MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Serverless platform within the Kubernetes infrastructure

Master's Thesis

MAREK BERNHAUSER

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Serverless platform within the Kubernetes infrastructure

Master's Thesis

MAREK BERNHAUSER

Advisor: RNDr. Lukáš Daubner

Department of Computer Systems and Communications

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Bernhauser

Advisor: RNDr. Lukáš Daubner

Acknowledgements

I would like to thank my advisor RNDr. Lukáš Daubner, for his support and thorough feedback. I would also like to thank Bc. Adrián Rošinec for his consultations. Finally, I would like to thank my family and friends for their continued support throughout my whole studies.

Abstract

The goal of this thesis is to survey available Function-as-a-Service frameworks for a new platform at the Institute of Computer Science at Masaryk University. It also describes the platform's target users and multiple use cases. A methodology is devised to select frameworks that fit the requirements, and a selected few frameworks are compared in detail. A lightweight application is then deployed for validation. The outcome of this thesis is a serverless platform, which is ready to be deployed to the e-INFRA infrastructure.

Keywords

serverless, Function-as-a-Service, Kubernetes, Fission, e-INFRA

Contents

List of Tables

List of Figures

Introduction

Cloud computing has become an important part of modern software development and data processing. It mitigates the need to own and maintain expensive hardware while offering performance unmatched by traditional desktop computers. Within the last years, a new paradigm called serverless computing has emerged within cloud computing ecosystem. It aims to decrease the running costs and maintenance required. A principal subset of serverless computing called Function-as-a-Service further strives to lower the cost of entry by simplifying the deployment of applications [1].

The Institute of Computer Science (ICS) of Masaryk University is a member of the e-INFRA cloud infrastructure[1]. The infrastructure, among others, contains a Kubernetes platform aimed at providing computational resources for the scientific or academic community.

This thesis aims to allow ICS to expand the service catalogue by exploring currently available Function-as-a-Service type frameworks and using one of them to create a platform for deploying serverless applications.

Chapter content

The first chapter aims to briefly explain the problems tackled in this thesis and the motivation behind it.

The second chapter describes the environment and the tools used, focusing on Kubernetes. The benefits it provides and principal components are described.

The third chapter is focused on users. It identifies target user groups and specifies their use cases for Function-as-a-Service.

An overview of serverless computing is described in the fourth chapter. The main features of serverless computing and their differences from standard cloud computing are described in more detail. It summarises its development and the ideas behind it. The chapter also discusses types of serverless computing, with a focus on Function-as-a-Service.

1. https://www.e-infra.cz/en

This common background is followed by chapter five that describes the inclusion and exclusion criteria used to filter the frameworks, and lastly, a brief description of the selected frameworks.

The sixth chapter gives more detail on the five suitable frameworks selected in chapter five and compares performance metrics for three of them.

Chapter seven includes details regarding the Fission framework.

The final chapter concludes the thesis and describes the problems solved and the avenues to explore in potential future work.

1 Problem definition

Cloud services are being used more and more in scientific research and other academic work. Therefore the Institute of Computer Science (ICS) participates in the e-INFRA[1] project to deliver the computational resources needed to execute the provided code.

However, the current solution is problematic as it requires scientists or other users to not only have knowledge in their respective fields. They also have to learn how to use the existing infrastructure. This encompasses the need to properly configure their applications, create workflows, containers, and correctly deploy them.

In order to stay competitive with modern software development, which tries to minimalize the time users have to spend not creating a value, a serverless approach will be used.

The technical challenge lies in selecting the technology that would best fit the needs of the scientific and academic community at Masaryk University and would fit within the existing Kubernetes infrastructure. However, the benefits of creating a solution that would be user-friendly, scalable, and robust are many.

They range from the increased productivity of academic staff and better accessibility of cloud computing, to simplifying the continuous integration and deployment of software developed by ICS employees. Increased accessibility also benefits non-technical users by lowering the entry boundaries for application development. Another potential benefit could be the revenue generated from sales of computing time on the platform to external customers.

1. https://www.e-infra.cz/en

2 Environment Description

This chapter describes the environment and focuses on the tools used to develop this thesis. It explains the main components of the Kubernetes platform and discusses the purposes of the Kubernetes architecture.

2.1 E-INFRA

E-INFRA CZ is an infrastructure operated by CESNET[1], CERIT-SC[2], and IT4Innovations[3]. Its goal is to provide capacities for storing and processing scientific or generally academic data in the Czech republic on a cloud [2].

2.2 Kubernetes

Kubernetes is an open-source container orchestration platform first designed by Google [3], now maintained by the Cloud Native Computing Foundation[4]. Kubernetes streamlines software deployment by providing all of the necessary code and configurations in the form of standardized containers with predefined ways of communicating.

Kubernetes performs many of the tasks that standard operating systems perform, but for cloud applications. Namely, it abstracts the differences between various cloud providers and private cloud, schedules, and scales them [4]. By abstracting the infrastructure, developers do not have to worry about deployment configurations for different servers or provisioning virtual machines. The immutability of containers means that a new version can be deployed while the old is still running. Afterward, traffic can be redirected, or the update can be easily rolled back [5]. Scheduling the applications on any available machine helps increase efficiency by lowering idle times. Additionally, Kubernetes can self-heal by automatically redeploying failed applications [6], increasing system reliability.

---

1. https://www.cesnet.cz/?lang=en
2. https://www.cerit-sc.cz/
3. https://www.it4i.cz/en
4. https://www.cncf.io/

2.2.1 Components

The main components of Kubernetes platform are:

- A *container* is a package of software that contains code and the dependencies necessary to run it. Container code is immutable, and they separate the program from the hardware running it, making it easier to migrate [7].
- A *pod* contains one or more containers sharing resources and existing in a shared context [8]. As containers in a pod are tightly coupled, they usually contain containers necessary to run one application. Having pods is a massive advantage for scaling applications because scaling is done by simply copying existing pods.
- A *node* is a machine that runs software, which allows it to contain multiple pods, which then contain running containers [9].
- A *cluster* consists of nodes and a control panel, which maintains the state of the cluster. Each node can be a separate virtual or physical machine, or multiple nodes can run on a single machine [10].
- A *kubelet* manages a single node by creating and maintaining containers and pods specified in a configuration YAML[5] file [11].
- A *kube-proxy* facilitates the network communication between users and pods in the node [12].
- A *namespace* is used to separate objects into groups. They allow users in teams or teams within organizations to work in isolated environments. Objects can not share the same name within a namespace, but the same name in multiple namespaces is allowed [13].
- *Kubectl* is a command-line tool that facilitates the control over the Kubernetes cluster via Kubernetes API [14].

5. https://yaml.org/

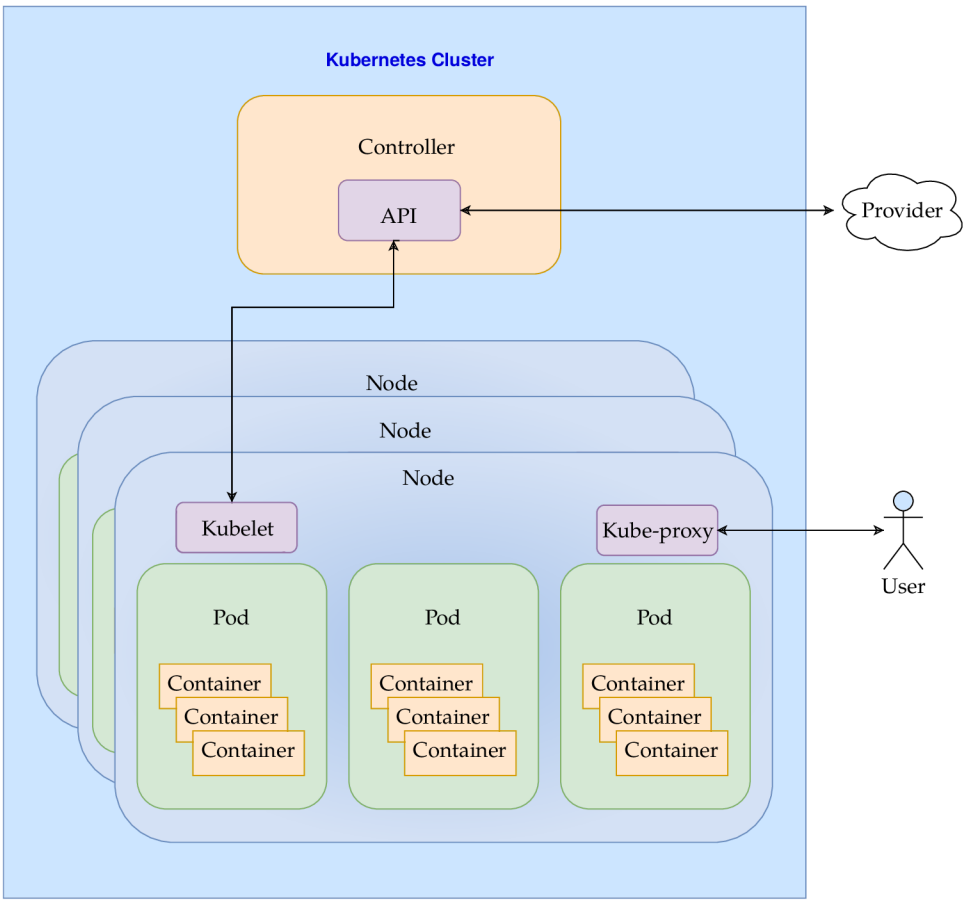


Figure 2.1: Main components of Kubernetes architecture [15]

2.2.2 Rancher

Rancher is an open-source platform that allows the operation of Kubernetes clusters on a larger scale and in production. It provides a centralized location for the maintenance of deployed applications with controls accessible via GUI[6] or command line. Rancher supports security policies, application monitoring, and metrics [16].

