Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Towards enabling Serverless Computing across the Edge-Cloud Continuum

Author: Emilian Simion (2681247)

professor:	Zhiming Zhao	(z.zhao@uva.nl)
1st supervisor:	Uraz Odyurt	(u.odyurt@uva.nl)
2nd supervisor:	Yuandou Wang	(y.wang8@uva.nl)

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

August 11, 2023

Abstract

In recent years, serverless computing has emerged as a transformative paradigm for building and deploying applications without the burden of managing traditional server infrastructure. Concurrently, edge computing has gained traction as a critical solution to reduce latency and enhance real-time data processing by bringing computation closer to data sources. However, while integrating these two powerful concepts – serverless and edge computing – has historically posed challenges, recent advancements have showcased the feasibility of running serverless platforms within edge environments. This paper introduces Knative Edge, a novel serverless platform that seamlessly orchestrates the deployment of serverless functions across both edge devices and cloud environments. Knative Edge leverages the unique characteristics of edge computing, such as proximity to data and reduced network latency, while harnessing the scalability and flexibility of serverless computing. By combining these two paradigms, Knative Edge empowers developers to build highly responsive and efficient applications that span the cloud-edge continuum. We present the architectural design, key components, and deployment workflow of Knative Edge, and provide experimental results showcasing its performance benefits. Our results demonstrate that Knative Edge can make efficient use of both cloud and edge resources by dynamically offloading function executing from the edge to the cloud periods of high activity, while reducing the overall application latency and increasing request throughput compared to an edge-only deployment.

Contents

1 Introduction **1**

2 Background **3**

2.1 Edge computing 3

2.1.1 Edge-Cloud Continuum 6

2.2 Serverless computing 8

2.2.1 Kubernetes 10

2.2.2 Knative 12

3 Overview **17**

4 Design **19**

4.1 Requirement analysis 19

4.2 Platform evaluation 21

5 Implementation **27**

5.1 Edge Controller 27

5.1.1 Cloud to Edge replication 27

5.1.2 Edge to Cloud offloading 30

5.2 Edge Operator 34

5.3 Deployment 35

6 Evaluation **39**

6.1 Results 45

7 Discussion **59**

7.1 Future Challenges 62

7.1.1 Platform interoperability 63

7.1.2 BaaS at the Edge 63

7.1.3 Edge-native serverless 65

8 Threats To Validity **67**

9 Related Work **69**

10 Conclusion **71**

References **73**

1

Introduction

The use of cloud computing has dramatically changed the way applications are developed and deployed. One of the key innovations in cloud computing is serverless computing, which enables developers to run their code without having to manage any underlying infrastructure. This eliminates many of the traditional barriers to entry for developing and deploying applications, including the need for managing and operating complex computing infrastructure and its associated costs.

With our work, we aim to implement a serverless platform that operates seamlessly on the Edge-Cloud continuum and can make use of the best characteristics of each technology involved. The implementation shall extend an existing serverless platform and transform it to support the requirements we set out for the platform.

Our contribution to the field is the implementation and evaluation of a serverless platform that operates on the Edge-Cloud continuum, providing insights into the trade-off and challenges involved in developing such a platform. The results of this study will be valuable for developers, cloud providers, and researchers interested in serverless computing and edge computing.

The motivation behind this thesis is to investigate the unrealized potential of the Edge-Cloud continuum for serverless computing. By implementing a serverless platform on this continuum, we aim to gain practical insights into the intricacies and challenges associated with this architecture. Our goal is to contribute to the advancement of serverless computing in cloud and edge environments by exploring the platform's feasibility and benefits in a real-world context.

In order to achieve our goal, we propose the following set of research questions that would guide the development of the serverless platform:

RQ 1. What are the advantages and disadvantages of using serverless at the Edge? Edge resources have a limited capacity by nature and serverless functions promise efficient resource utilization through their scale to zero capabilities. However, serverless computing comes with its own challenges, such as cold-starts and increased latencies. As such, we aim to investigate how serverless computing behaves when running at the Edge.

RQ 2. How can one deploy serverless functions transparently across Edge and Cloud? Since serverless functions abstract away the infrastructure on which they run, the serverless platform has the ability to choose whichever location it thinks is best suited to serve requests from. We aim to find out whether the research community has investigated approaches that take advantage of this fact and we aim to design a solution which addresses the research question at hand.

RQ 3. Can a serverless framework enable a Edge-Cloud continuum? Having a platform which is able to intelligently schedule requests to both the Edge and the Cloud, while also keeping abstracting away the complexity of it from an application developer is highly desirable. Such a platform has the potential of combining the best characteristics of both, the computing and network performance of cloud environments and the data locality of edge environments. As such, we are interested in whether serverless computing offers enough flexibility in order to achieve such a vision.

RQ 4. What are the challenges in developing an Edge-Cloud serverless platform? The serverless execution model has its challenges, such as cold-starts and high degree of dependability on external software solutions like storage, databases, and events routing, and adding an edge layer to the serverless cloud platforms has the potential to amplify such drawbacks. We aim to identify and discuss such issues in order to define future challenges and research topic which could be explored.

2

Background

2.1 Edge computing

Edge computing is a computing paradigm that brings processing, data storage, and applications closer to the edge of the network, where end devices such as IoT[1] devices and smartphones generate and consume data (1). The concept has gained significant attention in recent years, as it offers a number of benefits over traditional centralized computing architectures. Edge computing aims to reduce latency, increase computing efficiency, and save on bandwidth by processing data near the source rather than sending it to a remote centralized cloud location. This allows for real-time data processing and decision making at scale, which is essential for many use cases in industries such as transportation, manufacturing, and healthcare.

The *edge of the network* refers to the part of a computer network that is closest to the end users or devices, including sensors, smartphones, and IoT devices. The edge can be comprised of small data centers that are located close to the end devices and with minimal network hops, or it could be part of the same internal network as the devices.

Edge computing is a rapidly evolving field and involves a range of technologies, including low-power edge devices, distributed computing, and edge-optimized software and algorithms. The development of 5G networks, the proliferation of connected devices, and the growing demand for real-time and data-intensive applications are driving the adoption of edge computing across a range of industries such as healthcase, transportation, manufacturing, and smart cities.

[1] Internet of Thing, or IoT, refers to the billions of physical devices around the world that are now connected to the internet, collecting and sharing data.

2. BACKGROUND

Content Delivery Networks (CDNs) are considered to be the precursor to edge computing. The rise of the Internet and the increasing demand for high-performance and reliable delivery of content, such as images, videos, and web pages, led to the development of CDNs by Akamai in 1999 (2, 3). CDNs use a network of edge servers located in multiple geographic locations to cache and deliver content closer to the end user, reducing latency and improving user experience. The success of CDNs in delivering content efficiently demonstrated the benefits of storing data and serving requests closer to the end users.

A standard edge computing architecture comprises of the following fundamental components:

- *Edge devices.* These devices are strategically positioned at the periphery of the network and are responsible for data generation. Examples include sensors, smartphones, and various IoT devices.

- *Edge servers.* These servers are situated in proximity to edge devices. They facilitate data storage, processing, aggregation, and analysis. Edge servers may be deployed in compact data centers near end-users or devices, as well as within the same internal network. However, it's important to note that their available power and storage capacity are considerably more limited compared to cloud resources.

- *Centralized Data Centers or Cloud.* These serve as centralized hubs for data storage and processing. They can also offer additional processing power and storage on demand.

In an edge computing architecture, data is generated at the edge devices and transmitted to the edge server for processing. The processed data can then be transmitted to the Cloud for analysis, storage, and distribution. The edge servers can also run applications and services locally, reducing the need for communication with the central network and improving performance and reliability of the applications. Edge computing has several benefits which include:

- *Low latency.* Edge computing enables processing of data closer to the source, reducing the latency and improving performance. This is particularly important for applications that require real-time processing or have to make time-critical decisions.

- *Improved reliability.* By processing data locally, edge computing reduces the dependence on centralized data centers and reduces the risk of network outages.

- *Better data privacy.* Edge computing enables data to be processed locally, reducing the need to transmit sensitive data over the network. This improves the privacy and security of the data.
- *Cost savings.* Edge computing can reduce the cost of data transmission and storage by processing and storing data locally.
- *Energy efficiency.* In contrast the high power usage of large data centers, edge locations have to be energy-efficient in order to maximize their potential. In this regard, we see edge providers move towards more efficient computing resources, such as low-power CPU architectures like ARM.

However, edge computing also presents a number of challenges and limitations that must be considered before deploying edge-based solutions.

- *Deployment complexity.* Edge computing requires the deployment of edge servers, which can be complex and challenging.
- *Limited resources.* Edge locations have limited processing power and storage capacity compared to large data centers, limiting its real-world uses.
- *Lack of standardization.* There is currently a lack of standardization in the field, making it difficult for companies to integrate edge computing into their existing infrastructure.
- *Maintenance and upkeep.* Computing resources located in remote places requires ongoing maintenance and upkeep, including software updates and security patches. This can be time-consuming and costly.
- *Integration with existing infrastructure.* Integrating edge computing with existing infrastructure and applications can be challenging, requiring significant time and effort to migrate.

Edge computing plays a pivotal role across various industries and fields, for which the most prominent uses cases are:

Manufacturing. Manufacturing stands out as a prime beneficiary of edge computing's capabilities (4, 5). This technology enables real-time monitoring and control of manufacturing processes, leading to enhanced efficiency and cost reduction. By analyzing data from sensors and devices, it is possible to identify anomalies and predict failures, effectively minimizing downtime and boosting productivity.

Healthcare. In the healthcare sector, edge computing delivers substantial value by enabling real-time monitoring of medical devices, thereby improving patient safety and reducing costs and staff workload (6, 7). This technology empowers medical professionals to make prompt decisions and ensures the smooth operation of critical equipment.

Smart Cities. Edge computing is actively researched for its application in smart cities (8, 9). It facilitates real-time monitoring and control of urban infrastructure, addressing the evolving needs of modern cities. From optimizing energy consumption to enhancing public safety, edge computing contributes to the seamless functioning of these dynamic environments.

Transportation. The transportation industry leverages edge computing to monitor and manage motor traffic in real-time (10, 11). These applications utilize an array of sensors, including cameras, weight sensors, and magnetic sensors. This localized processing is vital to handle the immense volume of traffic data efficiently, without overloading centralized cloud systems. Although it shares similarities with smart cities, transportation is a distinct use case due to its focus on motor traffic and is not limited to the bounds of cities, whereas smart cities encompass a broader range of applications used in urban environments.

2.1.1 Edge-Cloud Continuum

Cloud and edge computing are well-established concepts in the existing literature. However, the emergence of the *Edge-Cloud continuum* introduces a novel trend for which the definition is not as clearly established as it is for the former concepts. In this section, we closely examine the relevant literature pertaining to this term and aim to clarify the definition we adopt for the *Edge-Cloud continuum*.

The literature contains several terms which are either synonymous or are closely related to Edge-Cloud continuum: IoT-Fog-Cloud Continuum (12, 13), Edge-to-Cloud continuum (12), cloud-to-things continuum (14, 15). Although not all the terms are equivalent, all of them share few common characteristic:

- *Hierarchical Continuum.* The term describes a hierarchical structure where computing resources are organized in a continuum, spanning from the edge (close to data source) to the cloud (centralized data center). This implies a spectrum of processing and storage capabilities, enabling data to traverse seamlessly between different layers, not too dissimilar to the concept of fog computing.

- *Resource awareness.* The continuum is aware of the moment to moment utilization of resources across the continuum, and ensures optimal utilization of edge and cloud resources based on specific application needs.

- *Real-time Responsiveness.* A common element is the emphasis on real-time or near-real-time data processing and decision-making, necessitating the utilization of edge resources to minimize latency.

- *Dynamic Adaptation.* The characteristic involves the dynamic adaptation of computational tasks, data storage, and processing based on changing conditions and demands within the continuum. In such a scenario, the continuum should have the ability to move compute units seamlessly across many different compute nodes.

The primary use cases for an Edge-Cloud continuum are IoT (14) and mobile environments (16, 17). While these two environments share many similarities, they also exhibit distinct requirements and constraints. For IoT devices, the emphasis lies on executing tasks on energy-efficient, resource-constrained devices, with a high degree a heterogeneity among devices. On the other hand, mobile environments focus on device mobility and operates within dynamic settings, and they necessitate rapid adaptation to changing network conditions.

In their systematic literature review regarding the "cloud continuum", Moreschini et al. (12) propose a new definition of the cloud continuum as "an extension of the traditional Cloud towards multiple entities (e.g. Edge, Fog, IoT) that provide analysis, processing, storage, and data generation capabilities." In contrast to the cloud-centric approach, Nastic et al. (18) note that a "more holistic view" is needed in order to fully capture the potential of the Edge-Cloud continuum. The authors "view Edge as a main constituent of a large-scale, geographically distributed and hierarchically organized compute continuum. [...] The Edge-Cloud continuum spans from the centralized cloud to the far edge of the infrastructure to offer computing, networking and data resources."

Given the diversity of viewpoints in the existing literature concerning the Edge-Cloud continuum, we adopt the following definition:

> The Edge-Cloud continuum embodies a dynamically interconnected and hierarchical arrangement of computational entities, including but not limited to Edge, Fog, and IoT devices, seamlessly extending beyond the conventional confines of the Cloud infrastructure. This continuum converges multiple tiers of

computing resources, data storage, analysis, and generation capabilities, spanning a spectrum from the centralized Cloud environment to the remote edges of the infrastructure.

2.2 Serverless computing

Traditionally, software development, whether in commercial or academic settings, primarily occurred on on-premise servers. Nonetheless, the operation and maintenance of such infrastructure demanded specialized personnel, including but not limited to hardware IT technicians, network specialists, and IT support personnel. As technology advanced and data center scales expanded, a new paradigm emerged with the rise of Cloud providers. These providers extended their infrastructure as a service (IaaS), ushering in a novel approach. This approach enabled Cloud users to cut costs associated with physical server maintenance while retaining a remarkable level of flexibility in shaping their infrastructure. Within the IaaS model, users gained the ability to configure virtual machines (VMs) and establish networking between them using the tools and APIs furnished by the cloud platform. This shift towards Cloud-based IaaS models revolutionized the landscape of software development and deployment.

To further reduce the complexity and cost of maintaining software solutions, software platforms started being offered under the name of Platform as a Service (PaaS). In the new system, the cloud provider would be responsible for the OS and other software dependencies, such as monitoring and logging. These solutions built on top of existing IaaS and standardized, within each cloud platform, the way the software was being configured and distributed. However, PaaS offered much less flexibility to developers than IaaS, which meant that the latter was still very much popular.

At the same time, much progress was made in virtualization and container technology. With the growing popularity of Docker (19), and Google's open-source container orchestrator Kubernetes (20), an autonomous system for managing infrastructure, these technologies allowed for elasticity and autoscaling of applications, while also decreasing the costs of maintaining the underlying infrastructure. Although the automation of the infrastructure was made possible, the configuration and maintenance of such systems remained non-trivial. Jonas et al. (21) argue in their paper that "autoscaling requires determining the need to scale; picking the type and number of servers to use; requesting the servers; waiting for them to come online; configuring them with the application; confirming that no errors occurred; instrumenting them with monitoring tools; and sending traffic at them to

test them". In the end, such tools highlighted the need for developers to abstract away the infrastructure and provided a foundation for technologies that would address this, namely *serverless computing*.

The term serverless is a misnomer and does not mean that there are no servers, but rather that developers do not have to account for the underlying infrastructure. Although sharing many similarities with PaaS and shared web hosting solutions, serverless differs through its autoscaling capabilities, strong isolation, platform flexibility, and service ecosystem support (21).

Function as a Service (FaaS) Serverless computing has been largely associated with Function as a Service (FaaS). In FaaS, user code is decomposed into small *functions* which are executed on resources provisioned and managed by the cloud provider. Functions are *short-lived* and their execution is *event-based*, meaning that they have a limited time to respond and they execute only when events need to be processed. Functions are executed based on *events*, which can come in different forms such as HTTP requests, cloud messages, scheduled triggers, or other vendor specific events.

Two main characteristics of the FaaS execution environment are the *zero-scaling* capabilities, meaning that the system allows for no resources to be provisioned to the user if their functions are not being executed, and the fast provisioning time, which allows for resources to be allocated quickly in response to workloads. In order to "have some skin in the game" (22), providers charge for the time resources are being actively used, not for the time the resources are allocated. In this way, the providers need to actively optimize the way the system is performing to both retain users and efficiently utilize computing resources.

Cold and warm instances Serverless functions are considered *stateless*, which means that state is not shared between invocations of the same function. Even so, serverless providers might choose to keep instances of the functions running and ready to respond quickly to reduce the function provisioning overhead. In this regard, we can differentiate two types of function instances: *cold* instances which need to retrieve the function code and initialize it, and *warm* instances which already have the functions initialized and can immediately respond to requests. The overhead that the initialization of the function adds - creating an instance, downloading libraries and code, and starting the function runtime - is commonly known as *cold-start*.

Backend as a Service (BaaS) Due to functions being stateless, supporting services are required to perform complex work. Such services are commonly known as Backend as a Service (BaaS) and they are usually being offered by the same cloud provider that hosts the serverless platform. BaaS offerings differ from one serverless platform to another, however there are three common categories of services we can identify: *object storage* (AWS S3, Azure DataLake, GCP Cloud Storage, etc), *document storage* (AWS DynamoDB, Azure CosmosDB, GCP Cloud Firestore, etc), and *messaging* (Amazon SNS, Amazon SQS, Azure EventHub, Azure EventGrid, etc). These categories perform different roles: object storage allows for efficient storage and retrieval of structured or unstructured data on the cloud, while document storage allows for efficient storage, indexing, and querying of structured or semi-structured data, and messaging services allow for the composition of FaaS pipelines through the use of custom events as triggers for functions.

