the desired state specified by the user. After executing the required operations, the function updates the resource's state in the datastore, if necessary. This update can, in turn, trigger other serverless controller functions responsible for managing related resources, creating a chain of event-driven operations that ensure the cluster remains in its desired state.

Another critical component in this architecture is the serverless scheduler function, which is a special type of controller that determines the placement of pods across the cluster's nodes. When new pods are created, the scheduler function is triggered to evaluate the current state of the cluster, including node capacity and resource availability, before making scheduling decisions. This function then updates the datastore with the scheduling results, ensuring that the pods are deployed according to the optimal resource allocation.

As opposed to the control plane, the data plane in the new design remains largely un-

changed. This is because there are already various mechanisms available that can scale the data plan up and down, such as horizontal/vertical pod autoscalers or managed Kubernetes services like Google Kubernetes Engine. The only difference is that the Kubelet now communicates directly with the datastore, to receive the specifications of the pods it needs to run and to report the status of the pods it manages. This communication loop ensures that the cluster remains healthy and allows the control plane to react dynamically to changes in the cluster.

This reimagined architecture, centered around the datastore and driven by serverless functions, offers a scalable and efficient Kubernetes environment. By eliminating the need for a continuously running API server and leveraging event-driven serverless functions, *Kubeλless* minimizes resource consumption, reduces operational complexity, and improves control plane elasticity.

4

Implementation of *Kubeλess*

In this chapter, we answer **RQ2:** *"What are the practical steps required to implement a serverless Kubernetes control plane?"*. *Kubeλless* is an implementation of the serverless architecture proposed in Chapter 3 for the Kubernetes control plane. However, due to the enourmous size and complexity of the Kubernetes source code (consisting of millions of lines of code) and the limited time we have for the work, the implementation contains only on a single control loop, namely that of the job controller. The rest of the control loops follow the same structure highlighted in Section 2.2 and could be easily be adapted to the new architecture following the same steps mentioned in this section.

4.1 Cloud Provider

As previously stated in Section 3.2, the *Kubeλless* control plane was designed to operate within a cloud environment, maximizing the benefits of cloud-native services. Although this design could have been implemented on any serverless cloud provider (such as Microsoft Azure or Google Cloud), Amazon Web Services (AWS) was chosen as its foundation for several compelling factors. Firstly, AWS offers a robust suite of serverless technologies that align perfectly with the design goals of *Kubeλless*. Its comprehensive range of integrated services, such as AWS Lambda, DynamoDB, and Identity and Access Management (IAM), provides the necessary infrastructure to build a scalable, secure, and efficient serverless control plane. Additionally, AWS's well-documented services and its extensive support make it easier to develop, deploy, and test a system in the cloud, ensuring a smoother implementation process.

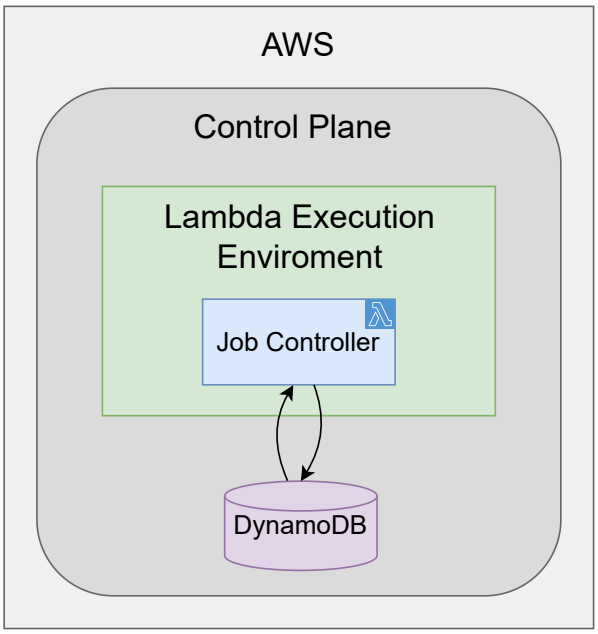


Figure 4.1: Implementation of *Kubeλess* in AWS

4.2 *Kubeλess* Architecture in AWS

In this section we explore how the main components of the serverless control plane are realized in the AWS environment, which is summarized in Figure 4.1.