It helps developers to create modern microservices without worrying about complex deployment.

6. Graphical user interface

2.2.3 Minikube

Minikube is an open-source Kubernetes platform that allows a local cluster deployment. Like Rancher, it is used to deploy applications and comes with a GUI for monitoring. It differs in only supporting a single node [17]. However, as it runs locally, the developers are allowed a greater level of control over the cluster.

3 Users

When developing a new platform, knowing its users is vital. This chapter explores the target user groups and presents the expected use cases for each one. The cloud team responsible for the infrastructure maintenance is also briefly introduced.

3.1 Automated Infrastructure Systems team

Automated Infrastructure Systems is a department of the IT Infrastructure Division at the Institute of Computer Science. This department encompasses the team responsible for developing and maintaining cloud solutions.

The requirements, usability, and available technologies were discussed with the team, as they will be responsible for operating the platform. Therefore the methodology and selected technologies are based on the team's current needs and experience.

3.2 Target user group

The primary target group are the users of e-INFRA infrastructure. Currently (spring 2022), e-INFRA has approximately 5000 users, with 300 active according to the cloud team. These are mainly members of the scientific community in the Czech republic. They generally have a university education. However, it does not have to be in informatics, though they are expected to write scripts and have user-level knowledge of using a command line.

The employees of the Institute of Computer Science are the secondary target users. They have experience developing and deploying applications. They might be interested in the platform to deploy production-ready applications or to use it for development purposes.

A tertiary target user group consists of the students of Masaryk University, mainly from the Faculty of Informatics. These users could use the platform to develop or deploy programs required for their school projects. Students have a university-level education in infor-

matics and some experience deploying applications, using command line and standard tools.

3.3 Use cases

The selected use cases were discussed with the cloud team to represent the most common uses for the platform. The types of use cases are also similar to those utilized in the industry, as identified by Eismann [18]. The identified use cases are:

Scientific community

- Batch processing of data: Infrequent processing of more substantial amounts of previously collected data. An application can scale to zero when no data is being processed and scale-up parallel instances to accommodate extensive data.
- Processing of data from devices deployed in a field: Measuring devices deployed in a field connected directly to the internet or via an edge[1] device can upload their data and have it immediately saved or processed. An application only runs infrequently when the device is ready to transmit the data.
- Backend for displaying or accessing data: An application serving processed data to a broader scientific or general community.

Institute of Computer Science employees

- Continuous integration and deployment: Removing manual tasks by automating the deployment and removing the pre-provisioned hosts, running unit tests, or notifying service users.
- Events processing: Processing events from a message queue, saving them to the database, or sending notifications in case of error messages.
- Migrate existing services to a newer platform: Deploy existing APIs to a platform that can better handle the required loads.

1. https://www.techtarget.com/searchnetworking/definition/edge-device

- Schedule-based jobs: An application can be inactive most of the time, only initializing when scheduled by Cron job[2] to perform tasks like clearing cache, sending messages, or others.

Students

- Deploy code for thesis: To demonstrate the working of their thesis, students can deploy APIs[3], backends for mobile applications, or other code on the platform.

Administrators

- Provide and maintain a cloud platform for users.
- Monitor the usage of services by users.

2. https://www.hostinger.com/tutorials/cron-job
3. Application programming interface

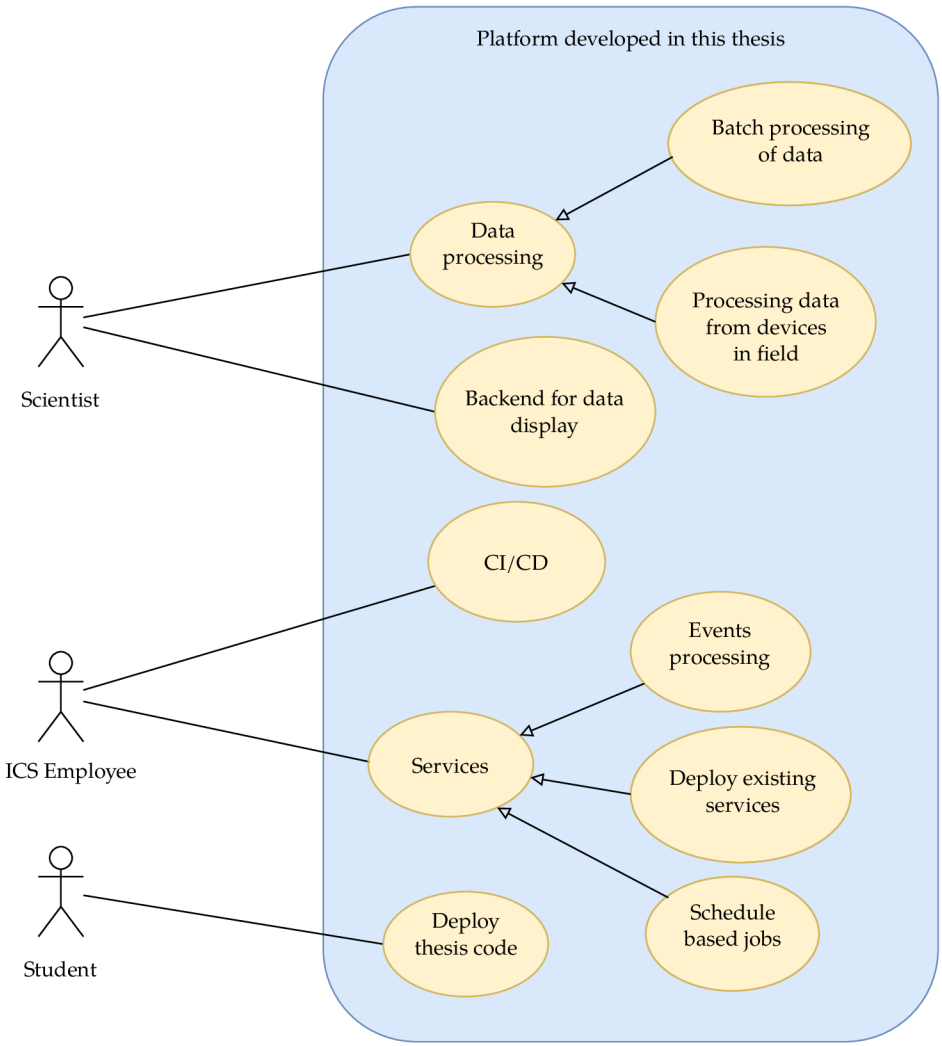


Figure 3.1: Use case diagram

4 Serverless technologies and Function-as-a-Service

Serverless is a cloud-based deployment model where the cloud provider is responsible for providing the resources needed to execute code [19] directly. The model is good at scaling the resources up to meet the current user demand or to scale down to zero when the application stops running [20] which saves on running costs. The main advantage from the user's perspective is increased productivity by reducing the maintenance and allowing them to focus on development.

4.1 Development of serverless computing over time

Before the concept of serverless computing, deploying an application meant either renting a whole server or at least a portion of it for a virtual machine. However, determining the correct scale was challenging. It could result in either buying too much hardware and squandering money by paying for idle time or buying too little and running into bottlenecks.

With the development of cloud computing also came the concept of providing services. These generally aimed to deliver user-friendly and straightforward solutions to customers following the service-oriented architecture [21], each providing a slightly different type of service. As a result, many different "as a service" models were created, eventually leading to them being called Everything-as-a-Service [22].

However, only three principal models were defined by NIST[1] [23]:

1. *Software-as-a-Service* (*SaaS*) is the highest level of abstraction, where customers can access applications via a web browser and can not control the cloud infrastructure.

2. *Platform-as-a-Service* (*PaaS*) provides a complete platform where customers can develop and deploy code without managing the infrastructure.

3. *Infrastructure-as-a-Service* (*IaaS*) provides the infrastructure necessary to run arbitrary software. The customers do not manage

1. National Institute of Standards and Technology, an agency of the United States Department of Commerce

the infrastructure; however, they can manage the operating system.

Nevertheless, the development of new models did not stop with NIST's definition of the three models. One of the new models that have gained popularity since the mid-2010s is serverless computing. This increase in popularity was primarily caused by the broader adoption of Amazon's AWS Lambda [24].

Since then, the growth in popularity has not slowed down, with many different companies now offering their products. Among other popular services are Microsofts Azure Functions and Google Cloud Functions.

The serverless market's expected compound annual growth rate[2] (CAGR) between the years 2021 and 2027 is 23% [25]. Such a rapidly growing market validates the need of having a serverless platform at the Institute of Computer Science, as more and more users will come to expect the features associated with these platforms.

4.2 Features of serverless computing

Many features of serverless computing are different from traditional cloud computing. This section describes the features that differ the most and discusses their potential advantages or disadvantages.

4.2.1 Statelessness

One of the main features of the serverless model is stateless computing. In a standard stateful application, the previous state is known and can be potentially returned to or influence current actions. When using a stateful application on the Internet, it has to run every time on one server that contains its state [26]. If a different server has to be used, the application state must be migrated. However, this can be difficult as certain aspects of servers can not be easily changed, making migrations in a cloud environment difficult.

With stateless computing, each application run is isolated from the previous runs or other concurrent running versions. This isolation

2. https://www.investopedia.com/terms/c/cagr.asp

dramatically increases robustness, as crashes can not influence future states, and any calculations can be easily repeated.

The stateless model even simplifies deployment, as any server can be orchestrated for any role [27], and applications can be easily moved between them.

With server management being simplified, one disadvantage of stateless computing is increased complexity in network management. The communication between many different types of devices across numerous networks has to be orchestrated [27]. This is especially the case in the Internet of Things devices, where efficiency is important, and best routing might not be known ahead of time. Some requests might be sent to the internet, processed by an edge device, or directly by a device on a local network [28].