Orchestrating functions Stateful serverless can be achieved by combining FaaS with BaaS, however even basic communication and synchronization patterns, such as broadcast or global barriers, can be hard to achieve. To combat the complexity, serverless providers have introduced *function orchestrators*, which provide support for all kinds of execution patterns, such as fan-in and fan-out or function chaining. The orchestrator implementations are specific to one serverless platform, meaning that they can take advantage of the internals to improve efficiency and performance, but it also means that they are restricted to a specific platform. Notable examples are AWS Step Function, Azure Durable Function.

2.2.1 Kubernetes

Kubernetes (23) is an open-source container orchestration system which is used for managing the deployment, scaling, and continuous operation of applications within a computing cluster. It achieves this through a set of components running on each node of the cluster which coordinate the operation of the cluster and through several high-level abstractions that allow extending the functionality of Kubernetes.

Every node in a Kubernetes cluster, regardless of its role, runs at least two components: *kubelet* and *kube-proxy*. The former is responsible for maintaining the running state of the node and coordinates the local containers inside the nodes, making sure that each container is configured and running appropriately. The latter is a network proxy and load balancer implementation that is responsible for the correct routing of network packets between the different containers running on the cluster.

Control plane In order to coordinate the nodes in a cluster, Kubernetes defines a set of components which together make up the *control plane*. At least one node in the cluster must be running the control plane in order to have a functioning cluster, and several nodes can run the control plane in order to provide high availability.

The first component of the control plane is the *API server*. It is responsible for managing the desired and reported state of the cluster, as well as authorizing, processing, and validating any updates to the state of the cluster. The API server exposes endpoints for observing the state of the cluster and reacting to those changes. The other component of the control plane is the *scheduler*. It is tasked with distributing work among the nodes of the cluster given different constraints such as reserved memory, reserved CPU time, and other factors such as CPU architecture or node role.

Controllers Building on top of the API server are *controllers*. By default, Kubernetes comes with its own core controllers which provide different kinds of functionalities, for example scheduling programs to run at specific times or running programs on one, several, or all nodes in the cluster. In essence, a Kubernetes controller is a control loop which listens to changes in cluster resources and updates the state the cluster towards a desired state in a deterministic way.

Resources The API server in Kubernetes provides a CRUD (Create, Read, Update, Delete) API interface through which users can interact with various *resources*. A *resource* in Kubernetes represents a specific object or entity, such as pods, services, deployments, and more. These resources are defined using YAML or JSON manifests, which specify the desired state of the resource.

Kubernetes maintains a "desired state" model, where users specify the desired configuration of resources, and the system works to ensure that the actual state of the cluster converges towards this desired state. The controllers mentioned earlier play a pivotal role in achieving this convergence. They continuously monitor the state of resources, detect any deviations from the desired state, and take corrective actions to align the system with the intended configuration.

Furthermore, Kubernetes employs a unique identification scheme for each resource, known as the *Resource Name*. This name, along with other metadata, helps in distinguishing resources, managing access control, and enabling robust orchestration.

In summary, Kubernetes' API server and the associated CRUD operations provide a powerful mechanism for users to define, manage, and control various resources within

the cluster. This design philosophy of declarative configuration, coupled with dynamic controllers, forms the foundation for Kubernetes' ability to automate the deployment, scaling, and maintenance of containerized applications.

2.2.2 Knative

Knative (24) is an open-source platform built on top of Kubernetes that enables developers to build, deploy, and manage serverless workloads. It provides a set of components and abstractions that simplify the process of deploying and scaling containerized applications without the need to manage the underlying infrastructure.

Knative was first introduced by Google in July 2018 (25) as an extension of Kubernetes, aiming to bring the benefits of serverless computing to containerized workloads. It was developed as a collaborative effort between Google, Pivotal, and other industry leaders as part of the Cloud Native Computing Foundation (CNCF). Knative has gained significant adoption and has a vibrant ecosystem of contributors and users. Its open-source nature and compatibility with Kubernetes have made it a popular choice for developers and organizations looking to leverage the benefits of serverless computing within their existing infrastructure.

One of the key goals of Knative is to abstract away the complexities of deploying and managing serverless applications, allowing developers to focus on writing code and rather on how to deploy it. Knative achieves this by providing higher-level abstractions such as *functions*, *serving*, and *eventing*.

Knative Functions The *functions* component enables developers to easily build and deploy serverless applications in the Knative environment without requiring in-depth knowledge of Knative, Kubernetes, containers, or Docker.

By offering a collection of code templates compatible with multiple programming languages, it presents a Function-as-a-Service (FaaS) interface comparable to other serverless platforms. The entire process of creating, deploying, and executing a serverless application can be accomplished using the command line interface (CLI) program that is offered by Knative Functions.

Knative Serving The *serving* component of Knative allows developers to deploy and manage serverless applications with ease. It provides automatic scaling and handles traffic routing, making it simple to deploy new versions of an application and perform canary releases or A/B testing.

At its core, Knative Serving introduces the concept of a "service" as the basic unit of deployment. A service in Knative Serving represents an application or microservice that can be deployed, scaled, and managed independently. It allows developers to define the desired state of their application, such as the number of instances, network configuration, and resource requirements, using declarative YAML files or command-line tools.

One of the primary benefits of Knative Serving is its ability to automatically scale applications based on incoming traffic and load. It utilizes Kubernetes' Horizontal Pod Autoscaler (HPA) to dynamically adjust the number of service instances based on the configured scaling policies and observed metrics such as CPU utilization. Later revisions of Knative introduced Knative Pod Autoscaler (KPA) which allowed using request rate and concurrency as a metric for driving the service instance scaling algorithm. This ensures that the application can handle varying levels of traffic efficiently while minimizing resource waste during periods of low demand.

Knative Serving also provides advanced traffic management capabilities, allowing developers to perform canary releases, blue-green deployments, and A/B testing with ease. By leveraging Knative's traffic splitting feature, developers can direct a percentage of traffic to different versions or revisions of their service, enabling gradual rollouts and the ability to compare performance and functionality between different versions.

Knative Eventing The *eventing* component of Knative enables event-driven architectures by providing a set of primitives for managing and processing events. It allows developers to define event sources, create event consumers, and wire them together using eventing channels and subscriptions. This allows for seamless integration with various event sources and provides a scalable and flexible foundation for building event-driven systems.

Event-driven architectures have gained significant popularity due to their ability to decouple components and systems, enabling efficient communication and scalability. Knative Eventing aims to bring the power of event-driven systems to Kubernetes-based environments, making it easier for developers to build and manage event-driven applications.

At the heart of Knative Eventing is the concept of *event sources*. An event source is responsible for emitting events into the system. It can be any external system or internal component capable of generating events, such as a message queue, a database, or even a custom application. Knative Eventing provides a pluggable architecture that allows developers to integrate various event sources seamlessly.

On the receiving end, we have *event sinks*, which determine the destinations for events. While serverless applications deployed through Knative Serving are commonly associated

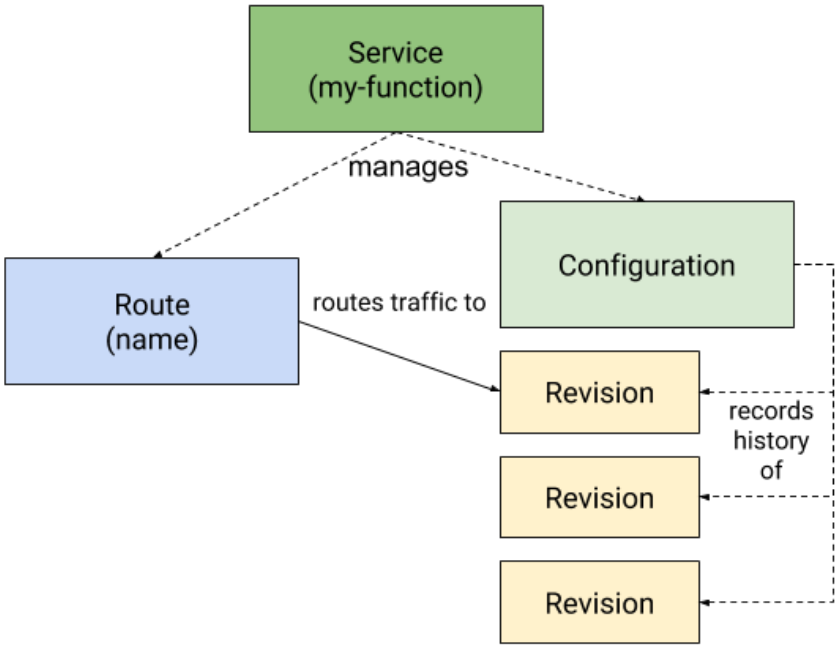


Figure 2.1: Knative Serving CRDs. A *Service* automatically manages the whole lifecycle of a serverless application. Changes to the service's traffic routes are reflected in the *Route* CRD, while changes to any metadata pertaining to the application is reflected in the *Configuration* CRD. *Revisions* are immutable copies of the Configuration, which ensures that changes to the application are atomic and do not influence running instances of the application.
Source: `https://knative.dev/docs/concepts/`

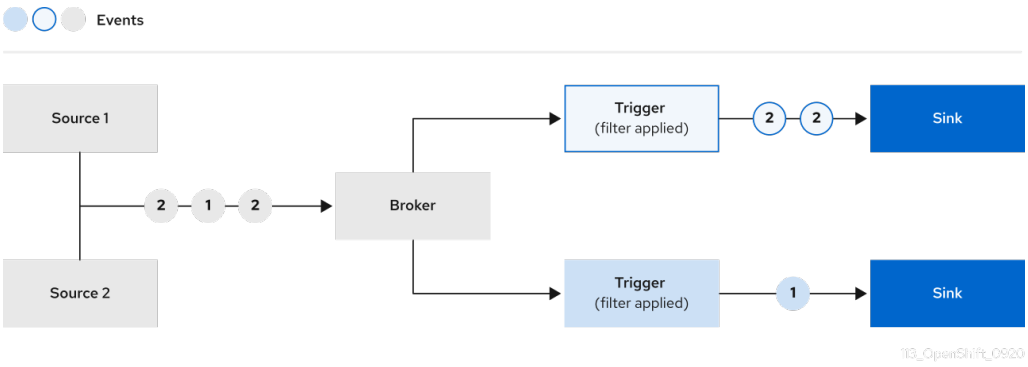


Figure 2.2: Knative Eventing allows the use of messaging brokers such that the event routing, persistence, ordering, etc. can be guaranteed. Triggers can be connected to brokers in order to subscribe to certain event topics and to apply filters before events are published to event sinks.
Source: `https://knative.dev/docs/concepts/eventing-resources/brokers/`

with sinks, they are not limited solely to this scenario. Other implementations of sinks encompass Apache Kafka (26) topics and Redis Streams (27).

Event routing can be accomplished through two primary mechanisms: *channels* and *brokers*. Channels establish direct connections to sinks, while brokers offer enhanced delivery guarantees alongside routing capabilities, being a more sophisticated event routing solution.

Knative Eventing supports different types of channels, such as in-memory channels and message broker-based channels. *Subscriptions* represent the binding between an event source and a specific channel. They define the filtering criteria for events and provide a mechanism to connect event sources to channels based on their compatibility. Subscriptions ensure that only relevant events are delivered to the intended consumers, reducing unnecessary overhead and improving scalability.

Brokers in Knative Eventing act as central event hubs, responsible for managing and routing events between event sources and sinks. They offer valuable additional capabilities, including event persistence and event aggregation, which facilitate more sophisticated event processing and integration patterns. Knative Eventing supports various broker implementations, ranging from popular open-source brokers like Apache Kafka (26) and RabbitMQ (28) to its own built-in broker, which is implemented using channels. *Triggers* are used in order to bind sinks to brokers and they contain additional features such a event filtering, dead-lettering, and retry mechanisms.

3

Overview

Our vision for a serverless platform which enables a Edge-Cloud continuum involves a platform which abstracts not just the infrastructure, but also the location where the functions are run at, and makes use of its available resources as efficiently as possible. Such a platform should be able to deploy functions in a way that allows an application developer to scale to multiple Edge locations at the same time, while also remaining cost-effective from the developer's perspective.

Our thesis proposes a pragmatic approach to achieving such a vision. We present the high-level solution design of our platform in Figure 3.

One key characteristic of our solution is the fact that each cluster of nodes, be it in the Cloud or at the Edge, run autonomous instances of the serverless platform, each maintaining a stable running state and scheduling resources based on local knowledge. This approach ensures efficient resource management and enhances overall platform performance.

In order to ensure consistency in function definitions across all locations, the source of truth is maintained centrally within the Cloud. Changes made by developers to function code or configuration are pushed to the Cloud, which then replicates these updates to all connected Edge locations. The interoperability of the Cloud and Edge serverless platforms is crucial for this process, ensuring they can communicate effectively and maintain a connection for seamless synchronization. This active connection enables the seamless replication of changes and ensures that the function definitions remain consistent across all locations.

To achieve optimal performance and cost-effectiveness, maintaining a consistent view across all locations is not enough for our serverless platform in order to facitilitate an Edge-Cloud continuum; it needs to efficiently schedule function executions across both Cloud and Edge locations. To accomplish this, we have implemented a scheduler at each

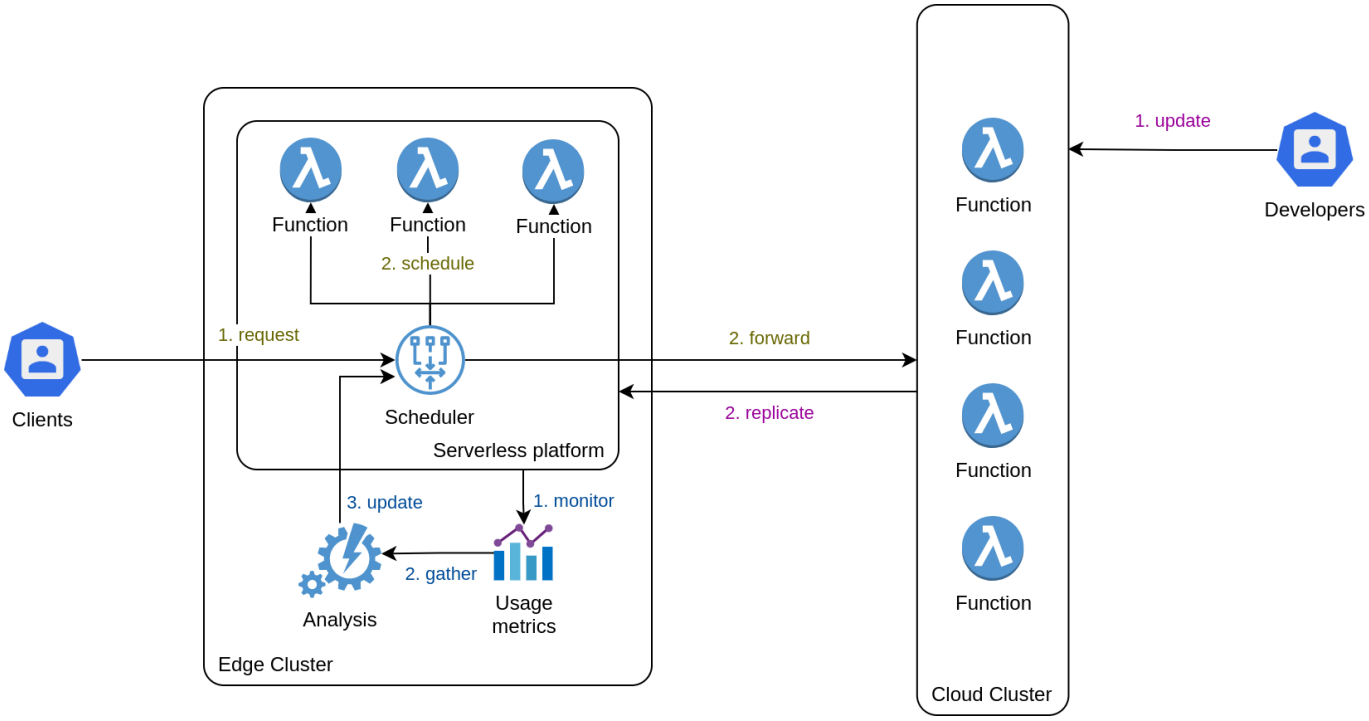


Figure 3.1: The design of the serverless platform abstracts the underlying infrastructure from both developers and clients. Functions published to the cloud platform are transparently replicated at the edge. Client requests are initiated at the edge and the scheduler determines if the request should be fulfilled by the edge or the cloud, based on usage and performance metrics collected and analyzed in the background. This enables optimized performance, cost-effectiveness, and scalability.

edge location that actively determines where requests will be served. The decisions made by the scheduler are driven by runtime metrics collected and analyzed in the background.

Our analysis methodology can range from simple heuristics based on current usage to self-improving machine learning algorithms considering both recent usage patterns and request-time metrics. This allows our scheduler to make informed decisions that optimize the platform's performance and cost-effectiveness, while also ensuring scalability. The introduction of a sophisticated scheduler is a key component in establishing our serverless platform within the Edge-Cloud continuum.

4

Design

The design of a serverless platform running on the Edge-Cloud continuum necessitates careful consideration of multiple factors, including scalability, security, cost-effectiveness, and performance. This platform must be capable of operating at both the Edge, close to the source of data, as well as in the Cloud, where it can leverage the resources of a large data center. The design should address the unique challenges presented by this hybrid environment, such as managing data transfer between the Edge and Cloud, and handling variable levels of traffic in real-time. By harnessing the advantages of both the Edge and the Cloud, a serverless platform on the Edge-Cloud continuum can provide a flexible and efficient solution for a wide range of applications.