The design of *Kubeλless* places the datastore at the heart of the control plane, making it essential to choose one that can efficiently handle the demands of a serverless, event-driven architecture. AWS offers several options for persistent storage, ranging from traditional relational databases (RDS) to NoSQL databases (DynamoDB). However, DynamoDB was ultimately selected as the datastore for *Kubeλless* due to its combination of scalability, low-latency performance, and seamless integration with an event-driven architecture. The native support for DynamoDB Streams allows for real-time event-driven operations, where changes in resource states trigger the appropriate control loops without the need for continuous polling or additional infrastructure. Moreover, DynamoDB has a strong functional overlap with the default Kubernetes datastore, making it relatively easy to use as a replacement.

To efficiently store and manage the state of various Kubernetes resources, the DynamoDB table in Kubeλless is structured with three key attributes. The primary key is the resource type (such as "job" or "pod"), ensuring that different types of resources are clearly separated within the table. The sort key is the resource name, allowing for quick retrieval

and organization of specific resources. The third attribute is the resource value, containing a string representation of the resource object's JSON. This JSON string encapsulates all the necessary details of the resource, making it easy for Lambda functions to parse and process. This implementation does not strictly follow the complex structure of Kubernets' default datastore, as it was simplified for our use case. However, it could easily be extended to cover the entire structure with all of its complexities.

In order to execute the control loops in response to DynamoDB triggers, they were implemented as independent AWS Lambda functions. These Lambda functions handle the core logic of the control loops, such as reconciling the desired state of resources with their actual state. For example, when a job is created or updated, the corresponding Lambda function is triggered by an event captured in DynamoDB Streams, processes the event, performs the necessary operations like creating or terminating pods, and updates the resource state in DynamoDB.

4.3 Implementation of the Serverless Job Controller

In this section, we outline the steps taken to transform the traditional job controller, as described in Section 2.2, into a serverless architecture suitable for the *Kubeλless* design. The process involved significant modifications to the way the controller operates, moving from a continuously running serverfull architecture to an event-driven, serverless model using AWS Lambda and DynamoDB.

The first step in transitioning to a serverless architecture is to create a Lambda handler function, using the *lambda* package provided by AWS. This handler is the entry point for the Lambda function and is automatically executed by Lambda whenever an event triggers the function. It is responsible for processing incoming events, determining the type of event (e.g., job creation, modification, or deletion), and initiating the appropriate actions to reconcile the job's state.

The next step is to remove the reliance of the controller on the Informer, and the watch mechanism provided by it, to retrieve changes in resources and trigger the corresponding event handlers. The Informer is an inherently serverfull component, as it requires a persistent connection to the API server to receive updates in real time. However, to eliminate the need of using the Informer, we can simply use the DynamoDB events passed to the handler. These events contain a lot of information, including the type of data modification and the actual record, which can be used to manually trigger the event handler functions.