4.2.2 Scalability

A significant advantage of serverless computing lies in simplified scaling when the same application can be deployed on multiple servers and users can access any of them, even changing servers between uses. The application can also scale up or down the number of instances running on a single server based on current demand. Scaling lowers the running costs when the demand is low and also limits the resources used by the server by not keeping numerous virtual machines mostly idling. However, it can become less efficient than the standard cloud in cases of constant high utilization and very predictable usage patterns, as it introduces inevitable overhead.

4.2.3 Security

The security of serverless applications compared to the traditional cloud is much more carried by the solution provider. With the customer having no control over the hardware, virtual machines, or operating system, the attack surface that users need to counter is narrowed. However, attacks on the service provider can be performed, which would affect every running application.

A potential security problem can be caused by the nature of loosely coupled applications that communicate together. This creates different

attack options, such as intercepting the communication within the application components.

A concern with third-party solutions is processing private data on a public server. A way of mitigating this risk is deploying a serverless framework on a privately-owned cloud.

4.2.4 Price

The price of serverless computing is one of its main advantages. With the ability to scale down to zero, the customers have to pay only for the time the application is running. In traditional cloud computing, the customers would have to pay for the ongoing maintenance of the servers or runtime of the virtual machines, even when these were not fully utilized [20]. The main factors determining the price are the number of requests and the computation time used. The provisioning time is usually not billed and is born by the provider.

4.3 Typical use cases

The typical use cases serverless computing is used for are synchronous APIs at 28% of use cases, asynchronous processing of events at 27% of use cases, large batch tasks at 23%, and monitoring other systems at 20% [18].

The main type of workload are short-running functions, typically up to a few seconds long. Still, over 20% of functions require multiple minutes to finish [18]. The workload is typically not distributed evenly over time. It is concentrated in short, and unexpected load spiked in most use cases [18].

The typical reasons customers adopt the serverless model are cost-saving, reduced or simplified operations, and the ease of scalability [29].

4.4 Types of serverless computing

The serverless model implements modern practices of providing services to users, with common examples being Containers-as-a-Service, where developers only manage their containers and code, not the un-

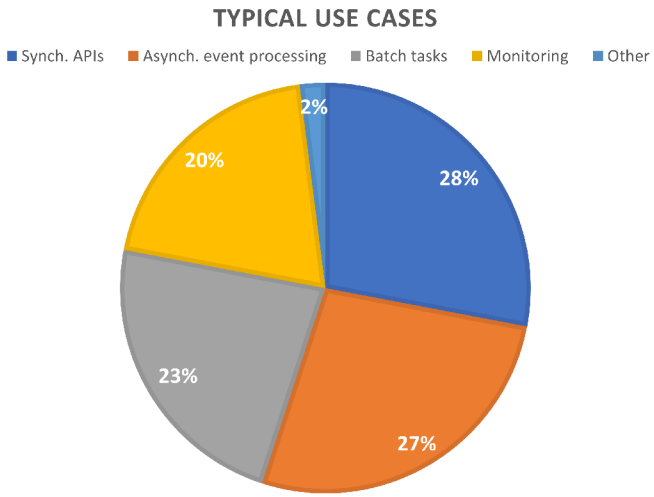


Figure 4.1: Typical use cases as identified by Eismann et al.[18]

derlying infrastructure. Backend-as-a-Service provides a platform for developers to focus on developing the frontend of their applications while using third-party libraries for the backend.

4.4.1 Function-as-a-Service

The majority of developers associate serverless computing with Function-as-a-Service or FaaS model [30]. FaaS is a subset of serverless computing. It is a way of implementing serverless computing, where functions run as stateless applications or containers without the developers needing to build and run their code [31]. The containers and the stateless applications execute on a cloud-based infrastructure that the provider maintains.

FaaS encourages distributed system design driven by events and loose coupling of functions. It is excellent for stateless computing, where each function is treated as unique and does not carry any historical information.

However, a disadvantage of FaaS is a cold start. As the functions can scale down to zero when not used for a few minutes, calling such processes requires creating new instances and loading them up, as depicted in figure 4.2. The time it takes is not insignificant and can be up to a couple of seconds [32]. Therefore FaaS is best utilized

in applications that require high throughput and are not limited by higher startup latency.

Higher latency is reflected in the real-world usage of serverless functions when only 2% of them are designed to be real-time applications [18]. Ways of improving startup latency range from preserving a small number of containers constantly warm for immediate use to reducing overhead when creating new instances.

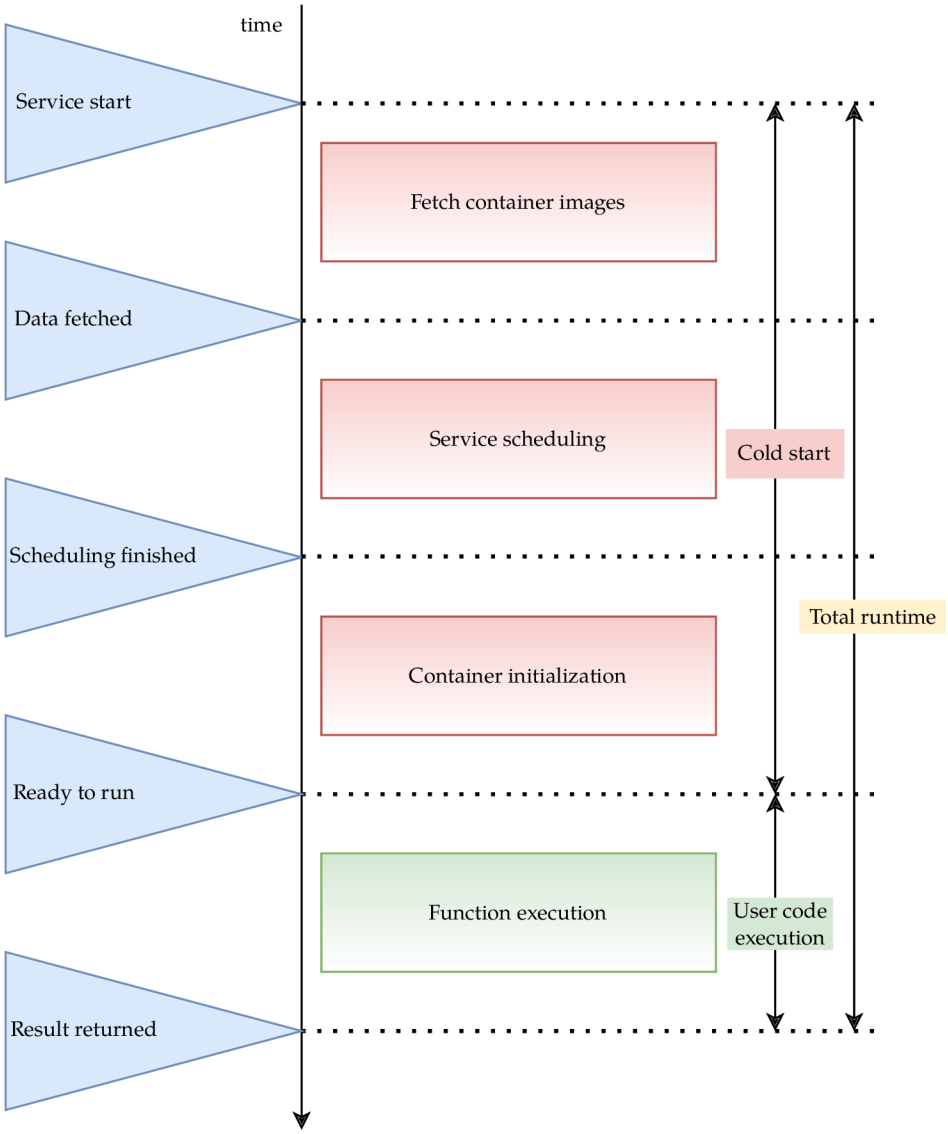


Figure 4.2: FaaS service initialization process [33]

5 Technology analysis

This chapter explores the process of finding serverless frameworks. With the available frameworks identified, a set of inclusion and exclusion criteria is formulated. The frameworks are then examined to determine which pass the criteria and can be studied in greater detail.

5.1 Technology selection process

With serverless computing gaining popularity, many frameworks exist, allowing the implementation of serverless support in a private cloud. They generally aim to achieve similar goals to each other, namely making application development simpler, faster, and cheaper.

To achieve this, they implement a variety of different approaches. That, however, means one does not simply choose the first available framework. Picking the right solution is also not easy, as many articles confuse FaaS and serverless terminology. Other problems are caused by a lack of independent comparisons, as many creators write the comparisons themselves.

Therefore the first step in deploying a serverless platform is exploring the potential frameworks and creating a collection from which a framework that best fits the purpose can be chosen.

The starting point for research was a blog post by Daniel Whatmuff [34] mentioned in the assignment of this thesis. It contains six popular frameworks with their comparisons. Similar blogs, such as one by Adrian Kosmaczewski [35], contained a wide variety of the more popular frameworks.
The second step utilized so-called "awesome lists", compilations curated by the community to include helpful information about projects and listed on GitHub[1]. The most exhaustive list found was created by user anaibol and contained a multitude of different frameworks with many other popular tools.
The next step included using search engines to identify any frameworks that previous steps could have missed. However, no new frameworks were identified this way.

1. https://github.com/

The last part of the research included finding every framework on GitHub or Stackoverflow[2] to validate their preliminary usability.

Overall 55 potential frameworks were identified. If any other frameworks exist that were not found during the research, they are presumed to have a user base and industry adoption that are too small for the purposes of this thesis.