Requirements analysis is a crucial step in the design process that involves identifying and evaluating both functional and non-functional requirements for a platform. Functional requirements encompass the capabilities and features necessary to fulfill the intended purpose of the platform, such as real-time and batch processing support, seamless integration with existing systems, and compatibility with various programming languages and frameworks. On the other hand, non-functional requirements encompass the quality attributes that are vital for the success of the platform, including performance, scalability, security, and cost-effectiveness. By conducting a thorough requirements analysis, we can ensure that the platform design aligns with its desired goals and objectives, which is crucial to the overall success of the project.

4.1 Requirement analysis

Functional requirements Functional requirements define the capabilities and features that a system must have in order to fulfill its intended purpose. They specify what the

system is supposed to do and what functions it must perform. These requirements are typically expressed in terms of inputs, outputs, and processes, and they provide a clear understanding of what the system must be able to do in order to meet the needs of its users. Functional requirements are a critical aspect of the design process, as they serve as the basis for the system's functionality and help to ensure that the final solution aligns with the desired goals and objectives. The functional requirements we propose for our Edge-Cloud serverless framework are the following:

1. The same function definition can be utilized to run serverless functions in both Edge and Cloud environments.

2. The platform can accommodate the addition of new edge clusters and nodes dynamically.

3. The Edge must have the ability to dynamically determine the execution location of a function, either at the Edge or in the Cloud.

Non-functional requirements Non-functional requirements are a crucial aspect of software design and development, and encompass the quality attributes of a system that are not directly related to its functionality. These requirements specify the performance, security, scalability, and other characteristics that a system must possess in order to be successful. Non-functional requirements are often just as important as functional requirements, as they directly impact the user experience and the ability of a system to meet the needs of its users. In our case, we define the non-functional requirements as such:

1. The system should support nodes with varying degrees of hardware capabilities (e.g. x86, ARM)

2. The system must possess the capability to dynamically allocate resources in accordance with workload demands.

3. If a worker node experiences a failure, the system can still operate normally.

4. The system should be robust to bad Edge-Cloud connection.

5. Communication between Edge and Cloud should be secure.

Platform criteria By clearly defining the functional and non-functional requirements beforehand, we can establish a set of selection criteria for the platform that will guide the design process. These criteria will serve as the foundation for our solution, ensuring that the final solution aligns with the desired goals and objectives.

1. *The serverless platform should be actively maintained.*

2. *The serverless platform should be open-source and have a permissive license.* We wish to be able to extend the platform while avoiding any restrictive licensing.

3. *The serverless platform must possess the capability to scale-to-zero and scale workloads according to actual demand*, which are essential features for its intended purpose.

4. *The serverless platform is able to run on limited resources and a varying degree of heterogeneous hardware.*

4.2 Platform evaluation

To proceed with the assessment of each serverless platform, we collect information from literature and official sources, such as GitHub statistics and documentation, and present it in a comprehensible format.

In Figure 4.1, we present the statistics of the git repositories of each platform. As each of the platform's code is hosted publicly on GitHub, this enables us to gather the same type of information, and it removes concerns such as a split community, which could be relevant if the hosting would be on different websites (e.g., GitLab, Bitbucket, etc).

In Figure 4.3, we present the capabilities of each platform as understood from the official documentation and from initial deployments of the platforms on our experimental setup. We note that not all platforms are shown here, as some (i.e., kubeless, Fn, and IronFunctions) can be filtered out using the requirements laid out previously and Figure 4.1.

To continue the requirements analysis process, we enumerate each open-source serverless platform that is available, and evaluate the reasons for including or excluding them from our platform options. This process of evaluation helps us to determine which serverless platforms are best suited to meet the functional and non-functional requirements for our

platform, and to make informed decisions about which platform to choose for our implementation. We summarize out findings in Figure 4.2 and explain our reasoning bellow for each serverless platform, on a case by case basis.

- *Kyma* (29). As a FaaS solution based on Knative, Kyma inherits the technological capabilities and constraints of this underlying framework. Our evaluation of the platform shows that it meets all of the requirements we previously defined.

- *Knative* (24) is a well-regarded serverless platform that has gained popularity both in academic research circles and within the open-source community. Our requirements analysis has determined that it, like Kyma, meets the necessary criteria, making it a suitable choice to be included as part of the platform options.

- *OpenFaaS* (30). Despite its technical capabilities, the licensing of some features in OpenFaaS presents challenges for our use case. The serverless solution, while mature, requires a paid license for crucial features such as scaling to zero and event handling using message brokers. This makes it a less desirable option for our purposes as we aim to utilize these features in our platform design.

- *Fission* (31). Our requirements analysis process has shown that Fission has the necessary capabilities and features to fulfill the intended purpose of the platform, as well as meet the non-functional requirements such as performance, scalability, security, and cost-effectiveness. As a result, we have included it in our list of viable options for the serverless platform.

- *OpenLambda* (32). Despite being well-suited for academic research, OpenLambda was found to be lacking in support for cluster deployment, which is a critical requirement for deploying our solution on both cloud and edge clusters. As a result, it was excluded from our list of potential technologies.

- *OpenWhisk* (33). Given the popularity and success of the Apache software projects, OpenWhisk is a well-established player in the serverless solutions market. Our requirements analysis indicates that it satisfies all of the previously established functional and non-functional requirements. As a result, we have decided to include OpenWhisk in our list of potential solutions for our serverless platform.

- *kubeless* (34). Despite being widely recognized for its performance, the decision was made to exclude kubeless from the design of the serverless platform. This was due to

the fact that kubeless is no longer actively maintained, and therefore may not provide the level of reliability and support required for this project. In order to ensure the success of the platform, it is important to consider the availability of support and the level of maintenance for any technology that is being considered.

- *Fn* (35). While Fn was one of the pioneers in the open-source serverless space, it is no longer maintained and therefore will not be considered in our analysis. This exclusion enables us to focus on the currently maintained serverless platforms that meet the specific requirements for our use case. By excluding unmaintained solutions, we can ensure that the serverless platform we choose is both reliable and secure, and can be supported in the long term.

- *IronFunctions* (36). Similar to Fn, IronFunctions is an early example of an open-source serverless solution. Unfortunately, like Fn and kubeless, it has since become inactive and, as a result, has been excluded from the list of platform options for our design.

After evaluating open-source serverless platforms based on the first three requirements, several options remained, namely Kyma, Knative, OpenWhisk, and Fission. To make a final decision, each platform was deployed on a cluster of physical edge devices to assess the final requirement. The ease of deployment and quality/depth of documentation were taken into account as factors for our platform choice.

After thorough evaluation, Knative was found to be the most straightforward to deploy and had the most user-friendly documentation. Although both Kyma and Fission were successfully deployed, their performance was not as optimal as that of Knative. Unfortunately, OpenWhisk could not be deployed due to memory limitations in deploying the necessary message broker.

Platform choice Having filtered the list of available open-source serverless platforms, we are left with several options, namely Kyma, Knative, and Fission. However, choosing the optimal platform among these alternatives is a intricate task, as it is influenced by various variables and is susceptible to bias. Definitively ranking these serverless solutions is unfeasible due to the multitude of variables at play. These factors encompass the specialized knowledge and experience required for platform setup, which can impact both performance and resource utilization. Moreover, the variability in the behaviour of the underlying container orchestrator, such as Kubernetes or OpenShift, can significantly influence the

placement of crucial components of the serverless platform on compute nodes, thereby affecting its overall performance.

Given the feature set we collected in Table 4.3 and the relevant statistics gathered about each platform in Table 4.1, we have chosen *Knative* as the base for our Edge-Cloud serverless platform. Our decision was motivated by several factors, including the ease of deployment, the active community, and the comprehensive documentation available. Additionally, the fact that Kyma is based on Knative makes it possible to incorporate the same modifications into Kyma. This choice provides a solid foundation for the development of a robust and flexible serverless platform that can meet the unique challenges posed by the Edge-Cloud model.

Serverless platform	GitHub Repositories	Stars	Forks	PRs (active)	PRs (closed)	Issues (open)	Issues (closed)	Commits	Last commit	Archived
Kyma	kyma-project/kyma	1368	382	92	9652	285	4544	7499	08/06/2022	No
Knative	knative/serving knative/eventing	5683	1400	21	12881	219	6246	10920	07/06/2022	No
OpenFaaS	openfaas/faas	21656	1767	3	892	31	753	2055	06/06/2022	No
Fission	fission/fission	7031	706	10	1425	183	735	1470	06/06/2022	No
OpenLambda	open-lambda/open-lambda	770	86	2	58	9	42	831	02/06/2022	No
OpenWhisk	apache/openwhisk	5700	1096	39	3231	319	1652	2961	02/06/2022	No
kubeless	vmware-archive/kubeless	6849	745	3	581	197	471	1043	15/12/2021	Yes
Fn	fnproject/fn	3010	5164	382	10	940	135	507	19/12/2019	No
IronFunctions	iron-io/functions	3010	231	7	386	86	219	904	20/08/2018	No

Table 4.1: Public statistics offered by GitHub for each of the repositories of the open-source serverless platforms as of the 8^{th} of June 2022. In cases where there are multiple relevant repositories, we show the cumulative values of all the repositories.

	Platform criteria			
Serverless Platform	#1	#2	#3	#4
Kyma	✓	✓	✓	✓
Knative	✓	✓	✓	✓
OpenFaaS	✓		✓	✓
Fission	✓	✓	✓	✓
OpenLambda		✓	✓	✓
OpenWhisk	✓	✓	✓	✓
kubeless		✓	✓	✓
Fn		✓	✓	✓
IronFunctions		✓	✓	✓

Table 4.2: For each serverless platform we investigated, we have assessed whether they fit our platform criteria which we set out.

Feature	Monitoring	Orchestrator	Cluster support	Autoscaling	Scale to zero	Warm instance pool
Kyma	Prometheus with Grafana	Kubernetes	Yes	Yes (per function; queue-based, cpu-based)	Yes	No
Knative	Prometheus with Grafana	Kubernetes	Yes	Yes (per function; queue-based, cpu-based)	Yes	No
OpenFaas	Prometheus with Grafana	Kubernetes, faasd	Yes	Yes (global; request rate)	No (only available with paid license)	No
OpenWhisk	None	Kubernetes	Yes	Yes (per function; queue-based)	Yes	No
Fission	Prometheus with Grafana	Kubernetes	Yes	Yes (per function)	Yes	Yes
OpenLambda	None	-	No	Yes (per function)	Yes	No

Table 4.3: Platform capabilities relevant to the defined requirements.

5

Implementation

Knative Edge serves as the implementation of our proposed design and functions as an extension for the open-source serverless platform, Knative (24). Developed and rigorously tested on Knative Serving 1.7 and Kubernetes 1.24, this implementation comprises three integral components that collaborate seamlessly to meet our predefined requirements: the *Edge Controller*, the *Edge Proxy*, and the *Edge Operator*.

Knative Serving augments Kubernetes with the Service custom resource, facilitating serverless functionality. A Knative Service encapsulates the serverless function's definition and can establish dependencies on other cluster resources. To ensure uniformity between the Cloud and Edge environments, the Edge Controller plays a pivotal role in replicating not only the service itself but also its associated Namespace, ConfigMaps, and Secrets, in alignment with the requirements of the Knative Service. This synchronization ensures consistency and reliability across the board.

5.1 Edge Controller

The core component of Knative Edge is the *Edge Controller*. As shown in Figure 5.1, it is makes use of several different components within edge cluster in order to achieve its goals.

It runs on the Edge clusters and its responsibilities are two-fold: it maintains the consistency of the serverless functions between Cloud and the Edge and it enables offloading of requests from Edge to Cloud based on the Edge cluster utilization.

5.1.1 Cloud to Edge replication

The Knative Edge Controller mirrors Knative Services from the Cloud cluster to the Edge cluster by watching for changes in the Kubernetes resources of both clusters and modifying

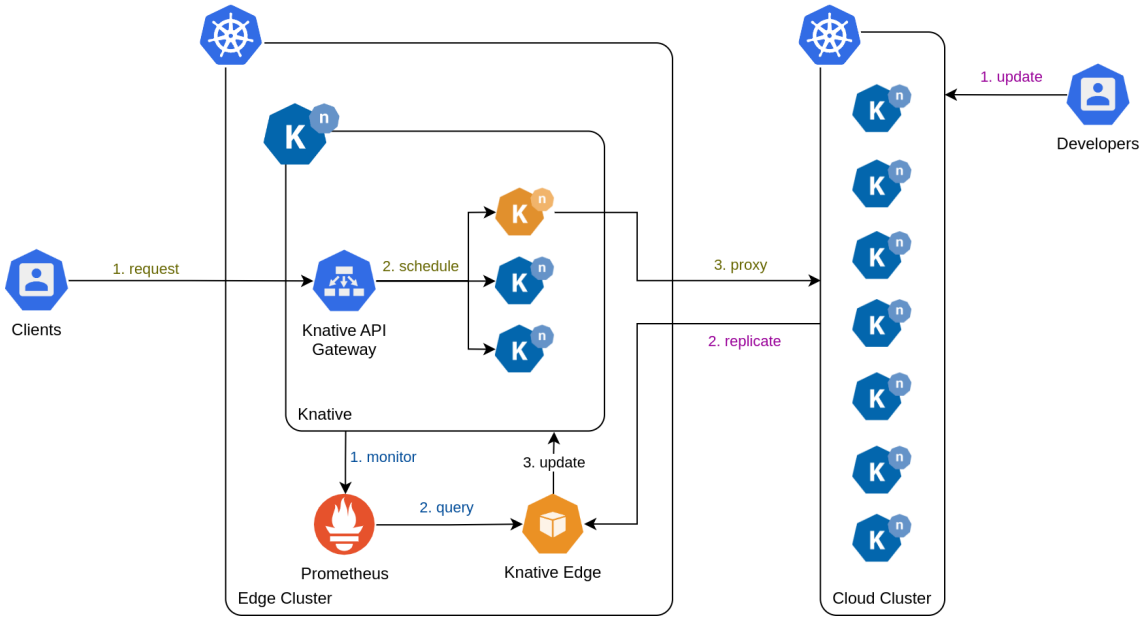


Figure 5.1: The figure shows the specific implementation of the Knative Edge Controller. It is responsible for maintaining a coherent view between the Cloud and the Edge, as well as influencing the scheduling of requests. To achieve its first purpose, it makes use of Prometheus in order to monitor and query performance metrics from Knative, which are then used in order to update how functions are scheduled at the Edge.

the Edge resource in order to keep a consistent definition of the service.

Kubernetes resources can be categorized by their scope within the cluster: they can either have cluster-wide existence or be namespaced. Knative Services are *namespaced* resources, which means they rely on the presence of a corresponding *Namespace* before they can be created. Furthermore, developers have the option to reference two other resource types: *ConfigMaps* and *Secrets*. These resources can be utilized by referencing them in the Knative service definition. For a comprehensive illustration, we refer to Figure 5.3, which presents both the YAML definition and a visual representation of a Knative Service's dependencies.

In total, the Knative Edge Controller replicates four types of Kubernetes resources:

1. Namespace
2. ConfigMap
3. Secret
4. Knative Service

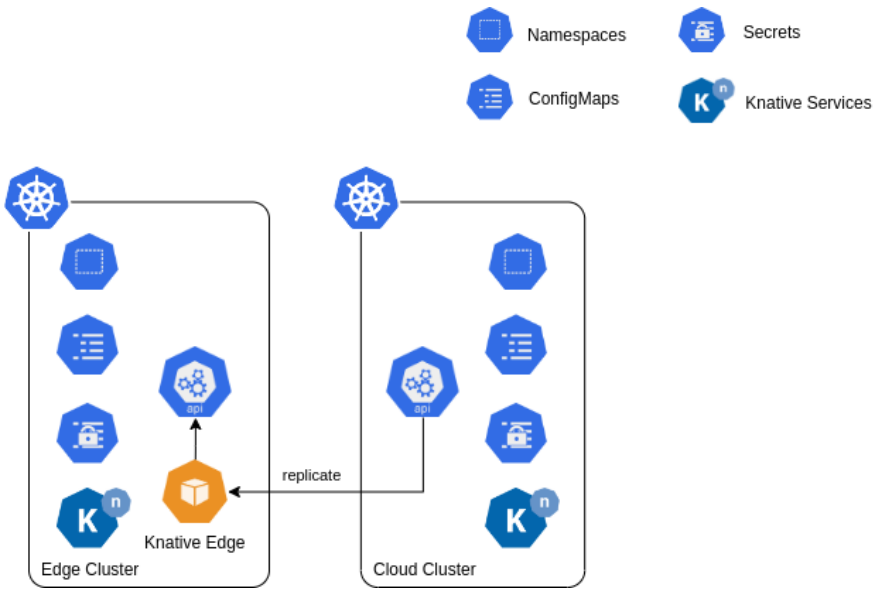


Figure 5.2: The Knative Edge Controller watches specific types of resources on the Cloud cluster - namely Namespaces, Secrets, ConfigMaps, and Knative Services -, mirrors and manages them on the Edge cluster. Furthermore, any changes to either the Cloud or the Edge resources trigger the Knative Edge Controller to reconcile any differences between the two clusters.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-function
  namespace: default
spec:
  template:
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld.go
      ports:
        - containerPort: 8080
      env:
        - name: TARGET
          value: "World"
      envFrom:
        - configMapRef:
            name: example-config
        - secretRef:
            name: example-secret
```

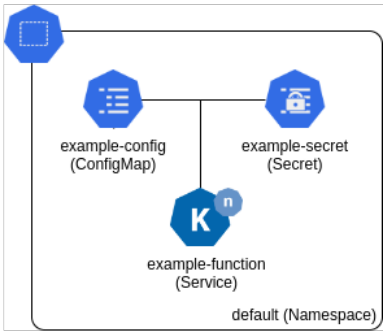


Figure 5.3: Example of a Knative Service definition and its visual representation.

When replicating a resource, the controller must take into account the Knative-related states which might reside in the definition of either the Cloud or the Edge, otherwise, it risks making unnecessary changes to the resources it replicates. This is most evident when a Knative Service is replicated, as any changes to it can trigger either Knative Serving or Knative Edge to react and make more changes to the resource. This feedback loop can cause degradation of the serverless functions running either on the Edge or Cloud, and can increase the traffic between the Cloud and Edge.

Our approach to this issue is to selectively compare fields in the Knative Service definition. Whenever the controller receives an update for services it replicates, it makes a copy of the current definition from the Edge and overwrites its fields using the known fields from the Cloud. When overwriting, we skip over the *state* of the service and any *annotations* used internally by the Knative Edge Controller. The new service definition is compared to the current definition on the Edge cluster and deployed to the cluster if any change is detected.

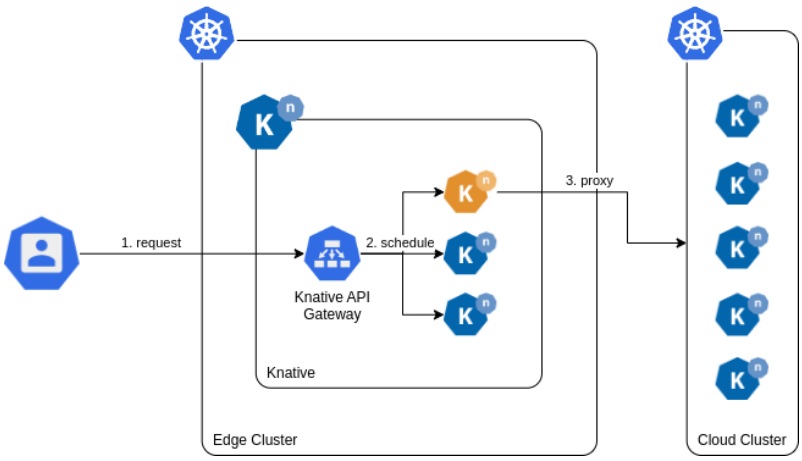
5.1.2 Edge to Cloud offloading

The second half of the Edge Controller's functionality pertains to dynamically offloading the workload from the Edge cluster when it is being strained. Figure 5.4 (b) shows in detail each component that makes up the functionality described.

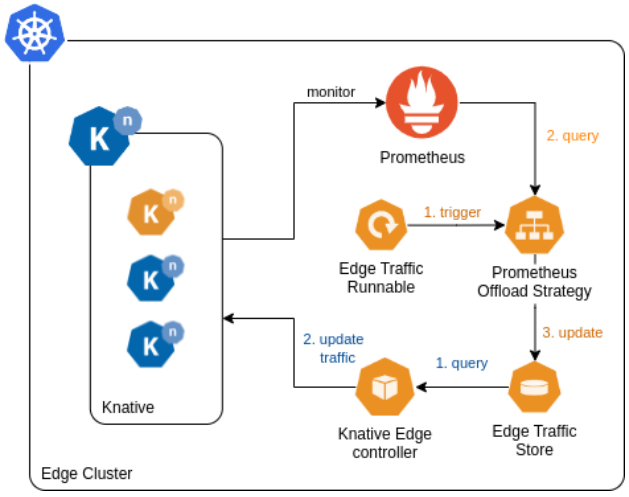
As shown in the solution design from Figure 3, the offloading feature has three components: the usage metrics, analysis, and scheduler.

Usage metrics. For efficiently monitoring and managing runtime platform metrics in our Edge cluster, we utilize an instance of Prometheus deployed on each Edge cluster. This instance is configured to be aware of all Knative components in the cluster, including all function instances that are running. It scrapes these metrics regularly and stores them temporarily in a time series database, which is optimized for both storing and querying metrics such as those generated by Knative. Since only recent data is being queried by Knative Edge, we configure a short data liveness to reduce as much as possible the overhead of Prometheus.

Scheduler. One of the integral components within the Knative framework is its API Gateway. This component serves as the primary entry point for all function calls within the platform. Its crucial role lies in efficiently routing incoming traffic to the corresponding running instances of functions, effectively managing the distribution of workloads.



(a) The entrypoint for each request is the Knative API Gateway, which is tasked with distributing the load between all of the running serverless instances. Knative allows us to deploy multiple versions of the same function, which we use to our advantage in order to deploy a special version of the function which routes requests from the Edge to the Cloud.



(b) In the background, we periodically gather usage metrics from the Edge cluster and use those metrics in order to calculate the traffic split between the normal Edge function and the special Edge proxy function. The results are stored in a separate in-memory store, further used by the Knative Edge controller to update the traffic split percentage inside the Knative API Gateway.

Figure 5.4: An overview of the offloading functionality as implemented by Knative Edge. On the left side (a), we show how requests are being split between the Edge and the Cloud, while the right side (b) shows the process behind picking the split percentage based on usage patterns.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-function
  namespace: default
  labels:
    edge.jevv.dev/edge-offload: "true"
    edge.jevv.dev/environment: "default"
spec:
  template:
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld-go
        ports:
          - containerPort: 8080
        env:
          - name: TARGET
            value: "World"
```

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-function
  namespace: default
  annotations:
    edge.jevv.dev/edge-proxy-traffic: "0"
    edge.jevv.dev/last-observed-generation: "0"
    edge.jevv.dev/last-observed-remote-generation: "0"
    edge.jevv.dev/remote-host: helloworld-function.default.example.com
    edge.jevv.dev/remote-url: http://example.com
  labels:
    edge.jevv.dev/created-by: "knative-edge-controller"
    edge.jevv.dev/edge-offload: "true"
    edge.jevv.dev/environment: "default"
    edge.jevv.dev/managed: "true"
    edge.jevv.dev/managed-by: "knative-edge"
spec:
  template:
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld-go
        ports:
          - containerPort: 8080
        env:
          - name: TARGET
            value: "World"
  traffic:
  - latestRevision: true
    percent: 100
  - revisionName: helloworld-function-edge-proxy
    tag: edge-proxy-00001
    percent: 0
```

Figure 5.5: Example of a function definition after it has been replicated from Cloud to Edge. On the left side, we have the initial definition which was deployed to the Cloud and on the right side is the definition that was replicated to the Edge. Knative Edge uses both annotations and labels in order to keep track of the function during its lifetime, so we have too keep this into account when synchronizing resources. Additionally, for the given function, we can see that the offloading functionality is enabled, as set in the labels, and is currently directing all traffic to the Edge cluster.

Knative boasts a range of features, among which is its capability to dynamically route traffic between various revisions of a specific function. In essence, Knative retains a comprehensive history of alterations made to a function, empowering developers to direct requests to specific versions that were configured at different points in time. Leveraging this advanced routing functionality, we introduce a distinct version of the function designed explicitly for request forwarding. This forwarding version of the serverless function takes on the responsibility of directing requests towards the identical function that operates within the Cloud environment. A YAML representation of this concept is provided in Figure 5.5, where the Edge function version facilitates traffic routing across multiple function versions, one of which is the Knative Edge Proxy.

The scheduling is driven by a load balancing algorithm which spreads the traffic to the different routes based on a defined percentage, as seen in Figure 5.5 in the traffic section

of the Knative service specification. In order to control the traffic, we implement several components that collaborate to determine the optimal value at any given moment, thereby controlling the traffic. These components work in conjunction with one another to decide the optimal value at any given time.

Analysis. Edge Traffic Runnable is a background running thread in the Edge Controller which periodically triggers the offloading analysis. It retrieves all of the functions which are deployed on the Edge cluster and it analysis recent usage metrics for each function. Since the offloading strategy is decoupled from the runnable, it is possible to replace the default strategy implemented in Knative Edge with better performing strategies.

We implement a simple default offloading strategy in Knative Edge which uses the request latency metrics of all of the functions running at the Edge. Specifically, it queries the values for $r_{latency}$ from Equation 5.1 for each function and calculates $r_{traffic}$, which is the intended traffic percentage that should be forwarded to the Cloud. Through different iterations, we found that directly using $r_{traffic}$ for setting the traffic percentage lead to unstable offloading. As such, we define $R_{traffic}$ in Equation 5.4 as a more stable way of updating the traffic percentage.

The equations used in the default offloading strategy are the following:

$$r_{latency}(t) = \frac{p_{95}(dist_{latency}(t))}{p_{50}(dist_{latency}(t))} \tag{5.1}$$

$$r'_{latency}(t) = \frac{\sum_{k=0}^{c_t} {c_{decay}}^k \times r_{latency}(t-k)}{\sum_{k=0}^{c_t} {c_d}^k} \tag{5.2}$$

$$r_{traffic}(t) = \begin{cases} 0, \text{ if } r'_{latency}(t) < c_{soft} \\ 100, \text{ if } r'_{latency}(t) > c_{hard} \\ 100 \times \frac{r'_{latency}(t) - c_{soft}}{c_{hard} - c_{soft}}, \text{ otherwise} \end{cases} \tag{5.3}$$

$$R_{traffic}(t) = R_{traffic}(t-1) \times c_{inertia} + r_{traffic}(t) \times (1 - c_{inertia}) \tag{5.4}$$
$$R_{traffic}(0) = 0$$

where

- $dist_{latency}(t)$ is the distribution of request latencies at time t and p_{50} and p_{95} are the 50^{th} and 95^{th} percentile respectively. n^{th} percentile is defined as the minimum value in a distribution for which at least $n\%$ of the values are smaller than it.

- c_t defines how many steps in time are used to calculate the weighed sum of the latest request latency ratios measured.
- $r_{latency}$ is the weighted sum of the latest latency response ratio. Colloquially, $r_{latency}$ measures the spread of the latency distributions over time, giving more importance to recent values than to older ones. This is implemented by using an exponentially decreasing weighted sum with c_{decay} as exponent.
- c_{soft} and c_{hard} are the soft limit and hard limit of the percentile ratio. For any $r'_{latency}$ that falls bellow c_{soft}, the traffic percentage is set to 0. For values above c_{hard}, the traffic percentage is set to 100. And finally, for values between, we interpolate between 0 and 100, depending on where the values lie in respect to c_{soft} and c_{hard}.
- $c_{inertia}$ is the inertia factor, which measures how much $r_{traffic}$ influences $R_{traffic}$.

5.2 Edge Operator

The Edge Operator is a component of Knative Edge that is responsible for managing the Edge Controller. It does this through the use of Custom Resource Definitions (CRDs),

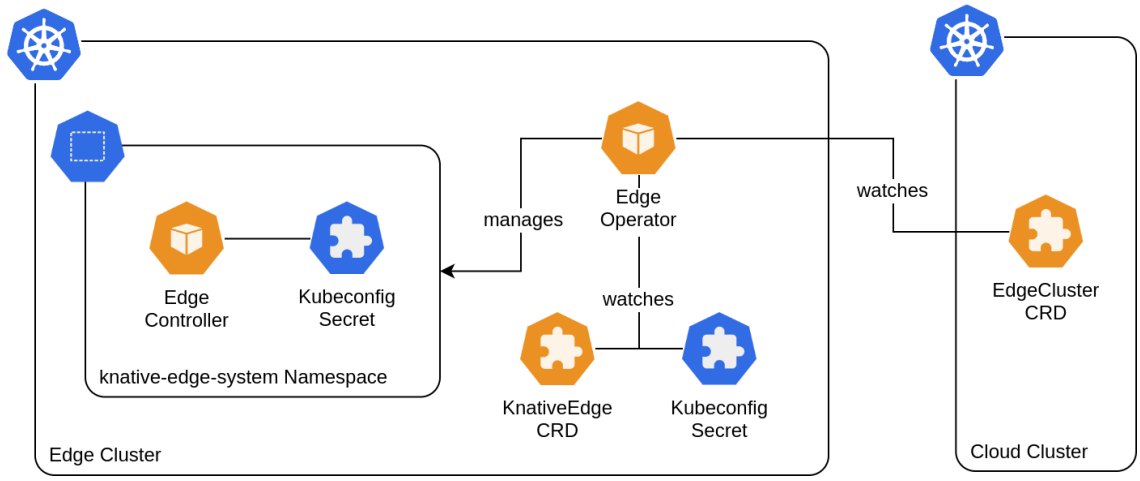


Figure 5.6: The role of the Knative Edge Operator is to oversee the deployment of the Knative Edge Controller by utilizing Custom Resource Definitions (CRDs) from both the local and remote clusters. The EdgeCluster CRD located on the cloud cluster contains information such as the name, zone, region, and allowed environments of the edge cluster. On the edge cluster, there is a corresponding KnativeEdge CRD that holds the configuration data for the Edge Controller. Lastly, the KnativeEdge CRD references a Secret holding both the credentials and certificates for the cloud Kubernetes API, providing a secure connection between the clusters.

specifically the KnativeEdge CRD. This CRD is used to link the edge cluster with the cloud cluster, providing a bridge between the two environments.

The KnativeEdge CRD references a secret that contains the credentials for the cloud cluster. This allows for secure communication between the Edge and Cloud, ensuring that sensitive information is protected. The Edge Operator is responsible for managing this secret, making it easy to update the credentials as needed.

In addition to linking the Edge and Cloud clusters and providing secure communications, the Edge Operator also provides a centralized point of control for Knative Edge. This makes it easier to manage and update the Edge Controller as needed, ensuring that the environment remains secure and stable. The Edge Operator also provides a way to configure and customize the Edge Controller, including the ability to set the execution period and lookback time for the offloading functionality.

5.3 Deployment

To run an instance of Knative Edge, one must start by deploying Knative Serving on both the edge and cloud clusters. This step starts the base serverless platform, which serves as the foundation for our solution. Next, the necessary YAML files for the Cloud and Edge clusters should be downloaded and deployed on each respective cluster to complete the setup process.

Cloud cluster The Knative Edge build script creates two YAML files, one for each location. The cloud YAML file only includes the definition of the EdgeCluster CRD. Figure 5.7 illustrates the schema of the edge cluster, which comprises of details such as the designated environments, cluster name, region, and zone.

Once the YAML file is deployed in the cloud cluster, an EdgeCluster instance can be created with the relevant information to establish a connection between the edge and cloud clusters. The next step is to obtain the cloud cluster credentials, which follow the format shown in Figure 5.8 that is used by other well-known Kubernetes tools, such as `kubectl`. For security purposes, it is recommended to create a new service account with limited access for each edge cluster that is connected. To accomplish this, a cluster role named `knative-edge-edgecluster-viewer-role` is included in the deployment YAML file, which can be utilized to grant access to EdgeCluster resources in the cloud.

```yaml
apiVersion: edge.jevv.dev/v1alpha1
kind: EdgeCluster
metadata:
  name: example-cluster
spec:
  zone: zoneA
  region: regionA
  environments:
    - envA
    - envB
```

Figure 5.7: The resource definition of the edge cluster. It is a cluster-wide resource which lives in the cloud cluster and is used by the Knative Edge operator in order to retrieve operational information, such as the environments, cluster name, zone, and region.

```yaml
apiVersion: v1
kind: Config
clusters:
  - name: example-cluster-cloud
    cluster:
      certificate-authority-data: *****
      server: https://example-cluster.cloud:6443
contexts:
  - name: edge-cluster-user@example-cluster.cloud
    context:
      cluster: example-cluster-cloud
      namespace: default
      user: edge-cluster-user
users:
  - name: edge-cluster-user
    user:
      token: *****
current-context: edge-cluster-user@example-cluster.cloud
```

Figure 5.8: Example of credentials for the cloud Kubernetes cluster. In order to enable the connection between the cloud and edge clusters, such a credential must be generated for the cloud Kubernetes cluster and saved on the edge cluster as a ConfigMap and referenced in the KnativeEdge definition.

The deployment steps on the cloud cluster can be summarized as follows:

1. Deploy Knative Serving.
2. Deploy the cloud version of the Knative Edge YAML.
3. Create a new resource of type EdgeCluster with the appropriate information and access control of the edge cluster.
4. (Optional) Create a service account with appropriate access.
5. Create a credentials YAML file to be used on the edge cluster.

Edge cluster The deployment procedure for the edge cluster is comparable to that of the cloud cluster. Upon applying the edge version of the Knative Edge YAML deployment file to the cluster, the Edge Operator will initiate and then await the completion of the setup process.

To finalize the setup process, a KnativeEdge resource and its accompanying credentials Secret must be created. The credentials must be retrieved from the cloud cluster and used to create a Secret with the key `kubeconfig` and the YAML credentials as its content. The next step is to create a KnativeEdge resource within the same Namespace as the previously created Secret. The appropriate information, such as the name of the associated EdgeCluster located in the cloud, must be entered into the resource. An example of a KnativeEdge definition is illustrated in Figured 5.9. Once the resourece has been deployed, the Edge Operator will then initiate an attempt to connect to the cloud cluster using the provided credentials. Upon successful connection, the Edge Controller will be deployed automatically and will commence replicating resources from the cloud cluster.

To summarize the deployment process thus far, the following steps can be defined:

1. Deploy Knative Serving.
2. Deploy the edge version of the Knative Edge YAML.
3. Create a new Secret using the crendetials retrieved from the cloud cluster previously.
4. In the same Namespace as the Secret, create a KnativeEdge resource and use the same cluster name as the corresponding EdgeCluster in the cloud.
5. Wait for the KnativeEdge resource to be updated with the outcome of the deployment.

```yaml
apiVersion: operator.edge.jevv.dev/v1alpha1
kind: KnativeEdge
metadata:
  name: knative-edge
  namespace: knative-edge
spec:
  clusterName: example-cluster
  secretRef:
    name: knative-edge-kubeconfig
    namespace: knative-edge
  observability:
    metrics: otel-collector.metrics:55678
```

Figure 5.9: The resource definition of Knative Edge. It is a resource which lives in the edge cluster and its role is to provide the configuration from which the Knative Edge controller will be created. This include the cluster name, which should be the same as the definition in Figure 5.7, the corresponding cluster credentials, and the metrics server associated with Knative.

In addition to providing a central point of control for Knative Edge, the Edge Operator can also be used to troubleshoot issues with the Knative Edge deployments on edge clusters. It does this by appending diagnostic events to the relevant KnativeEdge resource within the Kubernetes cluster. The events can be seen by using `kubectl describe` on the KnativeEdge resource, and they include diagnostic events such as whether the cloud cluster is available, whether credentials are missing or invalid, or if the deployment of Knative Edge has been successful.

6

Evaluation

In order to evaluate the performance of a system under different loads, a series of experiments were conducted to measure the requests latencies, CPU, memory, and network usage. The experiments involved making a series of requests to the system and collecting data on the response time, utilization of system resources, and network traffic. The following section provides a detailed description of the experimental setup, procedures, and results obtained from these experiments.

The experiments were performed on a system consisting of an edge cluster of 4 Raspberry Pi 3B+ and a low-power x64 edge device, as well as a cloud virtual machine (VM). The requests were made using a purpose-built experiment runner, which generates requests at a specified rate. The performance metrics measured included average response time, maximum response time, CPU utilization, memory utilization, and network bandwidth. The experiments involved making requests for a total of 15 minutes for each experiment, with a low request rate for the first 5 minutes, followed by a linear increase in the request rate to the upper bound for the next 5 minutes, and a maintained high request rate for the final 5 minutes. It should be noted that the experimental setup assumes a stable network connection and does not account for any other external factors that may affect system performance.

Experimental setup. The purpose of the experiments is to conduct a performance analysis of our solution, Knative Edge. To this end, the experiments were performed on two separate Kubernetes clusters, one deployed entirely within a cloud virtual machine and the other deployed on edge devices. The cloud virtual machine was configured with the default configuration of Knative, while the edge cluster required further configuration optimizations in order to provide a consistent performance.

Type	CPU	Memory	Storage
Cloud Virtual Machine	4 vCPUs	16GB	160GB NVMe
Edge device	2 x64 cores @ 1.6GHz	4GB	64GB Flash
Raspberry Pi 3 B+	4 ARMv8 cores @ 1.4GHz	1GB	32GB Flash

Table 6.1: The experiment includes several devices, each with specific hardware specifications. The edge cluster is comprised of an edge device and four Raspberry Pis, while the cloud cluster consists of a single virtual machine (VM).

First of all, the control plane of the edge Kubernetes cluster was entirely hosted on an x64 device with 4GB of memory. This decision was made due to the memory requirements of the control plane, which exceeded the available memory on the Raspberry Pi devices with only 1GB of memory. Through a series of trials and evaluations, we determined that while it was technically feasible to run the control plane on a Raspberry Pi, the software was prone to errors and the devices running the control plane could not accommodate the memory requirements of running Knative components in addition to Kubernetes components. As a result, it was necessary to host the control plane on an edge device with sufficient memory resources.

The second modification made to the experimental setup was to host the Knative components on the x64 edge device and use the Raspberry Pi devices as worker nodes. This decision was made in order to ensure consistent performance across the cluster. By default, Kubernetes attempts to distribute running containers evenly across all nodes in a cluster. However, when nodes are heterogeneous in nature, as was the case in our experiment, this can result in inconsistent results across multiple runs of the same experiment. Hosting the Knative components on a single, homogeneous node mitigated this issue and ensured consistent performance.

The following section contains comprehensive technical specifications, including detailed information about the hardware (6.1) and software (6.2, 6.5) configurations used in the experiments, as well as any relevant settings and parameters (6.3). Additionally, the network topology for our experimental setup is presented in Figure 6.1 (a). This information is important for understanding the conditions under which the experiments were performed, and for ensuring that the results can be accurately interpreted.

It is essential that the technical specifications, software versions, and network setup are thoroughly documented in order to ensure the replicability of the experiments. By providing this information, it is possible for others to reproduce the experiments and validate the results, which is a fundamental aspect of scientific research.

Type	OS	Kubernetes	Knative
Low-powered PC	Alpine Linux v3.15.6	1.24.3	1.7
Raspberry Pi 3 B+	Alpine Linux v3.15.6	1.24.3	1.7
Virtual machine	Ubuntu 22.04.1 LTS	1.24.3	1.7

Table 6.2: The software versions utilized in the experiments were kept consistent across all devices. The edge devices were equipped with a lightweight Linux distribution, Alpine Linux, while the cloud virtual machine was utilizing an Ubuntu image. The version compatibility of both Kubernetes and Knative was maintained uniformly across the cloud and edge devices.

Constant	Value	Source
c_{soft}	1.25	Equation (5.3)
c_{hard}	2.0	Equation (5.3)
$c_{inertia}$	0.6667	Equation (5.4)
c_{decay}	0.5	Equation (5.2)

Table 6.3: In the implementation of the offloading strategy, a set of values for the constants were selected based on what was deemed to be sensible. While it is feasible to set the constants individually for each function in Knative Edge, it was determined that the chosen values were sufficient for the needs of the implementation without undergoing further optimization specifically for the use case.

Workload	Minimum load	Maximum load	Total requests
Matrix Multiplication	1 req/s	5 req/s	2700
Image Processing	1 req/s	8 req/s	4050
Random I/O	1 req/s	20 req/s	9450
Mixed	1+1+1 req/s	4+8+8 req/s	10350

Table 6.4: For the first 5 minutes of an experiment run, we make requests at the minimum load. Afterwards, we linearly increase the number of requests made until we read the maximum load at 10 minutes. Finally, we continue making requests at that rate for another 5 minutes. For the mixed workload, we show the minimum and maximum load as the sum of the previous workloads, in the order they are in the rows above.

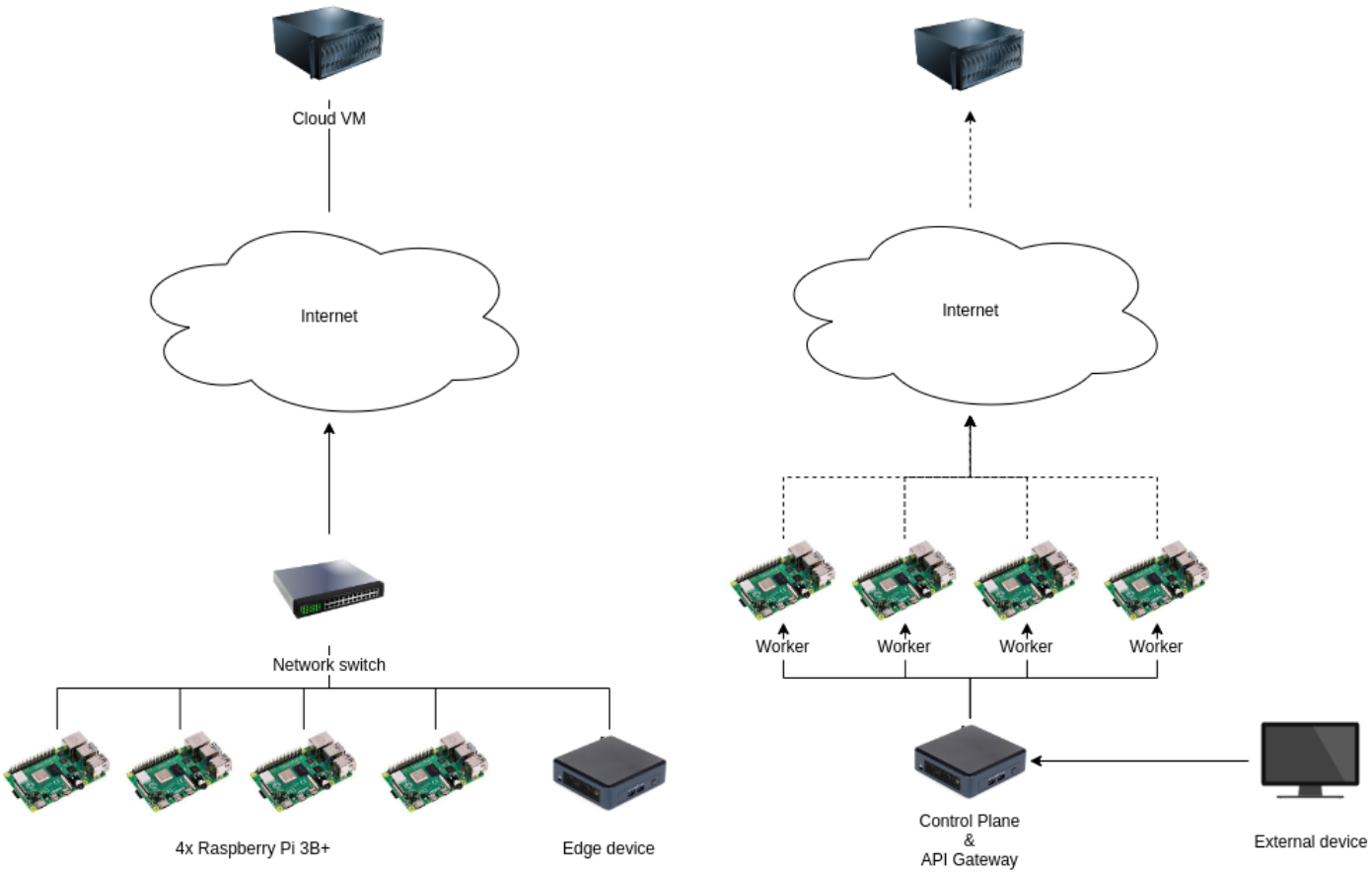
Option	Value	Note
Experiment concurrency limit	8	A limit was set in order to allows Knative to scale the functions before overloading a single edge node.
Experiment concurrency target	4	The intended level of concurrency was established at half the maximum concurrency limit to facilitate the scaling down of the function.
Experiment max scale	8	If not set, Knative will scale function instances based on the concurrency target, even if nodes are being overloaded. To prevent cluster failures, it is necessary to set a limit.
Edge proxy min scale	3	A lower limit was set over 0 to ensure consistent experiment results, as allowing the Edge proxy to scale to zero leads to inconsistent results.
Edge proxy max scale	5	A limit was set to prevent the creation of excessive Edge proxy instances, as the container is I/O bound and having more instances than worker nodes can be inefficient.

Table 6.5: In order to optimize the functions for execution on edge devices and to achieve consistent experimental results, modifications were made to the default run-time options provided by Knative.

Data collection. In order to gather host-level metrics such as CPU and memory utilization, we employed Prometheus, an industry-standard solution for monitoring and alerting. To facilitate data collection, a lightweight metric exporter was deployed on each device in the experimental setup. The Prometheus instance, which was positioned external to the experimental environment, was configured to gather host-level metrics from all nodes at 15-second intervals. The metrics were saved for subsequent querying and analysis.

In addition to host-level metrics, we gather request-level metrics, including metrics such as response time and the location of request execution (Edge or Cloud). This is achieved by exposing additional HTTP headers in the responses provided by the Edge Proxy and deployed functions. To minimize the potential overhead introduced by the metric gathering solution, careful consideration was given to the choice of solution, and it was selected with the goal of minimizing running overhead and not affecting the final results. To ensure consistency in the results, a documented experimental methodology was employed and each experiment was performed multiple times to validate its consistency.

Experiment specification. The experimental specification section of the thesis presents the methodology employed to evaluate the performance of the serverless platform. A



(a) Our experimental setup consists of a single VM as the cloud cluster and, as the edge cluster, we have one low-powered x64 edge device and 4 Raspberry Pis The edge cluster is on the same network and the nodes are connected by a single network switch, providing a stable and low latency environment. Although the network switch is capable of speeds of up to 1Gb/s per port, the edge devices are limited to 100Mb/s.

(b) In the experiments, requests are initiated from an external devices and are received by the API Gateway running on the control plane node. The requests are then routed to the worker nodes, which in turn might forward the request to the cloud cluster.

Figure 6.1: On the left, we have the network configuration of our experimental setup, while the right side shows how requests move across the devices in the experiment.

custom-made experiments runner was devised to generate requests at a specified rate, with the ability to restrict the total number of concurrent requests in order to prevent potential cluster failures. The arrival time between each request was calculated using a Poisson distribution to create conditions that closely resemble real-world scenarios. Furthermore, each request was designed to be independent of one another, with the exception being when the concurrency limit is reached.

The experiments consist of three phases, each of which serves to evaluate the platform's behavior under different request generation conditions. The first phase of the experiment involves generating a low number of requests for a period of 5 minutes, while the second phase involves gradually increasing the request generation rate linearly over a period of 5 minutes, until it reaches a pre-defined upper limit. The final phase of the experiment involves generating requests at the set upper limit for a period of 5 minutes.

The variables in the experiment template include the *workload type* and the *edge scheduling strategy*. The edge scheduling strategy determines the distribution of traffic between the Edge and the Cloud, and can is set to various levels of distribution in our experiments (0%, 25%, 50%, 75%, 100%, and auto). This enables the evaluation of the impact of different workload types and edge scheduling strategies on the performance of the platform. The results of these experiments provide valuable insights into the behavior and performance of the serverless platform operating within the Edge-Cloud continuum, and contribute to the understanding of the trade-offs involved in operating this type of platform.

Experiment workloads. For our experiments, we define a set of four workloads that we perform on our cluster. We devise the workloads in such a way as to measure the performance of Knative Edge over various types of tasks.

In order to thoroughly evaluate the performance of the system, multiple workloads were utilized in the experiments. The use of multiple workloads allows for a comprehensive analysis of the system's behavior under different types of loads and usage scenarios. The following are the workloads used in the experiments:

- Matrix Multiplication: A function where the function took as input two matrices of size 200x200 in JSON format and performed matrix multiplication three times before returning the result.

- Random I/O: A function where a 16MB temporary file was created, and random data was written to and read from the file.

Workload	CPU	Memory	Disk	Network
Matrix Multiplication	high	medium	none	medium
Image Processing	medium	medium	low	low
Random I/O	low	low	high	none
Mixed	high	medium	high	medium

Table 6.6: For each category of experiment workload, we provide an expected impact analysis on the usage of CPU, memory, disk, and network resources.

- Image Processing: A function where the function took as input a JPEG image of dimensions 1914x1276 and size of 1.13MB and returned the same image resized to a width of 256 whilst keeping the image dimensions ration consistent.
- Mixed: An experiment where all of the previous functions were combined into a single experiment, providing an evaluation of the system's performance under a more complex and demanding workload scenario

For each of the workloads aforementioned, we present our expected impact it would have in regards to CPU, memory, disk, and network usage in Table 6.6.

6.1 Results

The results section presents a comprehensive analysis of the performance of Knative Edge, as evaluated through the experimental methodology outlined in the previous section. The results provide insights into the behavior and performance of the platform under different request generation conditions and edge scheduling strategies.

For each workload, a set of graphs is presented that depict the timeseries data related to request latency, CPU utilization, memory utilization, and network utilization over the course of an experiment. All experiments run for a given workload are displayed in a single graph, and each experiment is labeled with $X\%$ or *auto*, signifying the percentage of traffic forwarded from the Edge to the Cloud. For instance, a label of 0% indicates that all traffic is handled by the Edge, while a label of 50% indicates that half the traffic is handled by the Edge. The label *auto* signifies that the default edge scheduler inside Knative Edge, as described in the implementation section, is used. For each of the timeseries graphs presented, we label three distinct phases with vertical indicators: first indicator shows the cold-start of the functions, the second shows the start of the second phase, and the third the start of the third phase.

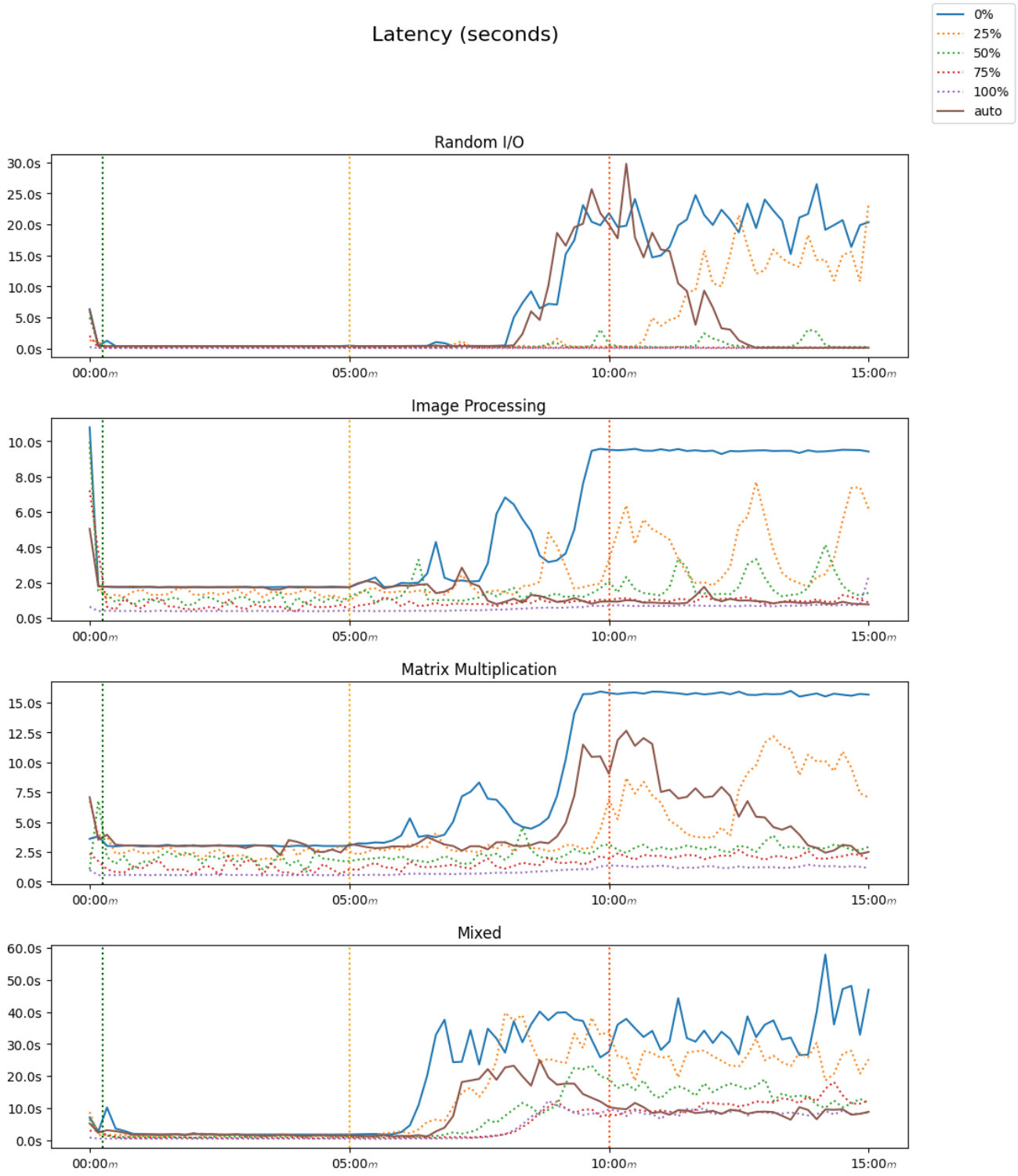


Figure 6.2: Average latency over all requests, measured in seconds.

From our results, we can see that the offloading functionality improves response times by a considerable amount.

For the random I/O and matrix multiplication workloads, we find that offloading 25% of the requests to the cloud is insufficient to prevent overloading the edge cluster, while offloading 50% results in only a slight increase in response times when the experiment is at maximum load. Our solution initially exhibits slower reaction times, leading to reduced response times. However, as more requests are offloaded to the Cloud, we observe that the response time converges towards the lower bound given by our edge network conditions.

In the image processing workload, we note that Eloud response times are lower than Edge due to limited edge devices' processing power compared to the cloud virtual machine. As the request rate increases, requests start being offloaded immediately, resulting in a decrease in response time. At maximum load, our solution shows a substantial improvement over no request offloading. Even with 25% of requests offloaded, there is still degradation in performance, while at 50%, occasional spikes in response times occur.

In the mixed scenario, a trend similar to that observed in previous workloads emerges. However, unlike the other workloads, which experience optimal performance when between 25% and 50% of the requests are offloaded, the mixed workload continues to exhibit some degradation in response times even when all requests are directed to the Cloud. Nonetheless, our solution convergences towards the lower bound of the response time as soon as the request rate is increased.

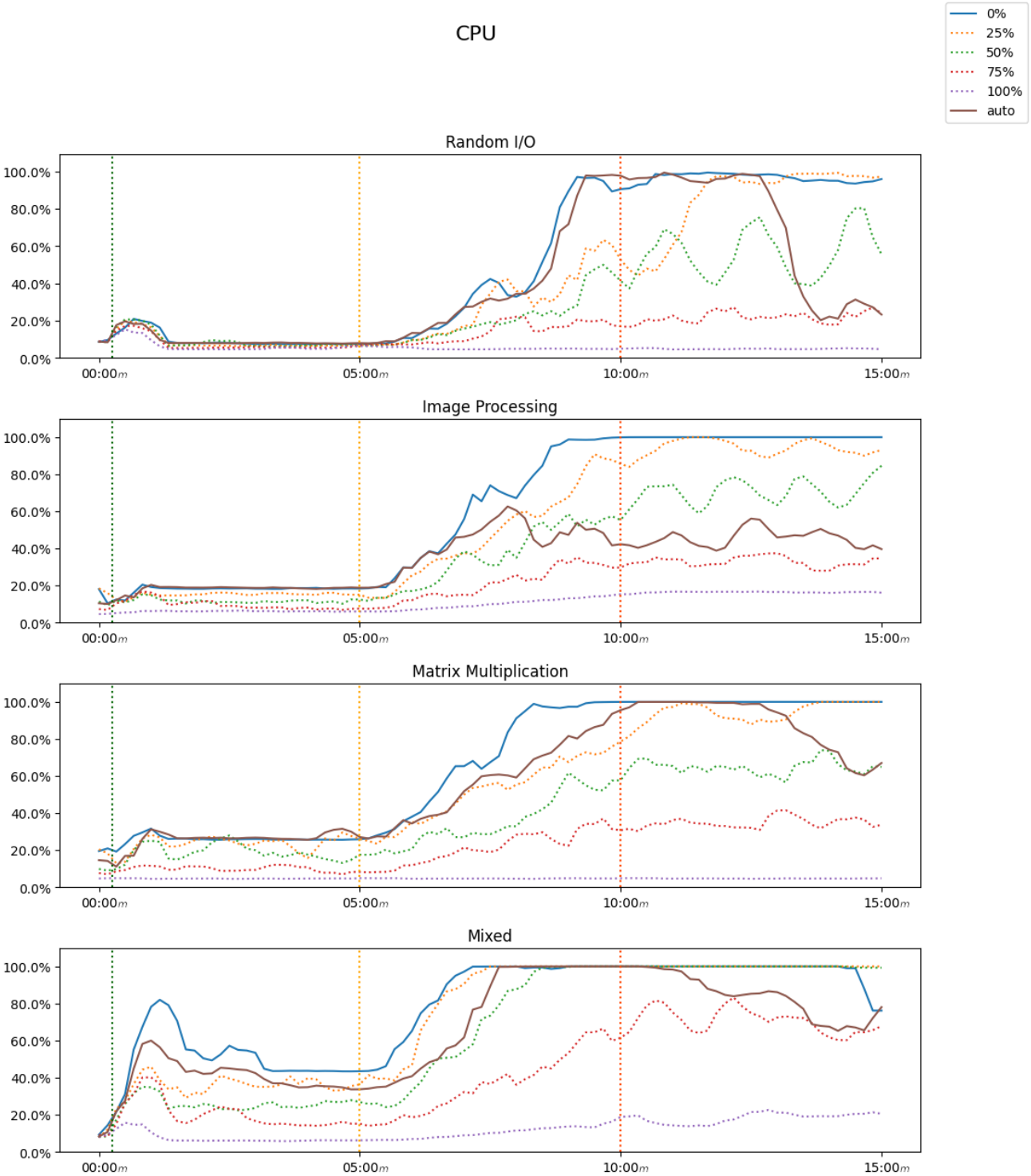


Figure 6.3: Average CPU utilization measured during the experiments

In our experiments, we have observed varying degrees of success in terms of CPU utilization. Our results indicate that our solution can effectively offload excessive load from the Edge cluster, but it may also underserve a significant number of requests. As our algorithm optimizes the response times of workloads, CPU utilization is reduced as a consequence. However, determining the optimal level of CPU utilization is not straightforward. If the CPUs of edge devices are not fully utilized but requests are being forwarded to the Cloud, it can be argued that the Edge is not efficiently utilized. Furthermore, reserving spare CPU resources could also benefit the Edge cluster when they are needed for possible bursts of requests or system operations.

In our random I/O runs, we observe a high CPU utilization on the Edge cluster, despite considering this workload to be light on the CPU. It is expected that the CPU utilization remains relatively consistent regardless of the number of requests in this scenario. While it may be tempting to attribute the overhead to the serverless platform, this would also imply that requests forwarded to the Cloud would experience the same overhead and could not improve latencies in the random I/O scenario. Nonetheless, we observe that both response times and CPU utilization are reduced, indicating that the CPU overhead is associated with the random I/O function.

Based on our findings, we observe similar patterns emerge. Following an initial period of high CPU utilization, there is a subsequent decrease in utilization as more requests are forwarded to the Cloud. In relative terms, the CPU utilization for all our experiment runs end up between the 50% and 75% scenarios.

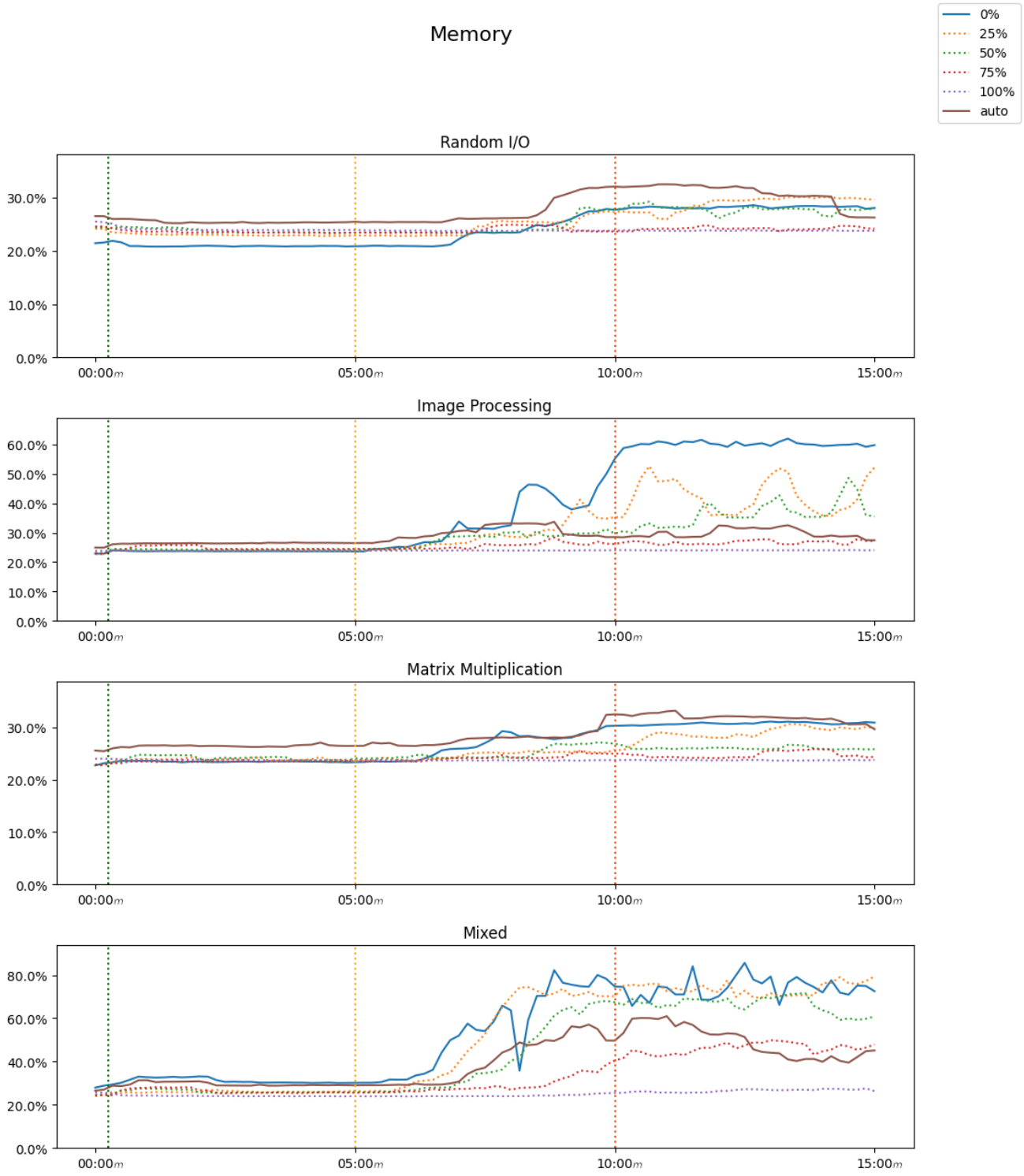


Figure 6.4: Average memory utilization measured during experiments

Based on our findings, we assess that, in most cases, memory utilization remains within reasonable bounds. Our initial assessment of the workloads suggested that we would observe a significant increase in memory utilization for both image processing and matrix multiplication. While the latter did see a noticeable increase in memory consumption, the former has seen a much lessened effect, performing similar to the random I/O workload, which we estimated to have no increase in memory.

The biggest increase in memory is observed in the mixed load experiment, which is the most demanding between all of the workloads. In conjunction with the increase in memory usage, we also observed a decrease in the stability of the system during the execution of the experiment. The effect of this can be seen in the variance of the memory usage measurements, which is much less present compared to other experiment runs. Comparing our solution with the fixed offloading scenarios, we can see a significant improvement in terms of both stability and memory utilization. When our solution is not active, we see that our system behaves poorly, even affecting our ability to measure its performance during the experiment. Even with a fixed offloading of 25%, we see an improvement in terms of the stability of the system, which can be seen in Figure 6.4 through the smaller variance in measurements. Our solution goes further than this, offloading around 50% at the beginning of the second phase and converging to 75% at the end of the experimental run.

In terms of memory usage, we find that our solution has a slight overhead compared to a vanilla Knative deployment in the initial phase, as more Kubernetes pods are running on the edge cluster. However, in the later stage of the experiment, we observe that our solution decreases the memory consumption due to requests being forwarded to the Cloud, relieving pressure from the edge cluster.

The effectiveness of our offloading algorithm varies from case to case. From the memory utilization perspective, we see that the image processing and mixed workloads see a noticeable decrease, however it is less pronounced during the random I/O and matrix multiplication, as the latter two experiments had little effect on the overall memory usage.

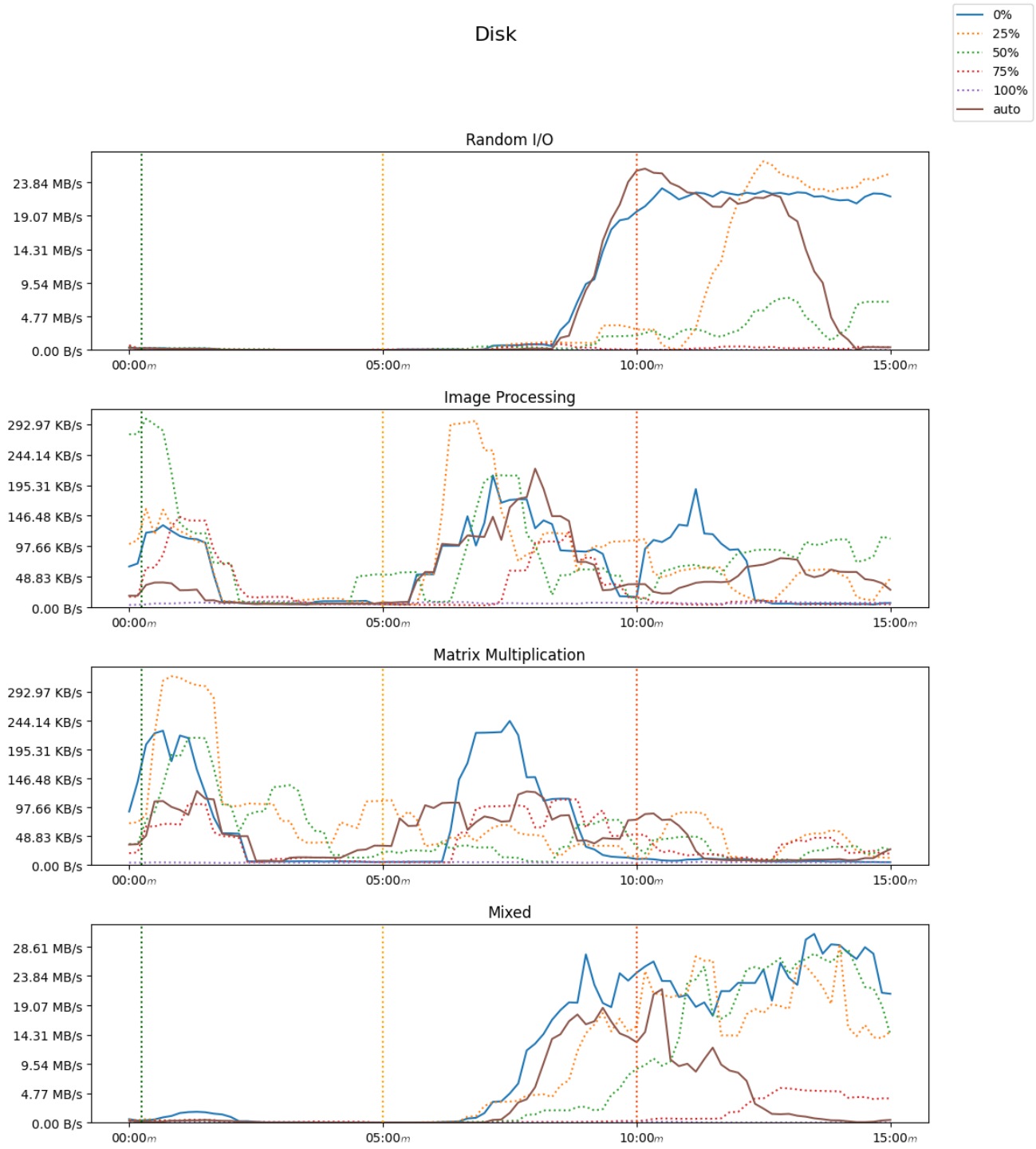


Figure 6.5: Disk utilization measured during experiments

For our experiments, we designed such that only the random I/O and mixed workload would have any disk activity. However, we observe a noticeable amount of disk usage throughout all of our experiments. Due to the timing in relation to the phase of the experiment, we can only deduce this disk activity is related to Knative spawning instances in order to execute the functions in our experiments. Our experiments have two distinct points where new instances would be spawned by Knative: at the beginning, when no instances of the functions are running, and during the increasing workload in the second phase of the experiment, when new instances of the functions need to be spawned in order to handle the increase in traffic. This can be distinctly seen in both the image processing and matrix multiplication workloads, both immediately after the start of the experiment and as soon as the workload is increased.

For the random I/O and mixed workloads, the disk utilization reaches around 24 MB/s during the maximum load phase, with some peaks of around 28 MB/s in the mixed workload. Compared to the fixed offloading scenario, we see a similar disk utilization as with offloading no requests during most of the run, however we note a large drops off near the last phase of the experiment. This suggests that our approach may be overshooting and offloading too many requests to the cloud, resulting in suboptimal edge resource utilization.

The scheduling algorithm we implemented for our platform uses the variance in the request timing in order to drive its decision on how much traffic is offloaded from the edge to the cloud. It does not directly measure disk activity, so the only way an overloaded disk can affect the offloading decision is through its effect on the latency. From our results, we can deduce that this methodology is inefficient for optimizing the disk utilization at the Edge and a more specialized algorithm is necessary, one that would account for disk activity directly - be it at node level or on a cluster-wide basis. Additionally, it is arguable whether high disk activity implies poor performance, as it could be one of the characteristics of the applications running on the platform. As such, a generic scheduling algorithm might not be well suited and one where both the hardware limitations and the behaviour of the application is learned and accounted for would be more desirable.

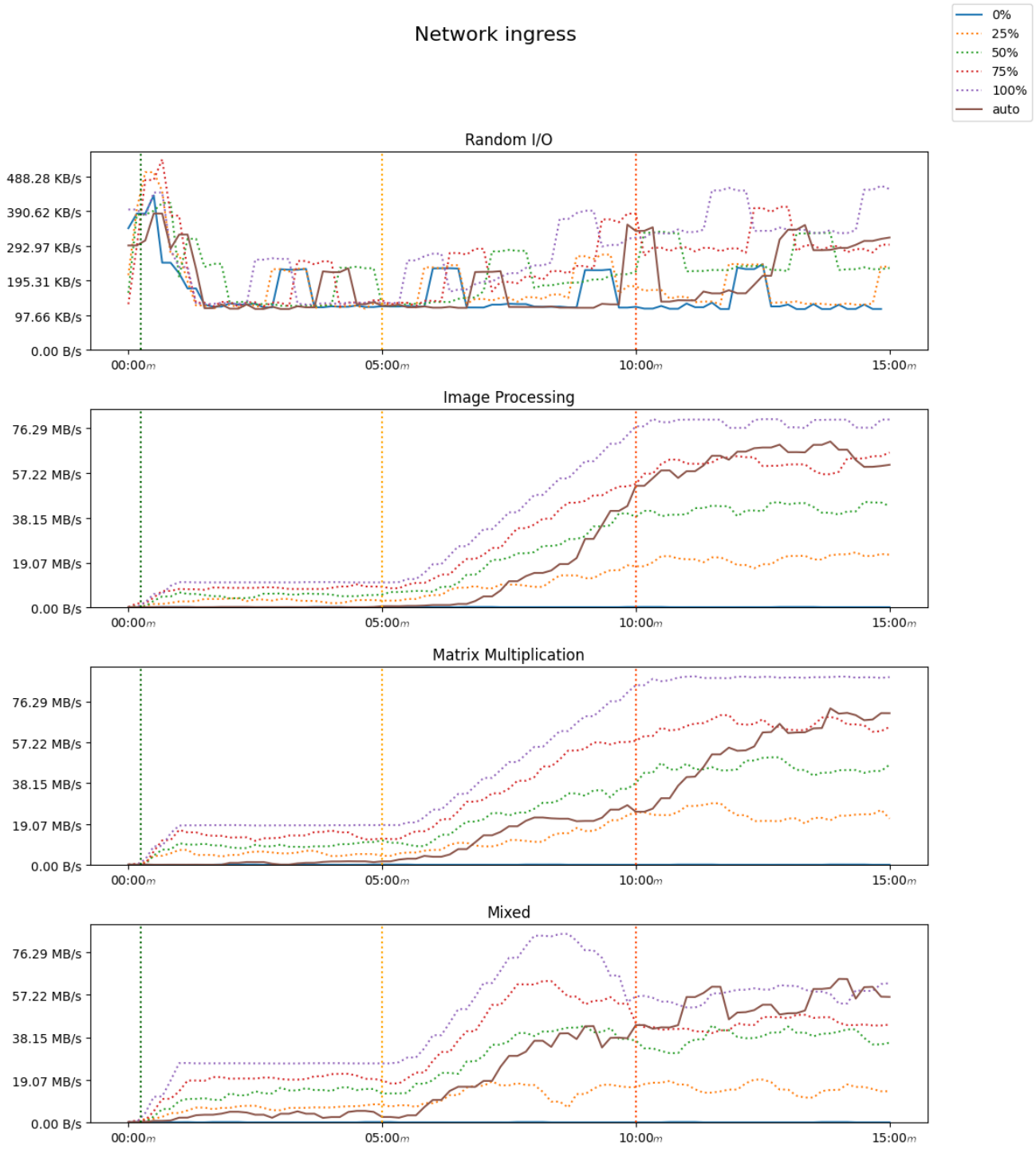


Figure 6.6: Inbound network utilization of the Cloud VM during the experiments

Although we anticipated minimal network utilization in the random I/O workload, our measurements indicate a persistent ingress network load of approximately 100 KB/s, with sporadic surges coinciding with the Cloud VM's increased workload. We attribute this additional network burden to the monitoring and measurement tools employed for assessing the virtual machine's performance. Furthermore, we highlight that the observed overhead is negligible in comparison to the bandwidth utilized in the other workloads.

One observation we make from the results above is that the inbound network bandwidth of the Cloud VM is never reaches 100 MB/s, which is the network bandwidth of our edge devices. Although both the image processing and matrix multiplication workloads appear to reach their limits, we shall see that this is due to the outbound bandwidth rather than inbound.

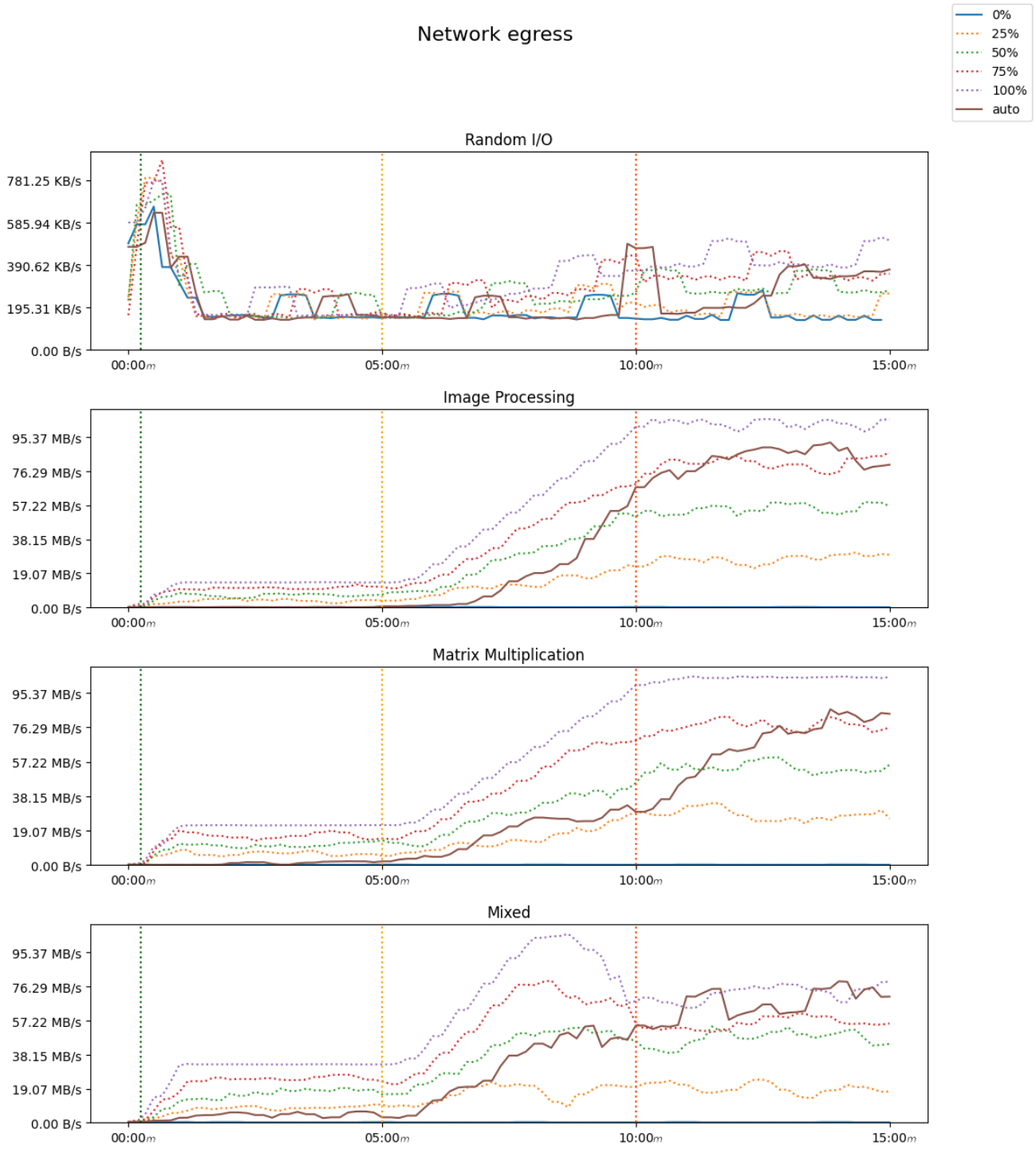


Figure 6.7: Outbound network utilization of the Cloud VM during the experiments

The network egress measured is directly comparable to the network ingress, as can be seen in the figures above. This is because the ratio between the egress and ingress for the cloud virtual machine is constant and depends on the workload.

Based on our results, we can see that bandwidth saturation is reached only when offloading all of the requests to the Cloud in the image processing and matrix multiplication workloads. Additionally, the mixed workload never reaches the maximum badwidth of 100 MB/s

In addition to our observations from the network ingress, we see that the image processing and matrix multiplication workloads are being limited by the network bandwidth between the Edge and Cloud cluster when 100% of the requests are being offloaded.

	Matrix Multiplication		Image Processing		Random I/O		Mixed	
Traffic	*Pct.*	*Total*	*Pct.*	*Total*	*Pct.*	*Total*	*Pct.*	*Total*
0%	0%	2406	0%	3627	0%	4852	0%	4152
25%	24.75%	2699	25.81%	4044	25.73%	5947	24.48%	5237
50%	49.07%	2664	49.04%	4045	50.39%	9371	49.67%	7486
75%	75.36%	2683	74.70%	3970	74.75%	9371	75.03%	8619
100%	100%	2668	100%	3969	100%	9408	100%	8725
auto	28.66%	2700	70.54%	4016	47.37%	6548	63.15%	7989

Table 6.7: For each workload type and traffic split, we display the total number of successful responses (right) and the percentage of requests offloaded to the Cloud (left).

In Table 6.7, we aim to show how requests are distributed between the Edge and Cloud for each experimental run. For each experiment, we show the total number of successful requests made and what percentage of those requests were served by the Cloud.

In order to validate that the Knative API Gateway splits the traffic as intended, we show the percentage of requests forwarded to the Cloud. For the runs with a static traffic split, we can see that the actual traffic ratio matches the static traffic ratio approximately. We note some slight deviations from our intended target in some experimental runs, however that is expected within reasonable bounds. In our case, we see a maximum deviation of approximately 2% which we consider acceptable.

In the experiments where the traffic split is calculated the using the algorithm in Section 5.1.2, we can see that the percentage of requests served by the Cloud depends on the specific workload. For the matrix multiplication workload, we see that the percentage is down to 13.28%, however the random I/O and mixed workloads go as high as 65.29% and 69.54% respectively. This indicates to us that the default offloading algorithm in Knative Edge can adapt to multiple types of scenarios.

Comparing the different traffic splits for our given experiments, we can see that offloading to the Cloud has a positive impact on the number of total requests served. However, we can also see that there are diminishing returns as the traffic split is increased.

In the matrix multiplication workload, we can see an increase of 293 of total requests served when 25% of the traffic is served by the Cloud compared to all traffic being served by the Edge. However, going from 25% to 50% decreases the total throughput, indicating that the most efficient traffic split ratio is close to 25%, which our appears to be converging to.

7

Discussion

In this section, we discuss the results of our paper in the context of the research questions proposed in the introduction.

Advantages and disadvantages of serverless at the Edge Through our experience in developing Knative Edge, we identify several key advantages and disadvantages of serverless computing in edge environments. These advantages and disadvantages include, but are not limited to:

- *Resource efficiency.* Serverless computing enables efficient resource utilization by scaling resources to match the workload. This is particularly beneficial in edge environments, where resources are limited and over-provisioning is a major concern.
- *Low latency.* Serverless computing enables low-latency processing by placing computing resources closer to data sources. This is particularly beneficial in edge environments, where low-latency processing is a major concern.
- *Vendor lock-in.* Serverless computing introduces vendor lock-in due to the lack of standardization of serverless platforms. This is particularly problematic in edge environments, where the diversity of hardware and software is greater than in conventional cloud environments.
- *Cold-starts.* Serverless computing introduces cold-starts by provisioning resources only when needed. This is particularly problematic in edge environments, where resources are limited and cold-start overhead has a much higher impact than in cloud environments.

- *Limited control.* Serverless computing limits the control over the underlying infrastructure. This is particularly problematic in edge environments, due to the diversity of hardware and software.

Edge-Cloud continuum The Edge-Cloud continuum is a vision of a unified computing environment that combines the best characteristics of both edge and cloud computing. The question of whether serverless computing can enable a Edge-Cloud continuum cannot be answered with a simple yes or no. Instead, it depends on the vision of the Edge-Cloud continuum and, as of now, there is no consensus on what the Edge-Cloud continuum should look like. Although a definition of the Edge-Cloud continuum can be found in the literature, it is not clear whether this definition is widely accepted. As a result, we cannot definitively answer the question of whether serverless computing can enable a Edge-Cloud continuum without first defining what the Edge-Cloud continuum should look like.

As per our definition - that is the Edge-Cloud continuum is a unified computing environment running accross a global infrastructure combining both Edge and Cloud cluster of computing resources - we believe that serverless computing can enable a Edge-Cloud continuum by providing a unified programming model for both Edge and Cloud environments, while also providing the necessary abstractions to hide the underlying infrastructure. In this way, the serverless functions can have a high degree of mobility, being able to run on vastly different infrastructures, while also being able to take advantage of the unique characteristics of each environment. For example, serverless functions can be deployed to the Edge to take advantage of the low-latency processing, while also being able to scale to the Cloud to take advantage of the high computing power. Additionally, it gives control to the platform provider to decide where to run the serverless functions, thus enabling more efficient resource utilization and better fault tolerance. In order to achieve this vision of a Edge-Cloud continuum, several challenges need to be addressed.

Challenges in developing a Edge-Cloud serverless framework The development of a Edge-Cloud serverless framework is a challenging task, both from a technical and a academic perspective. While the technical challenges in deploying either edge or cloud infrastructure are well understood, the challenges in unifying these two environments are not. In this regard, we note that fog computing has a big overlap with edge computing and research in this area can be leveraged to address the challenges in developing a Edge-Cloud serverless framework.

We can identify several ways in which serverless computing can be designed to enable a Edge-Cloud continuum. In our paper, we take a pragmatic approach and focus on a top-down approach, where the serverless functions are first deployed to a central cloud environment, and the edge environments synchronize with the cloud environment to deploy the functions at the edge. This approach is simple and straightforward, however it has several drawbacks. First, it required that the edge environment run an independent serverless platform, which might not be feasible in all scenarios. The alternative to this is run the control plane of the serverless platform in the cloud, and the worker nodes in the edge environment. This approach is more complex, but it has the advantage of being more flexible, scalable, and making it possible to have a global view of the entire system, thus enabling more advanced scheduling and resource management techniques. Second drawback of our approach is that, since there are multiple independent serverless platforms, there is no guarantee that the functions deployed in the cloud will be synchronized successfuly with the edge environments. Although we make an effort to address this issue by implementing a synchronization mechanism, it is not guaranteed that the synchronization will be successful. Third, our approach does not address the issue of cold-starts, which is a major concern in edge environments due to the limited resources available.

In terms of challenges, we identify several key challenges that need to be addressed to develop a Edge-Cloud serverless framework. These challenges include, but are not limited to:

- *Resource management.* Efficiently managing resources is a central concern in edge environments characterized by limitations in resources and a need to avoid excessive provisioning. Serverless computing addresses this challenge by automatically scaling running applications in response to varying workloads, optimizing resource utilization. However this process should also account for the various hardware and software capabilities that are present in edge environment, which cannot be trivially solved. This challenge is characterized by a mix of both technical and research challenges that need to be overcome to provide an ideal serverless platform for the Edge-Cloud continuum.

- *Scheduling.* Effectively scheduling tasks poses a significant challenge in edge environments, given the diversity of available resources and their constrained nature. The complexity in scheduling is raised even higher due to the mix of edge and cloud environments, which bring a very high degree of heterogeneity of hardware and software.

- *Fault tolerance.* Ensuring system resilience against failures is a crucial issue in edge environments due to external factors, such as regional power or network outages or the higher exposure to outside elements, which are harder to mitigate compared to cloud environment. The complex topology given by the Edge-Cloud continuum complicates the challenge further and robust solutions are needed.

- *Cold-start mitigation.* Serverless computing provides a solution to scale applications down to zero resources by only executing them when they are needed. Minimizing cold-start delays remains a key challenge for serverless computing, especially so in edge environments with constrained resources.

7.1 Future Challenges

In this section, we discuss the challenges that need to be addressed to realize the full potential of serverless computing in edge environments. In this regard, we identify several key challenges that need to be addressed to enable the widespread adoption of serverless computing in edge environments. These challenges include, but are not limited to:

- *Platform interoperability:* Platform interoperability refers to the ability of different serverless platforms to seamlessly interact and work together in a heterogeneous computing environment. In the context of edge computing, where the diversity is greater than in conventional cloud environments, interoperability becomes a crucial factor for achieving a cohesive and functional ecosystem.

- *BaaS at the Edge:* Backend as a Service (BaaS) plays a crucial role in serverless platforms, providing developers with pre-built backend services and features that simplify application development. However, the unique challenges posed by edge computing environments add complexity to the implementation of BaaS solutions.

- *Edge-native serverless platforms:* Although serverless platforms are designed to be cloud-agnostic, the majority of existing platforms are not designed for edge environments. As a result, there is a need for serverless platforms that are specifically designed for edge environments.

- *Federated serverless computing:* Federated serverless computing refers to the ability to deploy serverless applications across multiple edge locations, while maintaining a unified and cohesive ecosystem. This is a challenging task, as it requires the

integration of multiple serverless platforms, each with their own unique characteristics and requirements.

7.1.1 Platform interoperability

Platform interoperability refers to the ability of different serverless platforms to seamlessly interact and work together in a heterogeneous computing environment. In the context of edge computing, where the diversity is greater than in conventional cloud environments, interoperability becomes a crucial factor for achieving a cohesive and functional ecosystem. In the current landscape, various cloud providers offer their proprietary serverless platforms, each optimized for their specific infrastructure and services. Consequently, developers who wish to deploy applications across multiple edge locations, utilizing the advantages of different serverless platforms, encounter substantial obstacles.

The primary impediment to achieving platform interoperability is the divergence in runtime environments, containerization technologies, and function execution models employed by different serverless platforms. For instance, some platforms may utilize Docker containers, while others prefer lightweight, specialized containers, such as AWS Lambda's custom runtimes. Additionally, variations in function triggering mechanisms, event formats, and API specifications further complicate cross-platform compatibility.

Several strides have been made in this regard, both from the commercial side - through solutions such as `serverless.com` (37) - and from the academic side (38, 39). Interoperability between serverless platforms in the context of edge computing is a significant challenge, requiring collaborative efforts from both open-source communities and commercial cloud providers. Standardization, hybrid platform solutions, and cloud-agnostic development are promising approaches to address these challenges and pave the way for a unified and interoperable edge computing ecosystem. As both serverless and edge computing continue to evolve, addressing the platform interoperability challenge will be crucial in realizing their full potential.

7.1.2 BaaS at the Edge

Backend as a Service (BaaS) plays a crucial role in serverless platforms, providing developers with pre-built backend services and features that simplify application development. However, the unique challenges posed by edge computing environments add complexity to the implementation of BaaS solutions.

7. DISCUSSION

Several open-source and commercial BaaS platforms are available, however not all are specifically designed for either serverless or edge computing environments. Open-source solutions such as Appwrite and Supabase are designed for serverless environments, however their deployment in edge environment might pose challenges as their resource requirements are not optimized for edge environments. Although the vast majority BaaS solutions accompanying commercial serverless platforms are designed for cloud environments, we note the existence of some solutions that are specifically designed for edge environments. For example, AWS Greengrass and Azure IoT Edge are designed for edge environments, however they are not designed for serverless environments. Most notable commercial implementation is from Cloudflare KV, which is designed to work in tandem with Cloudflare Workers, a serverless platform designed for edge environments.

The principal characteristic of edge computing is the placement computing resources closer to data sources to minimize latency. In this context, BaaS must address the challenge of providing low-latency access to backend services to ensure seamless user experiences. As edge environments may have limited bandwidth and unstable connectivity, BaaS has to account for such constraints and implement mechanisms to mitigate their impact in edge environments.

Edge computing often involves processing sensitive data at the edge of the network, raising concerns about data privacy and security. On cloud environments, BaaS solutions are offered as a managed service by the cloud provider. Replicating this model in edge environments is challenging, as edge devices are not sufficiently powerful to run highly scalable and complex BaaS platforms. As a result, BaaS solutions for edge computing must be lightweight and resource-efficient, while still providing the same level of functionality and scalability as their cloud counterparts.

The convergence of serverless computing and edge computing offers new opportunities and challenges for BaaS solutions. Low latency, scalability, data privacy, and security are critical concerns that BaaS platforms must address to effectively serve edge environments. Although open-source platforms like Apache OpenWhisk and Knative do not offer comprehensive BaaS implementations for end-users, commercial platforms such as AWS Lambda and Azure Functions tend to offer mature and comprehensive solutions. However, these platforms are not designed for edge environments, and their implementation in edge environments is challenging. As a result, there is a need for BaaS solutions that are specifically designed for edge computing environments.

7.1.3 Edge-native serverless

Although opens-source serverless platforms are designed to be cloud-agnostic, the majority of existing platforms are not designed for edge environments. As a result, there is a need for serverless platforms that address the unique challenges posed by edge environments. In this regard, several commercial serverless platforms have been offered by cloud providers, such as AWS Lambda@Edge and Cloudflare Workers.

In terms of open-source solutions, we tested several platforms to evaluate their suitability for edge environments. We found that most available platforms are too heavy to run in resource-limited environments such as the one in our experiments. We found Knative to be the most suitable platform for our scenario, as it is lightweight enough to run on edge devices, while still providing the same level of functionality as other platforms. However, we note that Knative is not designed for edge environments, and its deployment in edge environments is challenging. As a result, we note that there is a need for serverless platforms that are specifically designed for edge environments.

8

Threats To Validity

Threats to the validity of the results of this thesis are inherent in any experimental study. It is important to consider these threats in order to establish the reliability and validity of the findings. The following is a discussion of the main threats to the validity of this study.

Internal validity refers to the accuracy of the causal relationships between the independent and dependent variables. In this study, the independent variables are the hardware and software specifications of the devices and the dependent variable is the performance of the serverless platform. One potential threat to the internal validity is the presence of extraneous variables that might impact the performance of the serverless platform. For example, other applications running on the devices could consume resources and impact the performance of the serverless platform. To this end, we minimized the impact of extraneous variables by running the experiments on dedicated devices with no other applications running. Another potential threat to the internal validity is the presence of confounding variables that might impact the performance of the serverless platform. For example, the performance of the serverless platform might be impacted by the network latency between the devices. To this end, we minimized the impact of confounding variables by running the experiments on the same network and by using the same hardware and software configurations for all devices.

External validity refers to the generalizability of the results to other populations and settings. In this study, the results are limited to the specific hardware and software configurations used in the experiment. The results might not be generalizable to other types of hardware or software configurations. Furthermore, the set of workloads used in the experiments may not be representative of all of the workloads seen in real-world scenarios.

Construct validity refers to the accuracy of the measurements used to assess the dependent variable. In this study, the dependent variable is the performance of the serverless

platform, which is measured by the response time and resource utilization. The accuracy of these measurements is dependent on the instrumentation and metrics used to gather data. A potential threat to the construct validity is the accuracy of the instrumentation and metrics used to measure the performance of the serverless platform. To this end, we used industry-standard tools and metrics to measure and report the performance of the serverless platform such that concerns about accuracy are mitigated.

Reliability refers to the consistency of the results over time and across different measurements. In this study, the results are based on a single experimental setup and a single run is shown for each experiment. The results might not be consistent if the experimental setup was changed in some way. In this regard, we designed the experiments to be repeatable and reproducible, and ran multiple runs in order to validate they are consistent. The experimental setup is described in detail in Section 6 and the source code for both the Knative Edge and the experiments used in this paper is available online[1].

[1] `https://github.com/jevvk/knative-edge`

9

Related Work

In this section, we present several research papers which are closely related to our presented solution. The literature contains relevant papers focusing on (a) scheduling functions in the resource-limited edge environment (40, 41) using various algorithms, from scheduling based on queuing theory to deep neural networks, (b) optimizing serverless resource usage in edge environments (42, 43), and (c) bringing serverless to the Edge-Cloud continuum (18, 44).

In their research paper, Wang et al. (40) showcase LaSS, a platform that uses a model-drive approach for running latency-sensitive serverless applications on the edge. LaSS is implemented on the OpenWhisk platform and makes use of queuing theory in order to determine the appropriate allocation for each serverless application and auto-scale the allocated resources in response to workload. The scheduling algorithm ensures a fair allocation of resources and guarantees a minimum of allocated resources for each function even when the edge is overloaded. Similarly, Tang et al. (41) propose a deep learning multi-agent task scheduling algorithm that improve resource utilization at the edge and evaluate their proposal using extensive simulation experiments. Whereas the authors focus on optimizing the scheduling of serverless applications on individual nodes at the edge through queueing theory or deep learning algorithms, we use the default round-robin scheduling algorithm in Knative and improve the response times by offloading work from the edge to the cloud using a sampling-based scheduling mechanism driven by the variance in the function response times.

Gadepalli et al. (43) consider that current sandboxing techniques used in FaaS, such as VMs and containers, consume too many resources to be feasible to run on edge devices efficiently. In this regard, the authors explore the potential of WebAssembly for serverless computing at the edge given its sandboxing capabilities and low overhead. Jeon et al. (42)

follow a novel approach in optimizing resource utilization at the edge by caching popular function dependencies using multi-agent deep reinforcement learning, thus also tackling cold-starts which are a main challenge in edge environments. In comparison, we reduce resource utilization by offloading the workload from the edge to the cloud, however we do not focus on optimizing the runtime overhead of the serverless platform.

Serverless Computing Fabric (SCF) is a solution presented by Nastic et al. (18) which aims to unlock the full potential of the Edge-Cloud continuum. In their paper, the authors present the reference architecture for SCF and discuss the opportunities and challenges which need to be overcome in order to achieve their vision. They identify several main oppotunities, such as edge-native backend services as complementary software for FaaS, sustainable resource usage at the edge, and enabling edge intelligence for AI-based applications. By contrast, our solution has a subset of the scope of the solution envisioned by the authors and does not include components such as edge-native backend services or infrastructure monitoring.

Ferry et al. (44) introduce a solution for the design, deployments, and maintenance of hybrid Cloud-Edge-IoT applications. The proposed solution implements a domain-specific modelling language that facilitates the specification of reusable and platform-independent hybrid applications on the Cloud-Edge-IoT continuum. Their novel solution is built from the ground-up to handle common IoT challenges such as poor connectivity and low computing resources. By comparison, our solution extends Knative, which is an existing open-source serverless platform and does not specifically target IoT scenarios. While the overhead of the platform might be too high in some IoT scenarios, we gain the flexibility and features of a mature serverless platform like Knative.

10

Conclusion

In this paper, we have presented Knative Edge, an extension of the serverless platform Knative, that enables deployments across edge and cloud environments and the offloading of requests from the edge devices to the cloud. We have shown that Knative Edge can be used to deploy a serverless application across edge and cloud environments and have demonstrated its offloading capability.

The topic of serverless computing on the Edge-Cloud continuum is still in its infancy. We believe that Knative Edge is a step in the right direction to enable such a vision. With this paper, we aim to provide valuable insights into the design of serverless platforms for a unified Edge-Cloud computing environment and will inspire future research in this area.

References

[1] Yuan Ai, Mugen Peng, and Kecheng Zhang. **Edge computing technologies for Internet of Things: a primer**. *Digital Communications and Networks*, **4**(2):77–86, 2018. 3

[2] Mahadev Satyanarayanan. **The emergence of edge computing**. *Computer*, **50**(1):30–39, 2017. 4

[3] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. **Globally distributed content delivery**. *IEEE Internet Computing*, **6**(5):50–58, 2002. 4

[4] Baotong Chen, Jiafu Wan, Antonio Celesti, Di Li, Haider Abbas, and Qin Zhang. **Edge computing in IoT-based manufacturing**. *IEEE Communications Magazine*, **56**(9):103–109, 2018. 5

[5] Qinglin Qi and Fei Tao. **A smart manufacturing service system based on edge computing, fog computing, and cloud computing**. *IEEE access*, **7**:86769–86777, 2019. 5

[6] Soraia Oueida, Yehia Kotb, Moayad Aloqaily, Yaser Jararweh, and Thar Baker. **An edge computing based smart healthcare framework for resource management**. *Sensors*, **18**(12):4307, 2018. 6

[7] Alaa Awad Abdellatif, Amr Mohamed, Carla Fabiana Chiasserini, Mounira Tlili, and Aiman Erbad. **Edge computing for smart health: Context-aware approaches, opportunities, and challenges**. *IEEE Network*, **33**(3):196–203, 2019. 6

[8] Latif U Khan, Ibrar Yaqoob, Nguyen H Tran, SM Ahsan Kazmi, Tri Nguyen Dang, and Choong Seon Hong. **Edge-computing-enabled**

smart cities: A comprehensive survey. *IEEE Internet of Things Journal*, **7**(10):10200–10232, 2020. 6

[9] Zhihan Lv, Dongliang Chen, Ranran Lou, and Qingjun Wang. **Intelligent edge computing based on machine learning for smart city**. *Future Generation Computer Systems*, **115**:90–99, 2021. 6

[10] Xuan Zhou, Ruimin Ke, Hao Yang, and Chenxi Liu. **When intelligent transportation systems sensing meets edge computing: Vision and challenges**. *Applied Sciences*, **11**(20):9680, 2021. 6

[11] Jie Lin, Wei Yu, Xinyu Yang, Peng Zhao, Hanlin Zhang, and Wei Zhao. **An edge computing based public vehicle system for smart transportation**. *IEEE Transactions on Vehicular Technology*, **69**(11):12635–12651, 2020. 6

[12] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. **Cloud Continuum: the definition**. *IEEE Access*, **10**:131876–131886, 2022. 6, 7

[13] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. **The internet of things, fog and cloud continuum: Integration and challenges**. *Internet of Things*, **3**:134–155, 2018. 6

[14] Mung Chiang and Tao Zhang. **Fog and IoT: An overview of research opportunities**. *IEEE Internet of things journal*, **3**(6):854–864, 2016. 6, 7

[15] Malika Bendechache, Sergej Svorobej, Patricia Takako Endo, and Theo Lynn. **Simulating resource management across the cloud-to-thing continuum: A survey and future directions**. *Future Internet*, **12**(6):95, 2020. 6

[16] Saadallah Kassir, Gustavo de Veciana, Nannan Wang, Xi Wang, and Paparao Palacharla. **Service placement for real-time applications: rate-adaptation and load-balancing at the network edge**. In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 207–215. IEEE, 2020. 7

[17] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. **A survey on mobile edge computing: The communication perspective**. *IEEE communications surveys & tutorials*, **19**(4):2322–2358, 2017. 7

[18] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. **A Serverless Computing Fabric for Edge & Cloud**. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–12. IEEE, 2022. 7, 69, 70

[19] Docker. `https://www.docker.com/`, May 2022. 8

[20] Kubernetes. `https://kubernetes.io/`, May 2022. 8

[21] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. **Cloud programming simplified: A berkeley view on serverless computing**. *arXiv preprint arXiv:1902.03383*, 2019. 8, 9

[22] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. **What serverless computing is and should become: The next phase of cloud computing**. *Communications of the ACM*, **64**(5):76–84, 2021. 9

[23] Kubernetes. **https://kubernetes.io**. 10

[24] Knative. **https://knative.dev**. 12, 22, 27

[25] Details on the 1.0 release of Knative. `https://knative.dev/blog/releases/knative-1.0/`, 2021. 12

[26] Apache Kafka. `https://kafka.apache.org/`, 2023. 15

[27] Redis Stream. `https://redis.io/docs/data-types/streams/`, 2023. 15

[28] RabbitMQ. `https://www.rabbitmq.com/`, 2023. 15

[29] Kyma. `https://kyma-project.io/`, May 2022. 22

[30] OpenFaaS. `https://www.openfaas.com/`, May 2022. 22

[31] Fission. https://github.com/fission/fission, May 2022. 22

[32] OpenLambda. https://github.com/open-lambda/open-lambda, May 2022. 22

[33] Apache OpenWhisk. https://openwhisk.apache.org/, May 2022. 22

[34] kubeless. https://github.com/vmware-archive/kubeless, May 2022. 22

[35] Fn. https://github.com/fnproject/fn, May 2022. 23

[36] IronFunctions. https://open.iron.io/, May 2022. 23

[37] serverless.com. https://www.serverless.com/, May 2022. 63

[38] Pedro Rodrigues, Filipe Freitas, and José Simão. **QuickFaaS: Providing Portability and Interoperability between FaaS Platforms**. *Future Internet*, **14**(12):360, 2022. 63

[39] Paul Castro, Vatche Isahagian, Vinod Muthusamy, and Aleksander Slominski. **Hybrid serverless computing: Opportunities and challenges**. *Serverless Computing: Principles and Paradigms*, pages 43–77, 2023. 63

[40] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. **Lass: Running latency sensitive serverless computations at the edge**. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–251, 2021. 69

[41] Qinqin Tang, Renchao Xie, Fei Richard Yu, Tianjiao Chen, Ran Zhang, Tao Huang, and Yunjie Liu. **Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach**. *IEEE Internet of Things Journal*, **9**(20):19634–19648, 2022. 69

[42] Hongseok Jeon, Seungjae Shin, Chunglae Cho, and Seunghyun Yoon. **Deep Reinforcement Learning for QoS-Aware Package Caching in Serverless Edge Computing**. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2021. 69

[43] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. **Challenges and opportunities for efficient serverless computing at the edge**. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615. IEEE, 2019. 69

[44] Nicolas Ferry, Rustem Dautov, and Hui Song. **Towards a model-based serverless platform for the cloud-edge-iot continuum**. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 851–858. IEEE, 2022. 69, 70