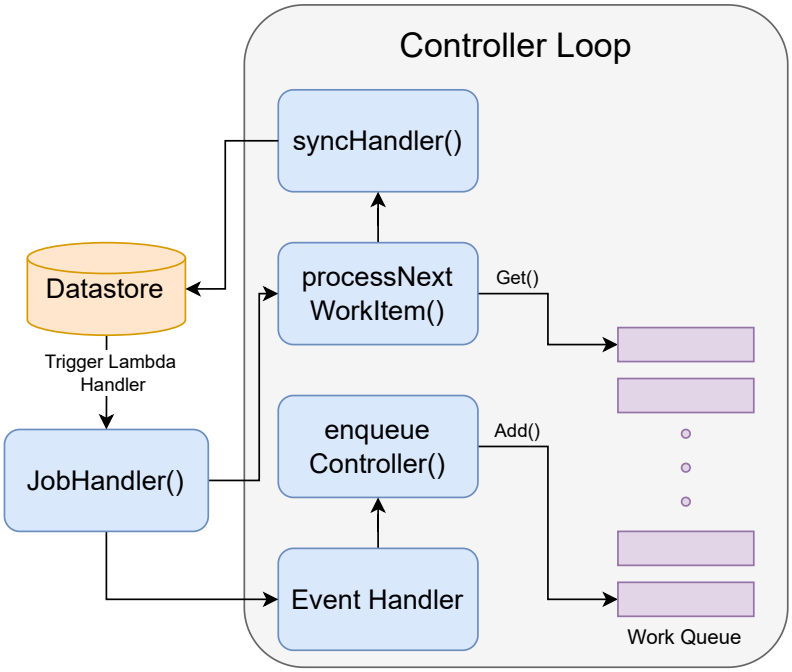


Figure 4.2: Internal components of the serverless job controller

The final step in the transformation is to modify all instances where the controller needs to commnuicate with the API Server to read and write information. In the serverless architecture, these interactions shall be directed to DynamoDB, which now serves as the central datastore. The DynamoDB API was utilized for this task, provided by AWS's *dynamodb* and *awa* packages.

Figure 4.2 shows the internal components of the resulting job contorller. As can be seen, we have tried to keep the core logic of the control loop the same in the new architecture. The only difference lines in the communication with the datastore, and how the event handlers are triggered.

5

Evaluation

In this chapter, we address **RQ3:** *"What are the performance characteristics of a serverless Kubernetes control plane, and how do they compare to the traditional Kubernetes?"*. To answer this question, we will evaluate *Kubeλess* against the traditional Kubernetes implementation through a series of microbenchmarks.

5.1 Experimental Setup

We execute the experiments in two different environments. The benchmarks for traditional Kubernetes are conducted on a single machine running Ubuntu 20.04, equipped with a 20 core Intel Xeon Silver 4210R CPU and 256 GB RAM. We use the Continuum framework [27] to emulate a 2-Node Kubernetes cluster on QEMU virtual machines with an 8 core CPU and 32 GB RAM. The Kubernetes version used for the the experiments is 1.27, equipped with custom logs to timestamp the start and end of workflows. In contrast, the benchmarks for *Kubeλess* are conducted on the AWS cloud, leveraging AWS Lambda for serverless functions and DynamoDB for state management.

In both environments, sample job descriptions are written to the datastore, either via *kubectl* or the AWS SDK for Python (Boto3), which are then sent to the job controller to be processed. As we are only interested in the performance of the control plane, the type of application chosen would not alter our results. In both environments, we repeat all experiments 5 times, to ensure the resuls are statistically significant, and report the best run.