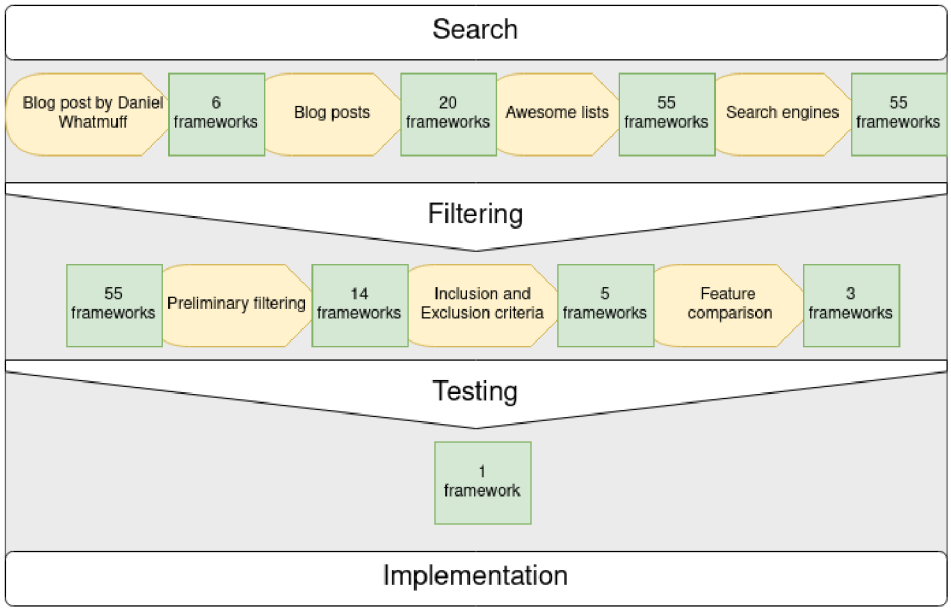


Figure 5.1: Visualized methodology

5.2 Technology filtering process

Detailed testing of every framework in the collection is outside the scope of this thesis. Moreover, not all frameworks are well suited for the intended purpose outlined in chapter 1. Therefore, the second step of the selection process was to filter available frameworks based on the critedia defined by the thesis assignment.

Three criteria were selected for preliminary filtering, which could be readily identified on the framework Github pages or the vendor website. The preliminary criterion are:

2. https://stackoverflow.com/

- the framework must support Kubernetes
- the framework must not support Node.js only
- the framework must not be a third party extension for existing commercial frameworks, as it needs to be deployed on a local cloud

Most frameworks supported just Node.js or were solely for AWS Lambda, with only 14 frameworks passing the first stage filtering. The frameworks that did not pass preliminary filtering will not be further considered.

In order to give equal opportunity to every remaining framework, a set of inclusion and exclusion criteria was formulated based on the requirements of the cloud team of the Institute of Computer Science. The framework must satisfy all inclusion criteria to be further considered in this thesis. Additionally, satisfying any exclusion criteria leads to disregarding the framework.
To meet the inclusion criteria, the framework must:

- be Function-as-a-service (FaaS) type of service
- be free to use or have a free to use version; in the latter case, only features of the free to use version will be evaluated
- have user documentation for setting up

To meet the exclusion criteria, the framework must:

- not be actively maintained, which in such a fast-moving field is having a commit activity on Github or releasing a new version over the past year and not claiming the end of support
- not officially support programming languages that are the most commonly used by teams at the Institute for Computer Science (C#, Java, Python, and JavaScript)

5.3 Available technologies

Relevant frameworks that were not excluded in preliminary filtering and were further investigated are summarized in this section and the comparison results are summarized in table 5.1.

Heroku[3] is a proprietary paid cloud Platform-as-a-Service (PaaS), owned by Salesfoce.com[4]. Presently, Heroku does not support C# and, as such, meets exclusion criteria while simultaneously not meeting multiple inclusion criteria of being free to use and a FaaS framework. It, therefore, will not be considered any further.

Kubeless[5] is Kubernetes native framework. The maintenance of Kubeless has ended at the end of 2021 and, therefore, will not be considered any further.

Knative[6] is a type of service named Container-as-a-Service or Caas. It requires users to manage their containers, not only functions. Not being FaaS means Knative does not meet inclusion criteria and will not be considered further.

Fn project[7] is a cloud-based serverless platform. Currently, support for C# is only community-based, and therefore Fn project meets the exclusion criteria.

Triggermesh[8] is an event-driven integration platform that has recently released an open-source version. It allows developers to deploy and manage functions on multiple clouds or on-premise.

Nuclio[9] is a serverless platform focused on data-science applications with open-source and enterprise editions. Nuclio comes with the support of GPU[10] computing and machine learning.

Dapr[11] is a runtime system designed to support serverless computing. It requires code to be deployed, running next to it, and talking via an API[12]. Therefore it is not a FaaS framework and does not meet inclusion criteria.

3. https://www.heroku.com/
4. https://www.salesforce.com
5. https://github.com/vmware-archive/kubeless
6. https://knative.dev/docs/
7. https://fnproject.io/
8. https://www.triggermesh.com/
9. https://nuclio.io/
10. Graphics Processing Unit
11. https://dapr.io/
12. Application Programming Interface

OpenFaas[13], **OpenWhisk**[14], and **Fission**[15] are serverless frameworks focusing mainly on simplifying the deployment of functions, scaling, and performance.

Azure Functions[16], **AWS Lambda**[17], **Google Cloud Functions**[18] and **IBM Cloud Functions**[19] are all paid services only allowing deployment to their cloud, which leads to vendor lock-in. Therefore they will be excluded from further tests.

Table 5.1: Summary overview of frameworks

framework	FaaS	pricing	documentation	maintained	required languages
Heroku	×	×	✓	✓	×
OpenFaas	✓	✓	✓	✓	✓
OpenWhisk	✓	✓	✓	✓	✓
Kubeless	✓	✓	✓	×	✓
Knative	×	✓	✓	✓	✓
AWS Lambda	✓	×	✓	✓	✓
Fn project	✓	✓	✓	✓	×
Triggermesh	✓	✓	✓	✓	✓
Nuclio	✓	✓	✓	✓	✓
Dapr	×	✓	✓	✓	✓
IBM Cloud Functions	✓	×	✓	✓	✓
Fission	✓	✓	✓	✓	✓
Google Cloud Functions	✓	×	✓	✓	✓
Azure Functions	✓	×	✓	✓	✓

13. https://www.openfaas.com/
14. https://openwhisk.apache.org/
15. https://fission.io/
16. https://azure.microsoft.com/en-us/services/functions/
17. https://aws.amazon.com/lambda/
18. https://cloud.google.com/functions/
19. https://cloud.ibm.com/functions

6 Detailed comparison of selected frameworks

To further compare frameworks that passed inclusion and exclusion criteria, several key features were examined for each framework to determine their potential usability at the Institute of Computer Science. Afterward, 3 frameworks were selected for implementation to compare quantitative metrics.

6.1 Overview of frameworks features

6.1.1 Autoscaling

Autoscaling is a crucial feature of serverless computing, allowing to assign resources dynamically. At times of high load, scaling up allows users a steady performance. During low load, functions can scale down to minimalize running costs.
Kubernetes offers three types of autoscaling:

1. Vertical Pod Autoscaler scales resources available for each pod
2. Horizontal Pod Autoscaler scales the number of pods running the function
3. Cluster Autoscaler scales clusters by adding nodes, typically when pods can not fully scale because the nodes can not handle the load or removing them when the nodes are not utilized

Two primary metrics can be utilized to determine when to scale the resources:

1. resource-based such as CPU or memory usage
2. workload based such as requests per second[1] or number of concurrent requests

Each framework can also implement custom metrics to determine the best time for scaling. Additional scaling can be provided by extensions such as KEDA[2], which can be installed alongside Kubernetes frameworks and provides event-based autoscaling.

1. also often called queries per second
2. https://keda.sh/

6.1.2 Triggers

Triggers are events that invoke a function and cause it to run. Frameworks standardly support triggers created by HTTP requests and schedule-based triggers, such as CronJob [36]. Many also support triggers based on a subscription to a queue, so-called message queues. Kubernetes itself supports a watch that monitors for changes in objects and triggers events accordingly [37].

OpenFaaS architecture contains a Gateway allowing invocation of multiple functions from one subject. Triggers can be passed to Gateway via Connectors that map queues from a messaging system to functions. In addition to HTTP and CronJob triggers, it supports custom cron-connector, which, similarly to CronJob, allows function invocation based on a schedule. Other Connectors include MQTT Connector or VMware Vcenter. A community-supported Connector for RabbitMQ also exists [36].

OpenWhisk triggers are managed by events, which also support calls from service APIs or Internet of Things devices.

Fission consists of multiple services running in containers. Among them, Executor and Router are used to handle triggers. Requests are handled by Router, which forwards them to pods, or in case pods are not running, Router requests them from Executor. Triggermesh supports KEDA by creating Keda ScaledObject, which creates a mapping between event sources (such as Redis or RabbitMQ) and the function deployment.

In addition to supporting the common triggers, Triggermesh focuses on integrating triggers from multiple proprietary cloud platforms. The sources include Google Cloud, AWS Lambda, or Slack.

Nuclio supports standard HTTP triggers and cron but is also the only framework from those investigated to support RabbitMQ out of the box.

6.1.3 Monitoring

When monitoring FaaS frameworks, there are generally two types of metrics to monitor. System metrics, focusing on overall system performance. User metrics monitor the condition of user functions and are the focus of this section.

OpenFaaS exposes multiple counters to monitor invocations and requests. It also provides histogram data of runtime and request duration. OpenFaaS also supports Prometheus[3], a query language for processing time-series data. Using Prometheus is also possible in Fission; however, it requires port forwarding of traffic [38]. A more straightforward way of monitoring functions is via Fissions OpenTracing, built on a Jaeger[4] backend. OpenWhisk contains multiple built-in functions for accessing metrics of running functions.

Table 6.1: Features

framework	autoscaling	triggers
OpenFaas	Yes Legacy scaling used for community version with single rule for all functions	HTTP, Cron, MQ
OpenWhisk	Yes	HTTP, Cron, MQ
Triggermesh	Yes In all editions via Knative Serving	HTTP, Cron, MQ
Fission	Yes Per function	HTTP, Cron, MQ
Nuclio	Only in enterprise version	HTTP, Cron, MQ

6.1.4 Popularity

Because the frameworks generally do not publish the number of their users, a number of stars on their GitHub projects was used. OpenFaas has the most stars, while Triggermesh has the least. The small number of stars of Triggermesh might be caused by Triggermesh being a closed source framework until a few months ago [39], with no GitHub page available before.

3. https://prometheus.io/
4. https://www.jaegertracing.io/

Table 6.2: Popularity on GitHub, ordered by a number of stars

framework	GitHub stars	GitHub contributors
OpenFaas	21.5k	163
Fission	6.9k	133
OpenWhisk	5.7k	193
Nuclio	4.4k	69
Triggermesh	289	16

6.2 Quantitative metrics

This section evaluates the performance metrics of Fission, OpenFaas, and Nuclio when deployed to a Kubernetes cluster. Fission and OpenFaas were chosen based on their features and their popularity in the developer community. Nuclio was chosen because it focuses on data science workflows, which might be interesting for the scientific community. The goal was to deploy the frameworks and test the performance under load and the autoscaling capabilities.

6.2.1 Setup

The metrics were gathered on a personal laptop with Ryzen 5 3005U CPU and 21,4 GB of available memory. The computer runs 64-bit Linux Mint 20.3 operating system with Linux kernel 5.13.0-40. Minikube was used to create a local Kubernetes cluster with one node for frameworks. The node has 4 CPU cores and 10,2 GB of RAM available. The versions of software and frameworks tested are described in Tab. [6.3].

Table 6.3: Software used

software	version
Kubernetes	1.23.6
Minikube	1.25.2
Fission	1.15.1
OpenFaaS	0.14.2
Nuclio	1.8.10

6.2.2 Test cases

A tool called JMeter[5] was used to create HTTP requests and to collect the metrics with 1, 5, 10, 25, 50, and 100 concurrent threads. To smooth out the initial load, a ramp-up period was set. Its length was chosen in every iteration to create ten threads per second (e.g., 5s ramp-up for 50 threads), except for iterations with 1 and 5 threads, where one second ramp-up period was chosen. Each thread sent ten consecutive HTTP requests to the application.

A simple Python function was chosen to compare frameworks, which would introduce minimal function overhead and reduce the number of dependencies. The function is invoked by an HTTP request, does not take any parameters, waits two seconds, and terminates. By doing so, framework autoscaling can be tested with many functions running simultaneously.

The median response time, 90th percentile, and 95th percentile response times were compared. The response time over the test elapsed time and throughput were also measured. Functions that would not respond after 60 seconds would return an error code 504, a time-out error. Thus an error rate was measured. No other error codes were encountered during the testing.

6.2.3 Scaling

Figures [6.1] and [6.2] show median response times, 90th percentile, and 95th percentile response times for Fission and OpenFaas, respectively. In both frameworks, median response times are close to the expected two seconds.

The open-source version of OpenFaas used does not support scaling down to zero. That is available only in the enterprise version. Therefore several pods are always ready, helping OpenFaas reach better response times than Fission.

Fission response times are similar to OpenFaas in cases up to 25 threads, when the 95th percentile response is received after 20 seconds. With 100 threads, the 95th percentile is 60 seconds because over 5% did not return a response.

5. https://jmeter.apache.org/

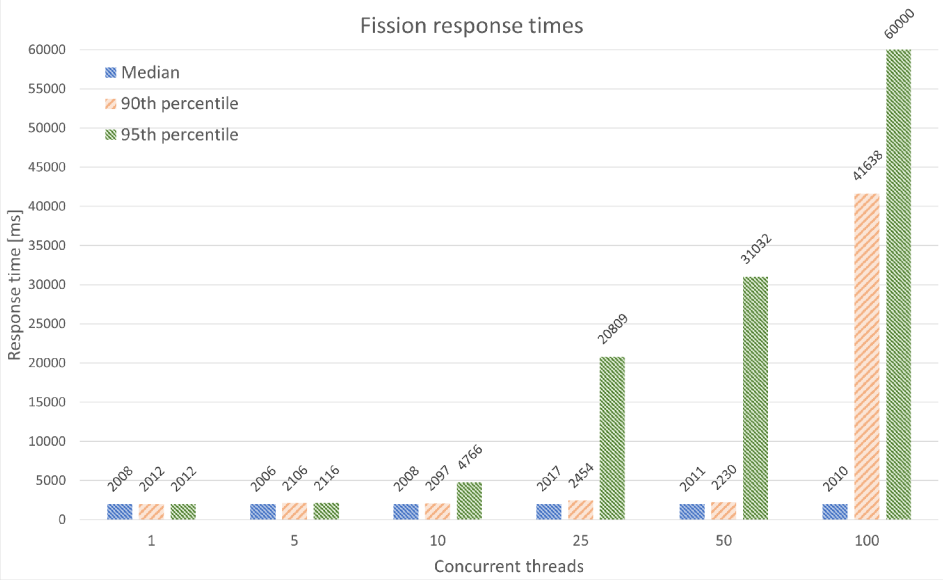


Figure 6.1: Median, 90th and 95th percentile of Fission response times

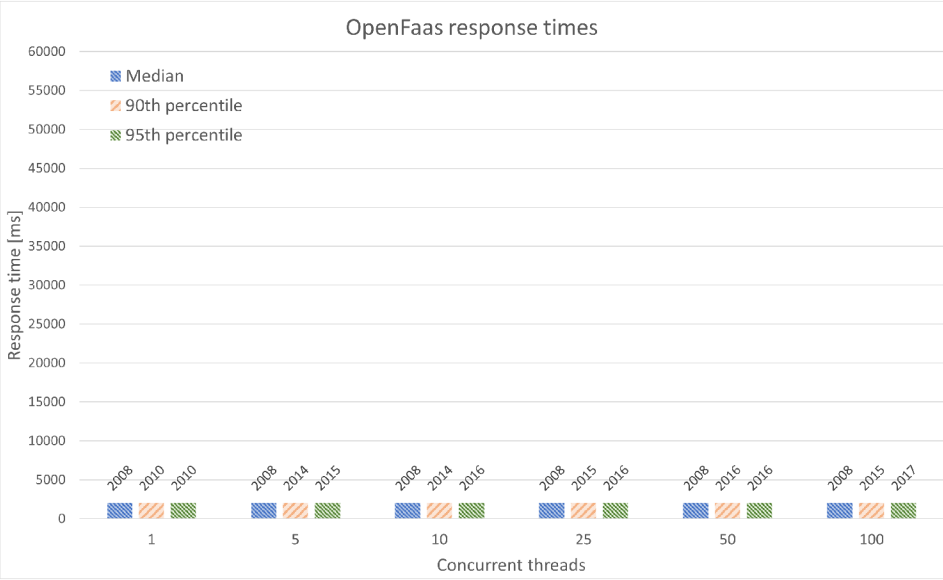


Figure 6.2: Median, 90th and 95th percentile of OpenFaas response times

The difference also shows in the request throughput. With one thread, Fission processed 29.8 requests per minute and OpenFaas 29.9. At 100 threads, Fission had a higher error rate, resulting in threads waiting for 60 seconds and thus a throughput of 6.9 requests per second. However, OpenFaas maintained high throughput at 33.4 requests per second.

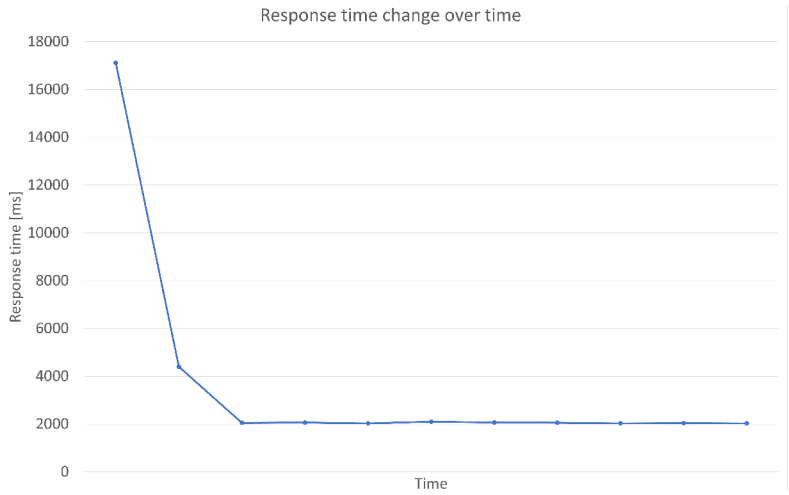


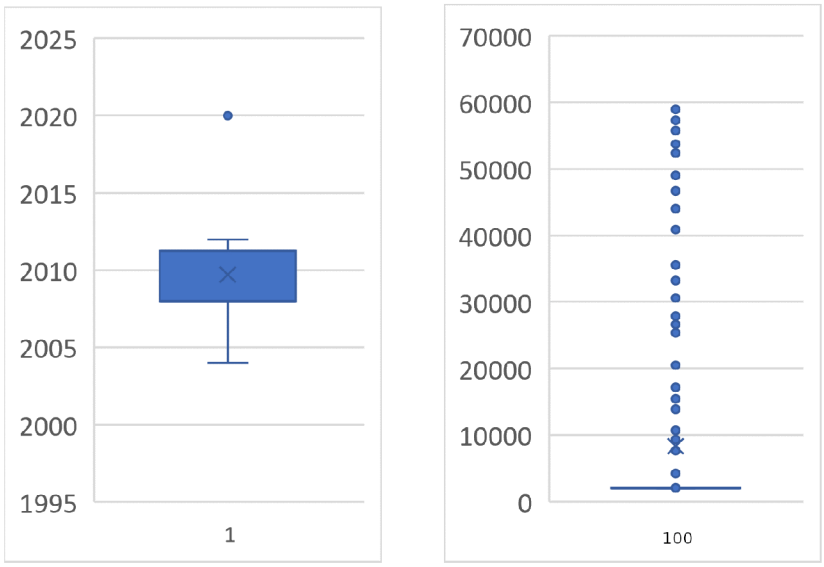
Figure 6.3: The change of Fission response times over time

Fig. [7.1] shows the trend of response time change over time for Fission functions. Response times are very long at the beginning of the test, which is caused by the need to scale up and initialize new pods. However, once enough pods are created, response times decrease close to the expected two seconds.