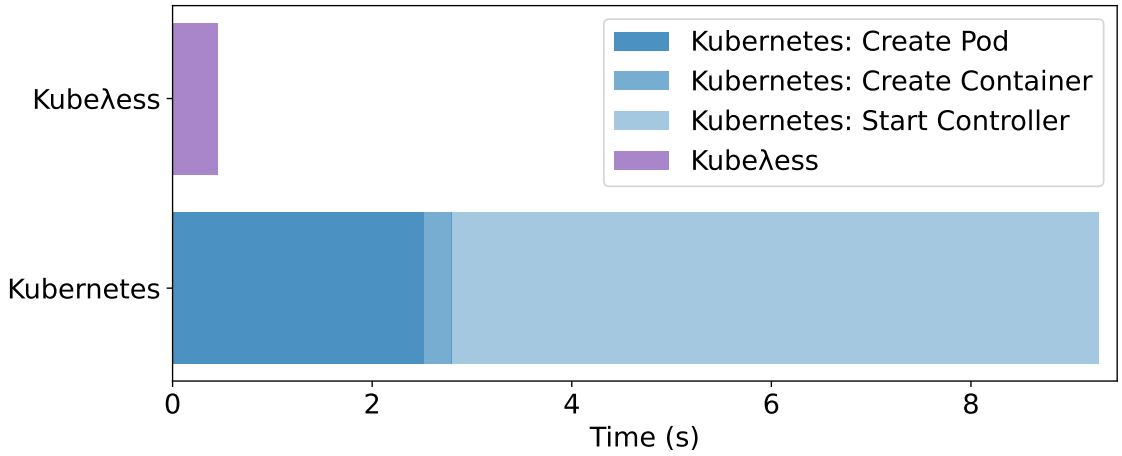


Figure 5.1: Time to start the job controller in Kubernetes and Kubeλess

5.2 Controller Startup Time

For our first experiment, we are going to measure and compare the startup times of the job controller in Kubernetes and *Kubeλess*. One of the main problems identified with Kubernetes' control plane was its inability to scale elastically due to the large size of its containers. By comparing the startup times, we can evaluate whether *Kubeλess* provides a significant advantage over traditional Kubernetes in terms of initialization speed.

We measure the startup time for Kubernetes' job controller across three distinct stages, using custom timestamps to capture the duration of each phase. The first stage involves measuring the time required to create the pod that hosts the controller manager, which includes operations like mounting volumes and setting up the pod sandbox. The second stage focuses on the time taken to create and start the container within this pod. Finally, the last stage measures the time it takes for the job controller to become fully operational after the controller manager has started. In contrast, for *Kubeλess*' job controller, we rely on the initialization times recorded in the AWS Lambda logs, which provide a direct measure of the function's startup time.

Figure 5.1 presents the results of our experiment, showing the comparison of startup times of the job controller between Kubernetes and *Kubeλess*. As was expected, the serverless job controller has a significantly faster startup time compared to Kubernetes, taking around 450. However, an interesting observation is that the time it takes for the job controller to become operational is significantly larger than the time taken to prepare and start the container for the controller manager, taking 6.48 seconds.

5.3 Pod Deployment Time

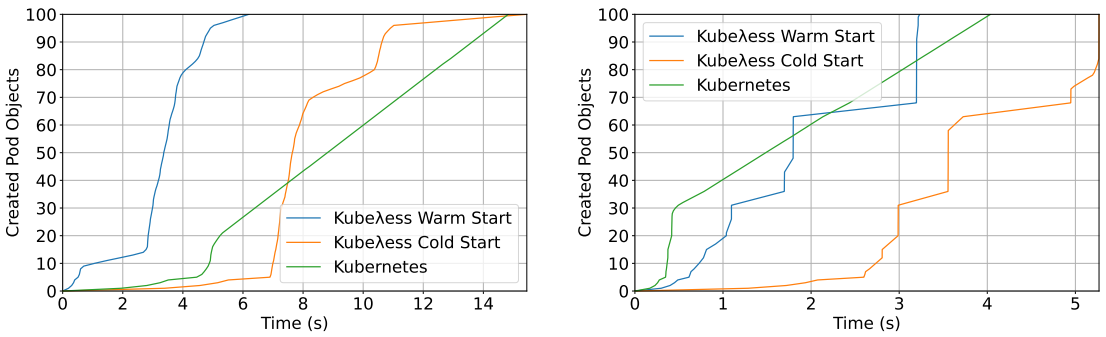
We aim to compare the deployment time of the traditional job controller with its serverless implementation. More specifically, as our architecture is focused on the control plane, we measure this time from the moment a user submits a job request until the controller creates the necessary pod descriptions and writes them in the datastore. By comparing the performances of both systems, we can assess whether the serverless control plane in *Kubeλess* can match or exceed the performance of a traditional Kubernetes control plane.

To find out how the form of a workload affects the performance of the controller, we compare two deployment methods. The first method submits 100 job requests (with a single pod per job) to the datastore, using either *kubectl* for the traditional Kubernetes setup or Boto3 for *Kubeλess*, while the second method submits a 100 pods in a single job. To gather the measurements for *Kubeλess*, we timestamp the job and pod objects the moment they are created. And for Kubernetes, we use custom timestamps to measure the time from the start of the *kubectl* command to the moment the API server receives a write request from the controller. This allows for a precise comparison of the time each system takes to process the job requests and create the necessary pods.

Moreover, to find out how the overhead added by the Lambda invocation affects the performance of *Kubeλess*, we measure both cold and warm starts. A cold start occurs when the function is invoked for the first time or after a period of inactivity, requiring the initialization of the execution environment. In contrast, a warm start occurs when the function has already been initialized, allowing subsequent invocations to be processed more quickly. To warm up the functions, we send dummy request to trigger them before conducting the actual experiments.

The results gathered from the experiments are presented in Figure 5.2. The x-axis represents the time in seconds, starting from when the job request is submitted until the neccessary amount of pods are registered in the datastore. And the y-axis represents the cumulative number of pods created.

The first thing to note is that there is a noticeable difference in performance between the cold and warm runs of *Kubeλess*. However, from the previous experiment it was found that the average initialization time for our function was around 450 milliseconds. Thus, this small overhead could not have cause such a significant difference in performance. After a closer inspection of the logs, it was found that the initial request to DynamoDB takes about 2 seconds, if the function is invoked with a cold start, while the subsequent requests only take several milliseconds. As the performance of *Kubeλess* is slowed down significantly



(a) Using a 100 job requests with a single pod

(b) Using a single job request with 100 pods

Figure 5.2: Time to create 100 pod objects in Kubernetes and *Kubeλess* using different deployment methods

by this initial DynamoDB request (not the startup times of the serverless function), and this overhead is typically not shared by other datastores, we choose to focus on the warm starts instead for a more precise comparison.

Another important observation is that *Kubeλess* performs considerably better than Kubernetes for the first deployment method, but not for the second. It takes *Kubeλess* 6.46 seconds to create 100 pod descriptions and write them to the datastore with the first method, which is more than twice as fast compared to traditional Kubernetes. One of the reasons for this performance disparity is the overhead added by having to call *kubectl* a 100 times. Additonaly, the controller in Kubernetes processes requests from a rate limiting queue, which while ensuring stability, can introduce latency with a large number of requests. This highlights the significant improvement in scalability of our system, as it is able to process jobs in parallel with minimal overhead.

Deploying 100 pods in a single job request presents much lower deployment times for both systems, with *Kubeλess* taking 3.29 seconds on a warm run and Kubernetes taking 4.14 seconds. The difference in performance between the systems is minimal in this scenario because both employ the same processing logic for handling a single job request. The slight edge in *Kubeλess* can be attributed to the difference in backend infrastructure—*Kubeλess* interacts directly with DynamoDB, optimized for fast, single-purpose transactions, whereas Kubernetes relies on the API server, which introduces more latency due to its more complex interactions and consistency checks across the cluster.

5.4 Limitations

We identified multiple limitations in the setup and design of experiments that could affect the relevance of our findings. Firstly, the number of concurrent Lambda executions was limited to only 10 by AWS (while the default is 1000), as the account was newly created. This lower concurrency limit may have artificially constrained the scalability and performance of *Kubeλess* in our experiments, potentially underestimating its full capability.

Furthermore, all experiments were conducted using synthetic workloads rather than real production workloads. As the implementation did not include a data plane where pods could have been deployed and applications executed, it would not have been feasible to monitor their state or evaluate the system under real-world conditions. As a result, the performance observed in these experiments might not fully reflect the behavior of *Kubeλess* in a production environment with live workloads.

5.5 Summary

To understand the performance charasteristics of *Kubeλess* and how they compare to the traditional Kubernetes implementation, we conducted a series of experiments. In terms of startup time, *Kubeλess* showed a significant improvement, with AWS Lambda's serverless architecture reducing the initialization time to around 450 milliseconds, compared to the 9.28 seconds it took Kubernetes.

In terms of pod deployment, *Kubeλess* outperformed Kubernetes when deploying multiple jobs with single pods, completing the task in 6.46 seconds when warmed up, which is more than twice as fast as Kubernetes. However, the results for deploying a single job with multiple pods showed a smaller performance gap, with *Kubeλess* taking 3.29 seconds and Kubernetes taking 4.14 seconds. The smaller difference can be attributed to both systems employing similar processing logic for single requests, but with *Kubeλess* benefiting from faster backend infrastructure.

In summary, the evaluation shows that *Kubeλess* can provide a more elastic and faster control plane than traditional Kubernetes. The system's ability to quickly scale and handle workloads can make it a compelling alternative, especially in scenarios where rapid elasticity is required.

6

Related Work

Over the past decade, significant research has focused on analyzing and enhancing the scalability of Kubernetes, given its prominence as a container orchestration platform. Some studies aimed to provide better mechanisms for Kubernetes' Horizontal Pod Autoscaler (HPA), by incorporating additional metrics such as traffic characteristics [33] or response latency [34]. Additionally, several works centered on implementing custom HPAs using machine learning or heuristic analysis [35, 36, 37]. In parallel, research on Kubernetes' Vertical Pod Autoscaler (VPA) has also continued to evolve with a focus on optimizing resource allocation based on observed usage patterns through systems such as RUBAS [38].

While the existing body of research has made substantial advancements in enhancing Kubernetes' scalability, these efforts have failed to address the underlying elasticity of the control plane itself. Kubeλess has filled this gap in the literature by introducing a serverless design able to manage highly variable workloads more efficiently. Its design was inspired by emerging trends in serverless architectures, found in various modern systems such as LambdaFS, Unum, and Ilúvatar. LambdaFS [39] is a serverless metadata service, enabling an elastic and high-performance architecture crucial for large-scale distributed file systems. Unum [40] is a decentralized orchestration system for serverless applications which reduces operational costs and improves flexibility. Ilúvatar [41] is a modular control plane for serverless computing, which reduces latency and allows for more efficient resource management.

7

Conclusion

Kubernetes, while being the most widely adopted container orchestration platform, faces significant challenges in providing the necessary level of elasticity in its control plane, particularly when managing large and dynamic workloads. The traditional architecture's inability to scale rapidly in response to workload fluctuations often leads to inefficiencies in resource utilization and increased operational costs. To address this critical challenge of elasticity in Kubernetes' control plane, we identified three research questions, the answers to which are given in the following section.

7.1 Answering Research Questions

RQ1 How can the Kubernetes control plane be redesigned to operate entirely within a serverless architecture?

To address this question, we developed the *Kubeλess* architecture by reimagining the core components of Kubernetes' control plane as serverless functions. In this new design, each control loop within Kubernetes, such as the job controller or scheduler, is transformed into a standalone serverless function. Moreover, the API server is removed in favor of a cloud-native datastore, which now acts as the central hub for triggering the functions. By shifting the control plane to a serverless model, the architecture eliminates the need for continuously running components, which traditionally consume resources even during periods of low demand.

RQ2 What are the practical steps required to implement a serverless Kubernetes control plane?

The implementation of *Kubeλess* involved several key steps: selecting a cloud provider capable of supporting a serverless architecture, structuring the datastore to handle event-driven operations, and transforming control loops into serverless functions. The practical steps detailed in Chapter 4 demonstrate how AWS services, particularly AWS Lambda and DynamoDB, were employed to achieve a responsive and scalable control plane.

RQ3 What are the performance characteristics of a serverless Kubernetes control plane, and how do they compare to the traditional Kubernetes?

The performance evaluation of *Kubeλess* demonstrated that it can significantly reduce the startup time of the controllers and achieves more efficient pod deployments in high-demand scenarios.

7.2 Limitations and Future Work

Due to the enormous size and complexity of Kubernetes, and the limited scope of this thesis, our work implemented only a single control loop as a prototype. Future work should extend the serverless architecture to other control plane components to fully realize its benefits. Additionally, the experiments conducted were based on synthetic workloads; thus, testing *Kubeλess* in a real-world production environment would provide further insights into its performance and scalability.

In conclusion, *Kubeλess* provides a promising direction for future Kubernetes architectures, demonstrating that serverless computing can effectively address the scalability and elasticity challenges of cloud-native environments. The insights gained from this research lay the groundwork for further exploration and refinement in the deployment of scalable, cost-efficient Kubernetes clusters.

References

[1] Maria A. Rodriguez and Rajkumar Buyya. **Container-based cluster orchestration systems: A taxonomy and future directions**. *Software: Practice and Experience*, **49**(5):698–719, 2019. 1

[2] Carmen Carrión. **Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis**. *Journal of Grid Computing*, **20**(4):42, 2022. 1

[3] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. **A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks**. *Applied Sciences*, **9**(5), 2019. 1

[4] Eddy Truyen, Nane Kratzke, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. **Managing Feature Compatibility in Kubernetes: Vendor Comparison and Analysis**. *IEEE Access*, **8**:228420–228439, 2020. 1

[5] Docker. **Swarm mode overview**, 2024. [Online; accessed May 15, 2024] `https://docs.docker.com/engine/swarm/`. 1

[6] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. **Mesos: a platform for fine-grained resource sharing in the data center**. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association. 1

[7] Cloud Native Computing Foundation. **CNCF Annual Survey 2023**, 2023. [Online; accessed May 15, 2024] `https://www.cncf.io/reports/cncf-annual-survey-2023/`. 1

[8] Benjamin Schmeling and Maximilian Dargatz. *The Impact of Kubernetes on Development*, pages 1–57. Apress, Berkeley, CA, 2022. 1

[9] Kubernetes. **Kubernetes User Case Studies**, 2024. [Online; accessed May 15, 2024] `https://kubernetes.io/case-studies/`. 1

[10] Portworx. **2021 Kubernetes Adoption Survey**, 2021. [Online; accessed May 21, 2024] `https://www.purestorage.com/content/dam/pdf/en/analyst-reports/ar-portworx-pure-storage-2021-kubernetes-adoption-survey.pdf`. 1

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. **Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade**. *Queue*, **14**(1):70–93, 2016. 1, 2

[12] Ahmed Barnawi, Sherif Sakr, Wenjing Xiao, and Abdullah Al-Barakati. **The views, measurements and challenges of elasticity in the cloud: A review**. *Computer Communications*, **154**:111–117, 2020. 2

[13] Kubernetes. **Kubernetes Components**, 2024. [Online; accessed May 15, 2024] `https://kubernetes.io/docs/concepts/overview/components/`. 2

[14] Salman Taherizadeh and Marko Grobelnik. **Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications**. *Adv. Eng. Softw.*, **140**(C), 2020. 2

[15] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association. 2, 3

[16] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. **Cloud Programming Simplified: A Berkeley View on Serverless Computing**. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, 2019. 2

[17] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. **The SPEC cloud group's research vision on FaaS and serverless architectures**.

In *Proceedings of the 2nd International Workshop on Serverless Computing*, WoSC '17, page 1–4. Association for Computing Machinery, 2017. 2

[18] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan S. Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**. *CoRR*, **abs/2206.03259**, 2022. 3

[19] Naim A. Kheir. *Systems Modeling and Computer Simulation*. Routledge, New York, NY, USA, 2nd ed. edition, 1996. 4

[20] Yair Levy and Timothy Ellis. **A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research**. *InformingSciJ*, **9**:181–212, 2006. 4

[21] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The atlarge vision on the design of distributed systems and ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, Proceedings - International Conference on Distributed Computing Systems, pages 1765–1776, United States, 2019. Institute of Electrical and Electronics Engineers Inc. 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019 ; Conference date: 07-07-2019 Through 09-07-2019. 5, 11

[22] Richard R. Hamming. *Art of Doing Science and Engineering: Learning to Learn*. CRC Press, 1st ed. edition, 1997. 5

[23] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *J. Manage. Inf. Syst.*, **24**(3):45–77, 2007. 5

[24] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc, New York, NY, USA, 1991. 5

[25] Gernot Heiser. **Systems Benchmarking Crimes**, 2019. [Online; accessed May 27, 2024] `http://www.cse.unsw.edu.au/~Gernot/benchmarking-crimes.html`. 5

[26] John Ousterhout. **Always measure one level deeper**. *Commun. ACM*, **61**(7):74–83, 2018. 5

[27] 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*, ICPE '23 Companion, page 181–188, New York, NY, USA, 2023. Association for Computing Machinery. 5, 19

[28] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, Tony Ross-Hellauer, René Schneider, Jon Tennant, Ellen Verbakel, Helene Brinken, and Lambert Heller. *Open Science Training Handbook*. Zenodo, 2018. 5

[29] Mark D Wilkinson, Michel Dumontier, Ijsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, Jildau Bouwman, Anthony J Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J G Gray, Paul Groth, Carole Goble, Jeffrey S Grethe, Jaap Heringa, Peter A C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J Lusher, Maryann E Martone, Albert Mons, Abel L Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. **The FAIR Guiding Principles for scientific data management and stewardship**. *Nature Scientific Data*, **3**(1):160018, 2016. 5

[30] Emery D. Berger, Stephen M. Blackburn, Matthias Hauswirth, and Michael W. Hicks. **A Checklist Manifesto for Empirical Evaluation: A Preemptive Strike Against a Replication Crisis in Computer Science**, 2019. [Online; accessed May 27, 2024]`https://blog.sigplan.org/2019/08/28/a-checklist-manifesto-for-empirical-evaluation-a-preemptive-strike-against-a-replicati` 5

[31] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is big data performance reproducible in modern cloud networks?** In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*, pages 513–527. USENIX Association, 2020. 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020 ; Conference date: 25-02-2020 Through 27-02-2020. 5

[32] Matthijs Jansen. **Columbo: A Reasoning Framework for Kubernetes' Configuration Space**. Under review, 2024. 7

[33] Le Hoang Phuc, Linh-An Phan, and Taehong Kim. **Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure**. *IEEE Access*, **10**:18966–18977, 2022. 25

[34] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. **pHPA: A Proactive Autoscaling Framework for Microservice Chain**. In *APNet 2021: 5th Asia-Pacific Workshop on Networking, Shenzhen, China, June 24 - 25, 2021*, pages 65–71. ACM, 2021. 25

[35] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. **Machine Learning-Based Scaling Management for Kubernetes Edge Clusters**. *IEEE Trans. Netw. Serv. Manag.*, **18**(1):958–972, 2021. 25

[36] Guangba Yu, Pengfei Chen, and Zibin Zheng. **Microscaler: Cost-Effective Scaling for Microservice Applications in the Cloud With an Online Learning Approach**. *IEEE Trans. Cloud Comput.*, **10**(2):1100–1116, 2022. 25

[37] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. **Adaptive Microservice Scaling for Elastic Applications**. *IEEE Internet Things J.*, **7**(5):4195–4202, 2020. 25

[38] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. **Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes**. In Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama, editors, *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, pages 33–40. IEEE, 2019. 25

[39] Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. **λFS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions**. In Tor M. Aamodt, Michael M. Swift, and Natalie D. Enright Jerger, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 394–411. ACM, 2023. 25

[40] David H. Liu, Amit Levy, Shadi A. Noghabi, and Sebastian Burckhardt. **Doing More with Less: Orchestrating Serverless Applications without an Orchestrator**. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 1505–1519. USENIX Association, 2023. 25

[41] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. **Ilúvatar: A Fast Control Plane for Serverless Computing**. In Ali Raza Butt, Ningfang Mi, and Kyle Chard, editors, *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2023, Orlando, FL, USA, June 16-23, 2023*, pages 267–280. ACM, 2023. 25

Appendix A

Reproducibility

A.1 Abstract

Kubeλess is a prototype of the architecture proposed in Chapter 3, implementing Kubernetes' job controller as a serverless function in AWS. In this section, we will describe the details of the artifact, how to use it, and how to reproduce the results.

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

- **Run-time environment:** Amazon Linux 2023
- **How much disk space required (approximately)?:**
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to access

The implementation can be accessed on GitHub.

A.4 Installation

To setup the required infrastructure in AWS, we need to:

1. Create a Lambda with 128 MB memory, running Amazon Linux 2023 on x86_64 architecture.
2. Create a zip of the code following the steps mentioned here and upload it to Lambda

3. Create a DynamoDB table, setting resource-name (string) and resource-type (string) as sort key

4. Add a trigger to your Lambda, setting DynamoDB as the source

A.5 Evaluation and expected results

The expected results are similar to the results discssed in Chapter 5