OpenFaas does not show a similar trend because the functions never scale to zero and thus do not need to initialize as many pods at the test start.

Fig. [6.4] shows response times for Fission functions with one thread and 100 threads, while fig. [6.5] shows the same for OpenFaas. OpenFaas performed relatively evenly with a small number of outliers which did not exceed the average by much.

On the other hand, Fission had many functions show latency many times higher than the average. These outliers also resulted in increased response times for the 90th and 95th percentile of requests.



(a) Response time, 1 thread **(b)** Response time, 100 threads

Figure 6.4: Fission response times in ms

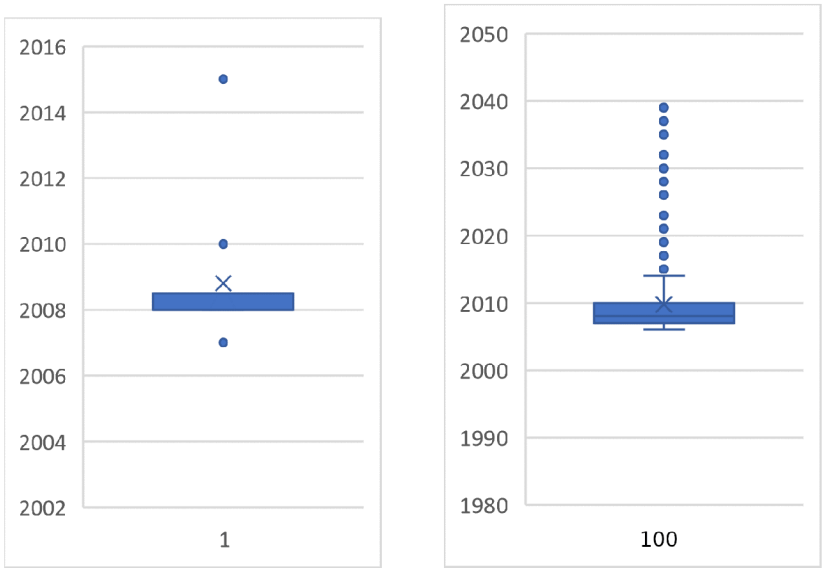
The outliers are approximately evenly distributed up to time-out, which suggests that if the time-out timer were set to longer than 60s, more functions would finish with high latency but without causing errors.

6.2.4 Error rate

Secondly, the error rate of calls was investigated. There are generally three types of failures serverless frameworks can generate. They are throughput limitations, increased latency, and time-out errors [40]. In the testing performed, if the response time exceeded 60 seconds, regardless of the cause, a time-out error was thrown and is thus the only type of error represented in the set.

Functions with increased latency that returned results before the time limit did not result in an error and are not included in this set.

Nuclio functions with one thread did not result in any errors. However, when testing Nuclio functions with more than one thread, all subsequent threads would time out. These errors are caused by the



(a) Response time, 1 thread **(b)** Response time, 100 threads

Figure 6.5: OpenFaas response times in ms

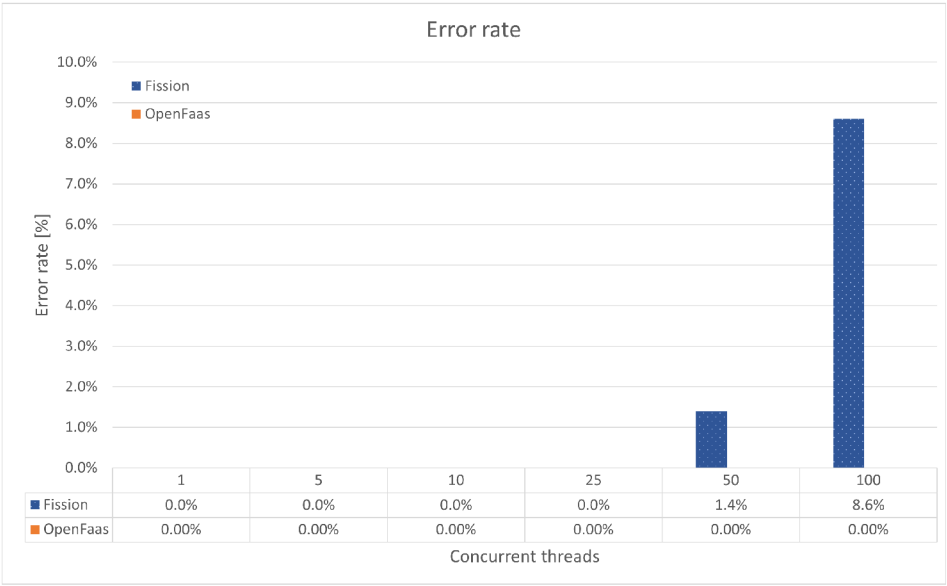


Figure 6.6: Error rate

open-source version of Nuclio not scaling up, which results in only one thread being processed at a time.

Fig. [6.6] shows the error rate of Fission and OpenFaas based on concurrent threads. OpenFaas functions did not cause errors in any of the test cases. This is partly attributed to a free version of OpenFaas not scaling down to zero and always keeping a certain number of pods warm, which helps at the beginning of the test.

Fission functions with less than 50 threads also did not result in any errors. However, 50 threads resulted in a 1.4% error rate, and 100 threads in an 8.6% error rate.

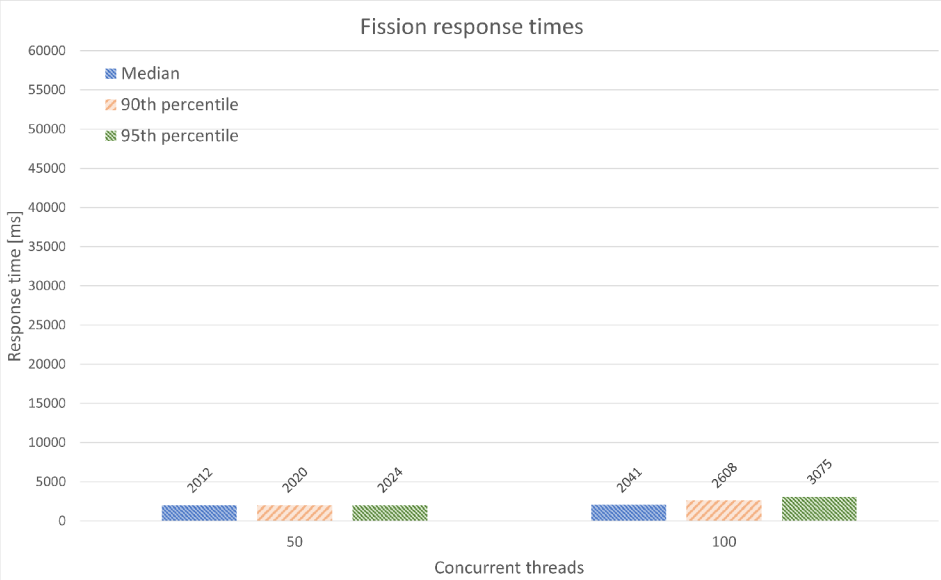


Figure 6.7: Fission response times with pre-warmed pods

6.2.5 Starting with pre-warmed pods

The last test performed was an ideal case scenario for Fission when all necessary pods were pre-warmed. The test was performed to give a more balanced comparison with OpenFaas, which always kept some pods warm, and to see what the impact of cold starts was. However, only test cases with 50 and 100 threads were performed, as tests with a smaller number of threads did not show much impact of cold starts.

Fig. [6.7] shows that the 90th and 95th percentile response times improved considerably with pods pre-warmed compared with fig. [6.1] and are now much closer to results achieved by OpenFaas. The error rate has decreased to 0% with 50 threads and 4.7% with 100 threads. Throughput has risen from 6.9 to 11.4 requests per second at 100 threads.

6.2.6 Conclusion

This section compared the performance metrics of Fission and OpenFaas, with a brief comparison of Nuclio. In single-thread tests, Nuclio performed slightly faster than other frameworks. However, multi-thread tests could not be performed because of the limitations of the open-source version. The one thread limit also means that the open-source version of Nuclio does not meet the requirements of this thesis.

The comparison of Fission and OpenFaas showed that OpenFaas achieved slightly better performance than Fission in all test cases. However, the performance difference grew with more concurrent users and was not fully mitigated by pre-warming all pods for Fission. Fission showed the impact of a cold start, but even with all pods running, a small number of requests resulted in increased latency to the point of timing out.

Even with some errors, the open-source version of Fission has demonstrated sufficient auto-scaling and significantly scaling to zero. Scaling makes Fission more fit for this thesis, as the open-source version of OpenFaas continues to use resources even when the function is not running.

7 Fission implementation

The implementation goal was to deploy and configure the framework properly and validate it by deploying a lightweight application that would work with Malleus Maleficarum[1] API.

To install Fission, specific prerequisites must be met. The required programs are kubectl and helm[2]. A Kubernetes cluster is also necessary. For the purposes of this thesis, Minikube was selected. The installation

Table 7.1: Minikube requirements [41]

CPU cores	memory	disk space	other
at least 2	at least 2GB	at least 20GB	Internet virtual machine support

process is described in the appendix of this thesis.

After successful deployment, Fission creates three namespaces. *Fission* and *fission-builder* which contains pods required by Fission. The third one *fission-function* contains user-created functions. Health of the

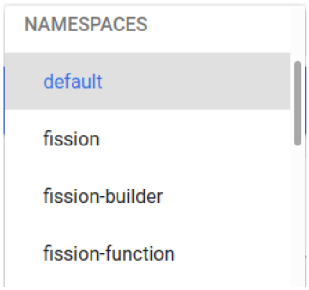


Figure 7.1: Namespaces created by Fission

framework can be verified by the status of its pods, shown in Fig.[7.2].

A Fission command-line interface (CLI) needs to be installed on client-side to control the framework. The CLI is used to create, manage, and update the environments, functions, and triggers (HTTP, message queue, and time). Creating a function is done by command:

1. Explained in more detail in 7.3.1
2. https://helm.sh/

```
      :~$ kubectl get pods -n fission
NAME                                     READY   STATUS      RESTARTS
buildermgr-774879664b-wpr7m              1/1     Running     46 (104m ago)
controller-6d455d7b75-cdrzs              1/1     Running     59 (102m ago)
executor-577ff745f-r7rcq                 1/1     Running     68 (102m ago)
fission-v1-15-1-fission-v1.15.1-251-d5rt8 0/1    Completed   0
kubewatcher-5dc7f4ccb5-hdpbg             1/1     Running     47 (102m ago)
mqtrigger-keda-79c5b694c7-l2rc8          1/1     Running     46 (102m ago)
router-579cb5c588-xtz2l                  1/1     Running     57 (104m ago)
storagesvc-c5f6d46f-vlpmj                1/1     Running     50 (104m ago)
timer-d98c67879-2s7pb                    1/1     Running     45 (104m ago)
```

Figure 7.2: Pod status

```
fission fn create --name <fn name> --code <source code> \
    --entrypoint <name.function name> --env <environment>
```

where *name* parameter is the name that Fission will use when referencing this function, *code* is the source code file. The *entrypoint* is the function in code which will be called, and *env* specifies the environment runtime. In production, users need a separate namespace for functions. The namespace can be specified using *–fnNamespace* flag.

Each function needs a trigger to invoke it, which can be created by:

```
fission ht create --name maintrigger --url \
/main --method POST --function main --createingress=true
```

ht specifies an HTTP trigger, the *name* is the trigger's name, *url* is the URL to send the requests to, and *function* is the name of an existing function that will be invoked. When creating names, they can not contain upper-case letters.

When deploying larger or more complex projects, deploying everything from the command line is not necessary. Fission supports a YAML file configuration called specs [42].

7.1 Pre-build environment limitations

To build and run functions, Fission uses environments. These are Docker[3] images downloaded from Fissions Docker Hub[4] and installed with a command.

```
fission env create --name python --image fission/python-env
```

3. https://www.docker.com/
4. https://hub.docker.com/

Each supported language has its environment, which facilitates the requests, logging, and dependencies. However, environments contain only a few of the most used libraries (for example, in the case of C#, it is only namespaces Microsoft.Win32 and System). To include other dependencies, it is necessary to rebuild the environment. That requires installing Docker on client-side for the rebuilding via *docker build* and creating an account on Docker Hub or another similar container repository to store the container. The source codes for environments, which can be extended, are accessible on the Fission GitHub repository[5].

7.2 Time-out flag

The default time-out of Fission functions is 60 seconds. Version 1.6 introduced a new way of setting the timer with flags *–fntimeout* or *–ft* [43]. However, when testing this setting with values greater than 60, the function would still time out after 60 seconds. If it were set to a value smaller than 60, it would time out correctly. This problem persisted with both *poolmanager* and *newdeploy* executors. Increasing other timers, such as *specializationtimeout*, did not help either.
A function can be tested before deployment and a test timer called by:
`fission fn test <fn name> --timeout=2m`
seems to be the only way of waiting for a response for more than 60s. However, this timer can not be set for functions deployed into production. The time-out flags have some effect, as the test timer can not exceed them. For example, if the *–ft* is set to 120 seconds and *–timeout* to 180 seconds, it will time out after 120 seconds. What the cause for these time-outs is, is unclear.

7.3 Ban expiration function

The function is based on the application MalleusMaleficarum.Tasks and serves demonstration purposes.

5. https://github.com/fission/environments

7.3.1 Malleus Maleficarum

Malleus Maleficarum is a service developed at the Institute of Computer Science. It is used by university staff to ban users of computer infrastructure if they break the rules. Moreover, the service is responsible for the whole life cycle of bans. This includes the Expiration function in MalleusMaleficarum.Tasks.

The function is run once a day and has a short runtime. As such, deploying it on a regular server would be cost-ineffective. Using a serverless approach is beneficial, as it also simplifies the deployment. For those reasons, the function is already deployed on Microsoft Azure. However, with the platform's deployment, the function can be migrated to it.

7.3.2 Example function

The example function has the same functionality as the Expiration function. It is invoked once a day and requests student bans. Those that have already expired but are still valid are terminated. An authentication certificate must be specified in the code to connect to the API, and firewall rules must be set to allow communication. Afterward, the function can be deployed to the cluster with *fission fn create*. A time trigger must also be created, which will invoke the function.

```
fission timer create --name malleusbans --function bans \
    --cron "0 0 12 * * ?"
```

The timing is determined by the cron string[6]. After the trigger is created, the following invocation time is shown.

```
$ fission timer create --name timetrigger1 --function timet1 --cron "0 0 12 * * ?"
trigger 'timetrigger1' created
Current Server Time:    2022-05-15T11:45:50Z
Next 1 invocation:      2022-05-15T12:00:00Z
```

Figure 7.3: Time trigger successfully created

6. https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm

8 Conclusion

This thesis aimed to explore the Function-as-a-Service frameworks and select one the Institute of Computer Science (ICS) would use to provide a serverless platform.

The first and second chapters introduced the problem solved and discussed the technologies used. Emphasis was put on describing Kubernetes, as it manages the cluster on e-INFRA infrastructure. The third chapter identified the target users for the platform. Common use cases were identified for each of the target user groups. In chapter four, the serverless technologies were described in detail. The history of the development was explained, together with its concepts and differences from standard deployment models. The chapter also contained a closer look at Functions-as-a-Service. The fifth chapter described finding frameworks and filtering them based on criteria formulated in this chapter. The selected frameworks were filtered down to five and described in greater detail in chapter six. From them, one was chosen for the platform in chapter seven. An example function, which runs on the platform and terminates expired bans, was created for demonstrational purposes.

This thesis offers a modern solution that can help the ICS expand its service offerings. Namely, ICS does not have to rely on the commercial cloud and can provide the academic community with a platform for computations that follows current development practices. The serverless platform and example function are ready to use and can be deployed to the institutes' Rancher.

Bibliography

1. *FaaS (Function as a Service)* [online] [visited on 2022-05-13]. Available from: https://www.intel.com/content/www/us/en/cloud-computing/faas.html.
2. *e-Infra* [online] [visited on 2022-04-17]. Available from: https://www.e-infra.cz/en/about-us.
3. *Kubernetes* [online] [visited on 2022-04-22]. Available from: https://www.ibm.com/cloud/learn/kubernetes.
4. POULTON, Nigel. *The Kubernetes Book*. 1st ed. Independently published, 2021. ISBN 979-8703756065.
5. BRENDAN BURNS, Joe Beda; HIGHTOWER, Kelsey. *Kubernetes: Up and Running*. 2nd ed. Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2019. ISBN 9781492046530.
6. *Self-Healing in Kubernetes* [online] [visited on 2022-05-10]. Available from: https://statehub.io/resources/articles/self-healing-in-kubernetes-what-about-the-data/.
7. *Containers* [online] [visited on 2022-04-23]. Available from: https://kubernetes.io/docs/concepts/containers/.
8. *Pods* [online] [visited on 2022-04-23]. Available from: https://kubernetes.io/docs/concepts/workloads/pods/.
9. *Nodes* [online] [visited on 2022-04-23]. Available from: https://kubernetes.io/docs/concepts/architecture/nodes/.
10. *What is a Kubernetes cluster?* [Online] [visited on 2022-04-23]. Available from: https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster.
11. *Kubelet* [online] [visited on 2022-04-25]. Available from: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.
12. *Kube-proxy* [online] [visited on 2022-04-25]. Available from: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/.

13. *Namespaces* [online] [visited on 2022-05-07]. Available from: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

14. *Command line tool (kubectl)* [online] [visited on 2022-05-13]. Available from: https://kubernetes.io/docs/reference/kubectl/.

15. *Kubernetes Components* [online] [visited on 2022-05-10]. Available from: https://kubernetes.io/docs/concepts/overview/components/.

16. *Why Rancher?* [Online] [visited on 2022-04-25]. Available from: https://rancher.com/why-rancher.

17. *Minikube* [online] [visited on 2022-05-10]. Available from: https://minikube.sigs.k8s.io/docs/.

18. EISMANN, Simon et al. A Review of Serverless Use Cases and their Characteristics. 2020.

19. *What is Serverless?* [Online] [visited on 2022-02-03]. Available from: https://serverless-stack.com/chapters/what-is-serverless.html.

20. *What is Serverless Computing?* [Online] [visited on 2022-02-03]. Available from: https://www.ibm.com/cloud/learn/serverless.

21. *What is SOA* [online] [visited on 2022-04-13]. Available from: https://collaboration.opengroup.org/projects/soa-book/pages.php?action=show&ggid=1314.

22. DUAN, Yucong et al. Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. 2015, pp. 621–628. Available from DOI: 10.1109/CLOUD.2015.88.

23. *The NIST Definition of Cloud Computing* [online] [visited on 2022-04-13]. Available from: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

24. *AWS Lambda Makes Serverless Applications A Reality* [online] [visited on 2022-04-16]. Available from: https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/.

25. *Serverless Computing Market - Growth, Trends, COVID-19 Impact, and Forecasts (2022 - 2027)* [online] [visited on 2022-04-17]. Available from: https://www.mordorintelligence.com/industry-reports/serverless-computing-market.

26. *Stateful vs stateless* [online] [visited on 2022-04-17]. Available from: https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless.

27. MISSBACH, Michael et al. *Stateless Computing: SAP on the Cloud*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. ISBN 978-3-662-47418-1.

28. PINTO, Duarte; DIAS, João Pedro; SERENO FERREIRA, Hugo. Dynamic Allocation of Serverless Functions in IoT Environments. In: *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*. 2018, pp. 1–8. Available from DOI: 10.1109/EUC.2018.00008.

29. EISMANN, Simon et al. Serverless Applications: Why, When, and How? *IEEE Software*. 2021, vol. 38, no. 1, pp. 32–39. Available from DOI: 10.1109/MS.2020.3023302.

30. LEITNER, Philipp; WITTERN, Erik; SPILLNER, Josef; HUMMER, Waldemar. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software*. 2019, vol. 149, pp. 340–359. ISSN 0164-1212. Available from DOI: https://doi.org/10.1016/j.jss.2018.12.013.

31. *What is Function-as-a-Service (FaaS)?* [Online] [visited on 2022-02-03]. Available from: https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas.

32. *Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP* [online] [visited on 2022-02-17]. Available from: https://mikhail.io/serverless/coldstarts/big3/.

33. XIE, Renchao et al. When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues. *IEEE Wireless Communications*. 2021, vol. 28, no. 5, pp. 126–133. Available from DOI: 10.1109/MWC.001.2000466.

34. *6 Serverless Frameworks on Kubernetes You Need to Know*. Available also from: https://www.appvia.io/blog/serverless-on-kubernetes.

35. *A (Very!) Quick Comparison of Kubernetes Serverless Frameworks*. Available also from: https://www.vshn.ch/en/blog/a-very-quick-comparison-of-kubernetes-serverless-frameworks/.

36. *Triggers* [online] [visited on 2022-05-07]. Available from: https://docs.openfaas.com/reference/triggers/.

37. *Efficient detection of changes* [online] [visited on 2022-05-07]. Available from: https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes.

38. *Using Fission's Prometheus Metrics* [online] [visited on 2022-05-14]. Available from: https://naughty-lumiere-b1b148.netlify.app/blog/using-fissions-prometheus-metrics/.

39. *Introducing TriggerMesh Open Source* [online] [visited on 2022-05-14]. Available from: https://www.triggermesh.com/blog/introducing-triggermesh-open-source.

40. *Why Serverless Apps Fail and How to Design Resilient Architecture* [online] [visited on 2022-05-10]. Available from: https://dashbird.io/blog/why-serverless-apps-fail/.

41. *minikube start* [online] [visited on 2022-05-15]. Available from: https://minikube.sigs.k8s.io/docs/start/.

42. *YAML Specs* [online] [visited on 2022-05-15]. Available from: https://fission.io/docs/usage/spec/.

43. *Configurable function-level timeout* [online] [visited on 2022-05-15]. Available from: https://fission.io/docs/releases/1.6.0/#configurable-function-level-timeout.

A Appendix

A.1 Source code

The source code of the example function is in the project archive.

A.2 Technology installation

When deploying Fission, the framework must be installed on a pre-existing Kubernetes cluster. The client must install Kubectl and Fission CLI to control functions on the cluster. In this thesis, deployment on a local machine was utilized, and Minikube was used to provide a Kubernetes cluster.

Installing Fission

The installation requires Helm and Kubectl.

- The official way of installing Helm differs per distribution and can be found in Helm documentation[1].
- The installation process for Kubectl is on Kubernetes documentation either for Windows[2] or Linux[3].
- The exact command to install Fission with Helm is in the official Fission documentation[4].

Installing Fission CLI

Getting the Fission CLI is performed by curl command or, on Windows, by downloading *fission.exe* file from the documentation.

---

1. https://helm.sh/docs/intro/install/
2. https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/
3. https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/
4. https://fission.io/docs/installation/

Minikube

The Minikube installation process for different operating systems is described in Minikube documentation[5].

The command to start the cluster is:

```
minikube start
```

A graphical overview is launched by:

```
minikube dashboard
```

The command to stop the cluster is:

```
minikube stop
```

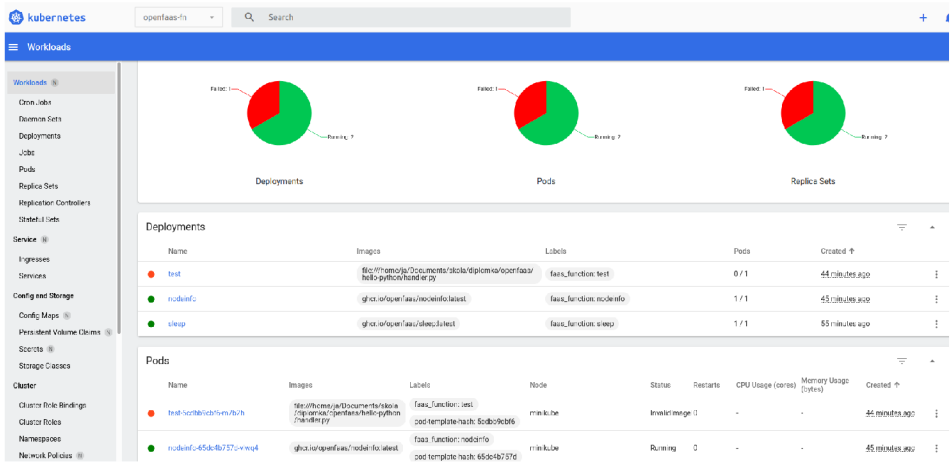


Figure A.1: An overview of Minikube dashboard

A.3 User development and deployment

When developing functions, it is usually necessary to process requests. In Python, Fission uses Flask[6] to handle requests. An example Python function might look like this:

```
from flask import request

def main():
```

5. https://k8s-docs.netlify.app/en/docs/tasks/tools/install-minikube/
6. https://pypi.org/project/Flask/

```
    req_body = request.data
    #other function code
```

In C# Fission uses custom dependency *Fission.DotNetCore.Api* to process requests. The *Execute* acts as the main function and gets invoked with the parameter *FissionContext*, which contains the request.

```
usingFission.DotNetCore.Api;
public class Example
{
 public string Execute(FissionContext context)
 {
  //code
 }
}
```

A.3.1 Setting up the environment

Fission uses environments that contain the necessary dependencies or tools to compile code to run user functions. Environments for supported languages are available on the Fissions Docker Hub repository, and more information, with a complete list, is available on a Fission website[7].

Python environment can be created by a command:
`fission env create --name <name> --image fission/python-env`
The flag name is the name of the created environment and can not contain upper-case letters. The flag image is the Docker Hub user name and container name. Other flags can be specified when creating an environment. The full list of nonmandatory flags is available in Fission documentation[8].
Whether the environment was created successfully can be verified by command:
`fission environment list`

7. https://environments.fission.io/
8. https://fission.io/docs/reference/fission-cli/fission_environment_create/

```
                $ fission environment list
NAME   IMAGE              BUILDER_IMAGE POOLSIZE MINCPU MAXCPU MINMEMORY MAXMEMORY EXTNET GRACETIME
dotnet fission/dotnet-env               3        0      0      0         0         false  0
nodejs fission/node-env                 3        0      0      0         0         false  0
python fission/python-env               3        0      0      0         0         false  60
```

Figure A.2: List of created environments

A.3.2 Creating and updating functions

Fission functions are created and deployed from the command line. To create a function, flags name (name that will be created for the function), code (source code file), entrypoint (function that will be called), and environment must be specified. A complete list of flags is in Fission documentation[9].

```
fission fn create --name <func_name> --code <soucre> \
--entrypoint <source.function> --env <env_name>
```

```
                                $ fission fn create --name prime --code prim.py --entrypoint prime.main --env python
Package 'prime-3a1ea7e8-4cdb-42e0-8b1d-1daf2e93c4c8' created
function 'prime' created
```

Figure A.3: Successfully created function

Updating a function is achieved via command:

```
fission fn update --name <func_name> --code <source>
```

The flags which will not be specified in update will not change and will have the values from before.

A.3.3 Triggers

HTTP

To invoke the functions, it is necessary to create triggers.
HTTP trigger is created by specifying the trigger name, URL, method, function's name, and whether to create ingress.

```
fission ht create --name <trigger name> --url  /<url> \
--method <method>  --function <function> --createingress=true
```

When using HTTP trigger with Minikube, it is necessary to allow ingress with command:

```
minikube addons enable ingress
```

Command to find the request URL on Minikube:

9. https://fission.io/docs/reference/fission-cli/fission_function_create/

```
$ fission ht create --name main --url  /main --method POST  --function main --createingress=true
 trigger 'main' created
```

Figure A.4: Successfully created trigger

`minikube ip`
The request URL is <minikube ip>/<trigger url>

Time

Invoking functions can be based on a schedule. The schedule is specified by cron string[10].

```
fission timer create --name <trigger name>  \
--function <fn name> --cron "<settings>"
```

```
$ fission timer create --name timetrigger1 --function timet1 --cron "0 0 12 * * ?"
trigger 'timetrigger1' created
Current Server Time:    2022-05-15T11:45:50Z
Next 1 invocation:      2022-05-15T12:00:00Z
```

Figure A.5: Time trigger

10. https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm