DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Data-Aware Function Scheduling on a Multi-Serverless Platform

Christopher Peter Smith

DEPARTMENT OF INFORMATICS

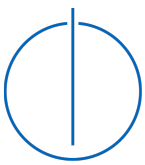
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Data-Aware Function Scheduling on a Multi-Serverless Platform

Datenabhängige Ablaufplanung für Function-as-a-Service-Plattformen

Author:	Christopher Peter Smith
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Anshul Jindal
Submission Date:	15.12.2021



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.12.2021 Christopher Peter Smith

Acknowledgments

Foremost, I want to extend my sincere gratitude to my supervisor, Prof. Dr. Michael Gerndt, and my advisor, Anshul Jindal, for giving me the opportunity to work on this thesis and providing me with support and guidance throughout the project.

My heartfelt thanks also go to my friends, who were by my side when I needed them, providing advice and wisdom to guide my work and distractions when I needed to clear my head.

Last but not least, I would like to thank my family. Their unceasing support throughout my studies carried me through student life's various struggles and got me to where I am today.

Abstract

Function-as-a-Service (FaaS) is an attractive cloud computing model that simplifies application development and deployment. However, current FaaS technologies do not consider data placement when scheduling tasks. With the growing demand for multi-cloud, multi-serverless applications, this flaw means serverless technologies are still ill-suited to latency-sensitive operations like media streaming.

This thesis proposes a solution by presenting **FaDO**, the *Function and Data Orchestrator*, which is a proof-of-concept application designed to allow data-aware function scheduling on a multi-serverless platform.

The application comprises a back-end server and API, along with a high-performance load balancer, a database, and a frontend browser client. These components allow users to interact with the application easily and seamlessly schedule functions onto the multi-serverless platform according to their data requirements.

FaDO further provides users with an abstraction of the platform's storage, allowing users to interact with data across different storage services through a unified interface. In addition, users can configure automatic and policy-aware granular data replications, causing the application to spread data across the platform while respecting location constraints.

The implementation thus enables users to distribute functions across a heterogeneous platform through data replication, balancing location constraints and performance requirements, and optimizing throughput using different load balancing policies.

The application fulfills its requirements, and load testing results show that it is capable of load balancing high-throughput workloads, placing tasks near their data without contributing any significant performance overhead. A qualitative evaluation of the system's design further indicates that FaDO has the ingredients necessary to make a reliable and performant network application.

Contents

1. Introduction

Cloud computing services give users access to powerful resources without the headaches that come with owning and managing complex infrastructure. Paired with attractive pricing models such as Pay-as-you-Go, these services have become very approachable and facilitate a multitude of use-cases, be it in industry or academia. Thus, cloud computing has reached the masses, and this is evidenced by the over one hundred billion dollars the market is worth [Syn].

It can be quite surprising then to remember that Google and Amazon only started using the term "cloud computing" in 2006 [Reg]. Indeed, cloud computing has evolved fast and continues to do so. As a result, providers now offer a plethora of different services that can be categorized into various models such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS).

A relatively new model to the market is Function-as-a-Service (FaaS). It belongs to the serverless execution model and first appeared on the market around the year 2010 [Rao]. It became popular after Amazon Web Services introduced it into their portfolio under the name AWS Lambda in 2014 [Amaa]. Since then, all major cloud providers have introduced FaaS into their service lineup, with Microsoft and IBM in 2016 [Kir; Bod], and Google in 2017 [Pol].

FaaS services provide users with facilities to easily deploy functions while keeping the complexities of the underlying infrastructure and mechanisms hidden from the user. These functions are modules of code designed to run discretely and are only executed on-demand when invoked through triggers prepared by the FaaS provider. Dynamic deployment, execution, and scaling tasks are all seamlessly automated in the background.

The typical pricing model for these services fits this on-demand paradigm by specifying costs-per-period consistent with the customer's timed and measured usage of the services in that period. In this way, customers do not pay an overall flat fee for a period and can expect costs to drop if their usage diminishes, and the inverse if it increases. For instance, AWS Lambda functions incur charges per milliseconds of execution time, per Gigabit-seconds of memory usage, and per one million accumulated function calls [Amac].

Before this model became popular, the best alternative in many scenarios was often to use services from the PaaS or IaaS models. However, while PaaS services share many

of the same advantages as FaaS services when it comes to the simplicity of use, they also share the same disadvantages as IaaS solutions when it comes to an application's idle time.

Indeed, while using PaaS or IaaS solutions, applications are constantly running and incurring charges, even when they are sitting idle. This is inefficient for both the user and the cloud provider, as the user is being charged for idle time, and the provider is unable to allocate idle resources to other tasks. The FaaS model allows for much more efficient use of computing resources and is, in many cases, a lot simpler and more cost-effective for the end-user.

1.1. Motivation

The attractive pricing model and simplicity boasted by FaaS solutions make the technology very desirable for end-users who want to focus their energy on developing their applications rather than on convoluted and time-wasting environment setups and deployments. And indeed, with the modern approach to application development that emphasizes modularity and microservice architectures, many end-users have adapted their systems to the FaaS model.

However, the increased simplicity of FaaS solutions comes at a cost. The end-user is afforded much less control over their code and how it is run. This thesis is specifically concerned with control over function placement with regard to data locality.

Indeed, at the time of this writing, typical FaaS solutions do not afford end-users much control over where a function is executed. This becomes a problem when a function requires data that is not proximal to its execution location, as it will cause data transfers and a significant idle period while the function waits for the data transfer to finish.

Disregarding data locality when scheduling functions thus causes increased response times and inefficient network traffic, incurring more costs and potentially crippling service-level objectives (SLOs). Latency-sensitive tasks, such as media streaming or complex distributed machine learning calculations, are thus not well suited to the current FaaS model [SP19].

Moreover, with the growing market and popularity of the Internet-of-Things (IoT), an increasing number of large applications are run on heterogeneous and distributed computing clusters. These platforms combine computing resources in different locations and with varying constraints, making the control over function placement, and for that matter also data placement, a very desirable feature.

1.2. Contributions

This thesis aims to implement a proof-of-concept application that dynamically places both functions and data on a heterogeneous platform composed of multiple clusters of colocated FaaS and storage services. The application is also to provide a unified interface to the various storage services on the platform, allowing users to upload and download data objects and organize them into storage buckets.

Towards this, we develop the **Function and Data Orchestrator** (FaDO). The key contributions of this work are:

- Transparent and data-dependent function placement through header-based HTTP reverse proxying and load balancing
- Configurable and automated granular data replication
- Unified access to heterogeneous storage services
- User-friendly management through an intuitive frontend client

FaDO's replication mechanisms build on the *storage bucket* paradigm ubiquitous in cloud computing storage services [Amad; Goo; IBM], where a storage bucket is a basic container for data objects. FaDO uses these buckets as units of replication and recognizes them as either master copies or replicas of a master copy. This thesis will refer to these as master buckets and replica buckets, respectively.

Users can then organize their data objects into storage buckets and set bucket-specific replication policies to influence how FaDO distributes the data across the platform. The application monitors the different storage buckets and watches for changes to their replication policies in order to keep all replica sets up-to-date and consistent with their policies. This includes creating new replica buckets or deleting unneeded ones, and sending new data to out-of-sync replica buckets.

Users of the application can query FaDO to find which objects are in which buckets and, with this information, can send function invocations to the application. The invocations are HTTP requests, with an added header that specifies the requested bucket for the function. Since FaDO knows where a bucket and its replicas are stored, it can reverse proxy and load balance invocations to the correct set of FaaS endpoints.

Figure 1.1 shows a client invoking a function requiring the storage bucket named *images*. When FaDO receives the invocation, it looks for the *X-FaDO-Bucket* header and uses its value to select the appropriate subset of FaaS endpoints. And so, in this case, the application load balances the client's function invocations to the FaaS services at *https://faas-1.fado* and *https://faas-2.fado*.

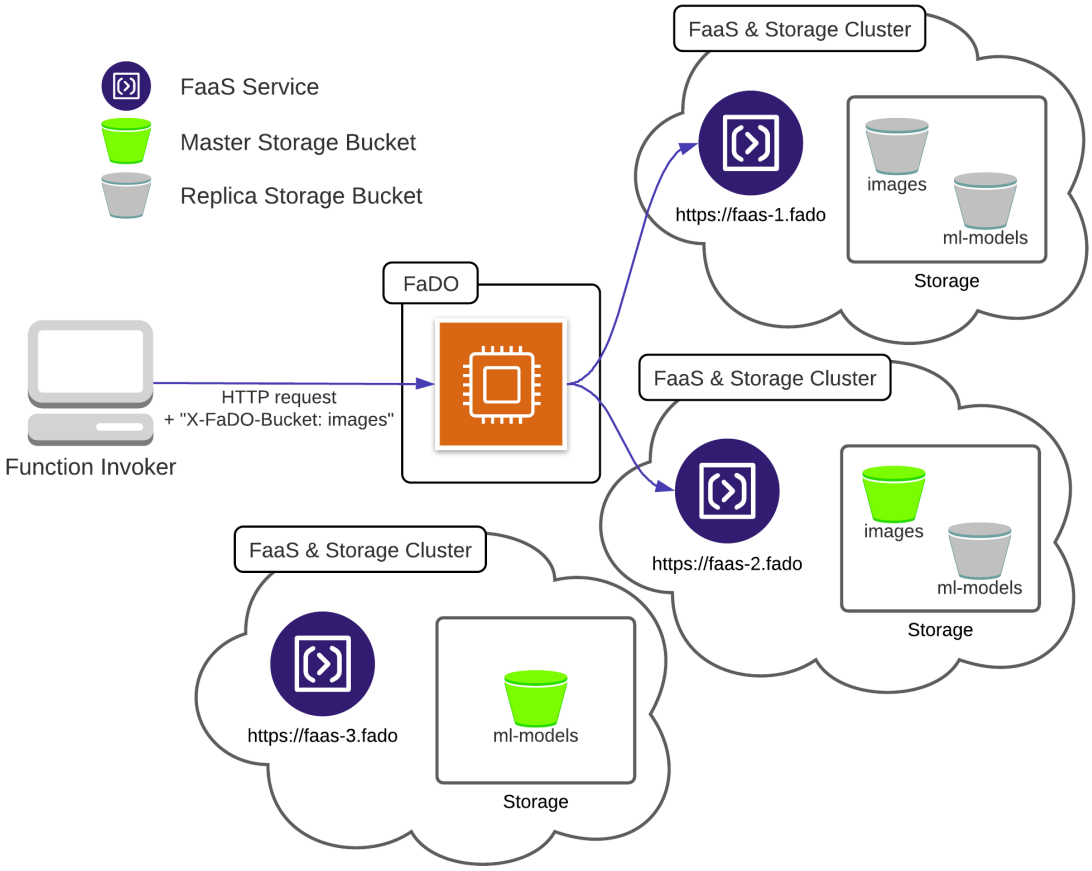


Figure 1.1.: Using FaDO to place functions close to the *images* storage bucket.

1.3. Thesis Organization

This thesis first covers related academic work, including the current body of work from the Chair of Computer Architecture and Parallel Systems here at the Technical University of Munich. This will provide a more detailed context for this work and help paint the greater picture in which FaDO fits in.

Following this, chapter 3 discusses the methodology behind this project, detailing the setup and technologies that were necessary for development and evaluation, and showing how the construction of FaDO was achieved through the combination of four components which include a database, a load balancer, a backend server, and a frontend client.

Chapter 4 then delivers a thorough breakdown of the database design, first going

over the data model and the different conceptual entities that are dealt with within the application, and then going over more specific aspects of the database's mechanisms that FaDO leverages to facilitate its operations on the data.

Chapter 5 lays out the implementation work that went into creating the FaDO backend server, which is the orchestrator behind all of FaDO's operations. This includes descriptions of the various mechanisms the server has built-in to interact with the database and load balancer, and remote storage and FaaS services. It will also explain the server's internal logic, its organization, and its functionalities are exposed through an API over HTTP.

Chapter 6 introduces the FaDO frontend client, a single-page Application web-application written in JavaScript. The chapter gives a tour of the frontend client's different pages and features, explaining how it is structured and how it enables users to interact with the FaDO application and its data intuitively.

Chapter 7 assesses FaDO, examining the viability of its design with regards to the requirements imposed upon critical network applications and discussing the implications of performance results obtained by load testing the application's load balancing and replication features under various scenarios.

The results show that FaDO is a successful proof-of-concept application that delivers significant performance increases for data-sensitive tasks run on FaaS technologies in heterogeneous cloud platforms. However, the results and an assessment of the application also show that FaDO's current design presents some limitations regarding replication performance, adapting to different cloud technologies, and making multivariate metrics-based load balancing decisions.

The thesis will thus conclude with a discussion of possible avenues for future work and how FaDO's design can be adapted and improved to better fulfill its requirements.

2. Related Work

Chapter 1 introduced the context of this thesis and presented the work that took during this project. This chapter will explore the growing body of research concerned with heterogeneous and multi-cloud platforms and relate it to FaDO.

Amongst this research are solution proposals that seek to optimize applications and workflows that deal with different cloud providers and resources with contrasting properties. While some of these topics are merely adjacent to the one at hand, such as the SeaClouds proposal by Brogi et al. [Bro+14], others very much encapsulate it, such as the paper on the Function Delivery Network by Anshul et al. from the Chair of Computer Architecture and Parallel Systems here at the Technical University of Munich [Jin+21]. However, all of these works provide helpful insight into how FaDO should operate and how it can be further improved.

2.1. Policy-aware scheduling in IaaS or PaaS

Brogi et al. present SeaClouds, a software system that intends to act as a layer of abstraction on top of a heterogeneous multi-cloud platform. The software handles the different cloud provider interfaces and hides their particularities behind a unified API. This consistent interface facilitates the deployment of complex and modular applications onto resources of varying characteristics and from different cloud providers. Pairing this with technological and quality-of-service (QoS) policies, SeaClouds aims to orchestrate an application's modules according to their requirements and monitor them to ensure that requirements continue to be met. If the requirements change, or if they are no longer met, the system dynamically re-arranges the placement of the application modules, allowing users to take full advantage of the resources and prices offered by the different cloud providers.

SeaClouds and FaDO are both concerned with issues application developers face with heterogeneous cloud platforms, and their purposes can overlap when it comes to colocating logic and data. Indeed, it is foreseeable to define an application module's requirements in such a way that SeaCloud's Planner, as depicted in figure 2.1, would place it close to its required data.

However, SeaClouds proposes simplifying application module orchestration in the IaaS and PaaS models, whereas FaDO is purely concerned with applications built on

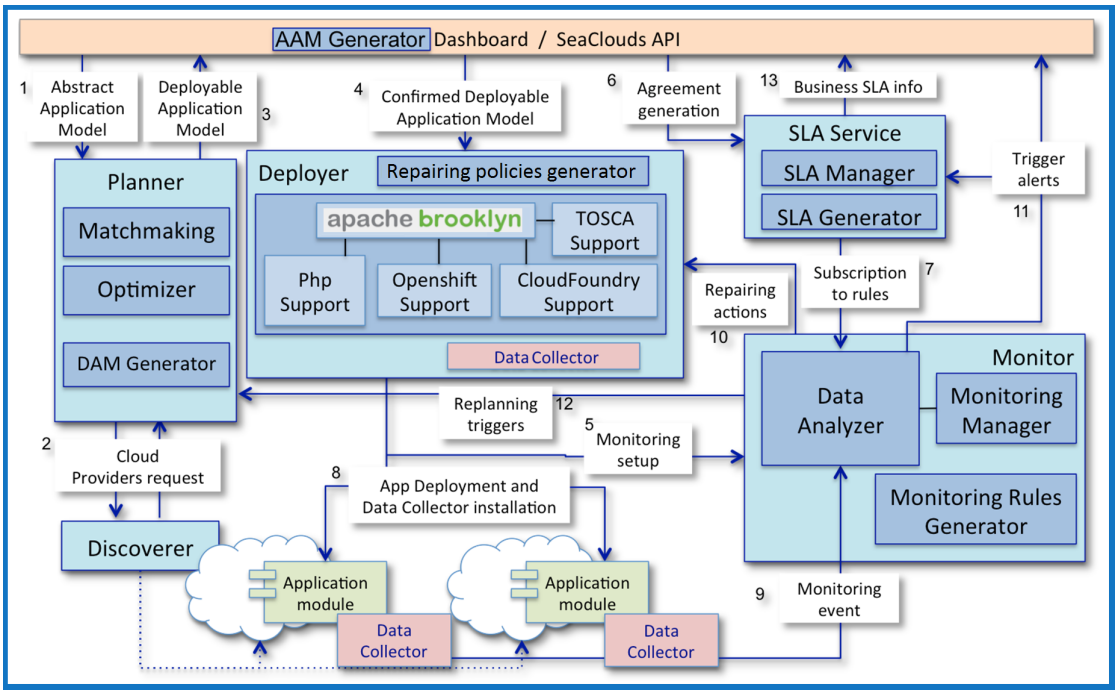


Figure 2.1.: System architecture from the SeaClouds proposal [Bro+14].

the FaaS model. Furthermore, at this time, FaDO is significantly more hands-off in its approach, as it does not take responsibility for function deployment. In fact, FaDO tries to be as transparent as possible when it comes to function placement and implements a load balancer to proxy interactions between the user and the FaaS services as seamlessly as possible.

Another distinguishing factor lies in FaDO's data placement and replication features, as it intends to add a layer of abstraction between the user and multiple storage services. This functionality is absent from SeaCloud's definition.

2.2. Data-aware scheduling in Serverless Computing

2.2.1. Multi-Cloud Serverless Platforms

Currently, FaDO cannot boast of being multi-cloud ready as it is built around a single storage technology: MinIO. However, there is great value in supporting multi-cloud serverless platforms, which is why Baarzi et al. propose a means to achieving this in their paper "On Merits and Viability of Multi-Cloud Serverless" [Baa+21]. This paper introduces the concept of a virtual serverless provider (VSP). This VSP is a provider

that allows customers to deploy serverless applications to different cloud providers through a consistent interface, hiding the differences from the users and helping them escape provider lock-in.

They argue that serverless applications are especially well suited to multi-cloud deployments and that VSPs would allow customers to optimize costs by always selecting the best offering for a given task. These mechanisms would take away a cloud provider's customer loyalty gained through application lock-in, where the costs to change providers outweigh the perceived benefits. In doing so, VSPs would force cloud providers to compete and innovate in order to maintain customer income.

The proposal also touches on the issue of data locality and mentions the importance of placing computations as close as possible to the data they access since this data can be costly to more or even illegal due to regulatory restrictions. As such, there is a certain kinship between the VSPs and FaDO, and there is much to gain in designing FaDO for multi-cloud deployment.

2.2.2. Function Delivery Network

Here at the Technical University of Munich, Jindal et al. proposed the Function Delivery Network (FDN), which is part of an overarching research project from the Chair of Computer Architecture and Parallel Systems.

The proposed FDN is a system that would receive configuration information for a heterogeneous distribution of FaaS and storage services and engineer the placement of both functions and data according to various metrics such as data requirements, function completion times, and hardware constraints. In addition, through monitoring, the FDN would track function placements and completion times to create behavioral models that help to optimize the scheduling process.

As such, the FDN and FaDO share very similar purposes, and indeed, the proposal for the FDN is the most significant inspiration that led to FaDO's development. Nevertheless, the FDN is a bigger and more complex application that outlines features that were not within the scope of this thesis. Notably, the FDN strives to optimize function and data placement through nuanced monitoring, behavioral modeling, and by taking into account hardware constraints.

However, some of FaDO's components are analogous to different elements of the FDN's architecture, as depicted in figure 2.2. For instance, the FaDO server and load balancer play similar roles to the FDN Control Plane, and FaDO's database can be related to the FDN's knowledge base.

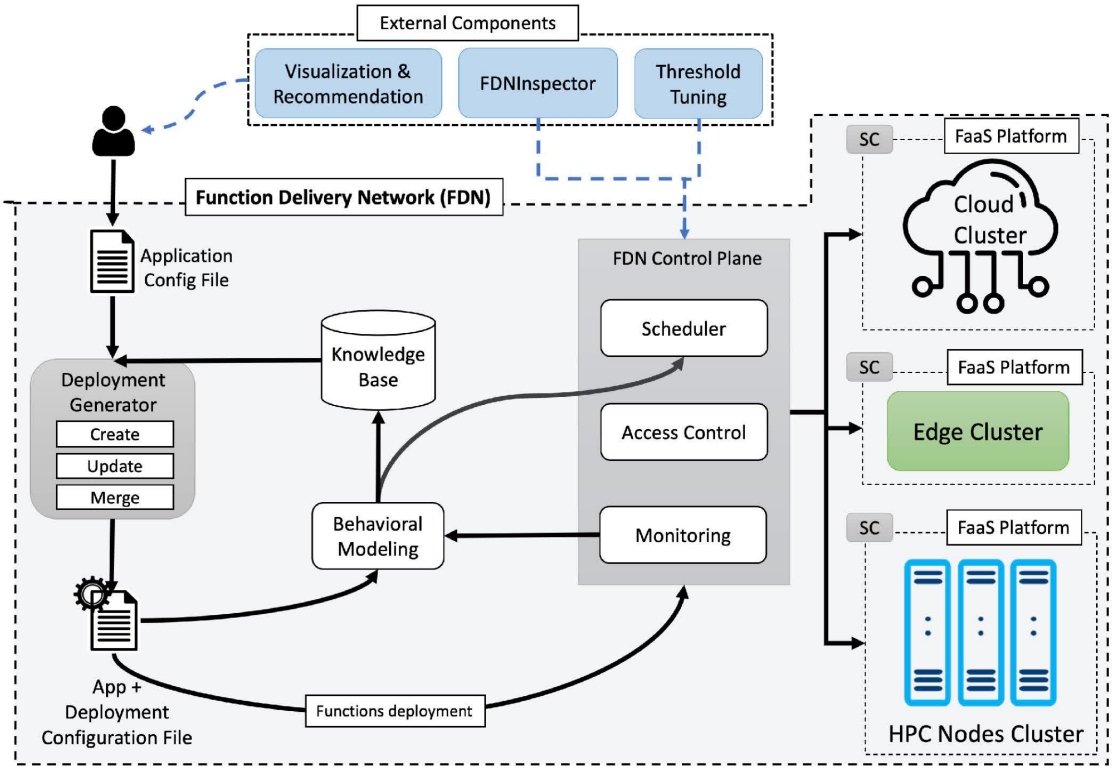


Figure 2.2.: The Function Delivery Network's architecture [Jin+21].

2.2.3. FaaST

Possani's master thesis on self-adaptive data management for heterogeneous FaaS platforms [Pos20] lays the groundwork for a project like FaDO, and so this thesis follows in its footsteps. In his work, he demonstrates that optimizing function placement according to data locality, and migrating missing data close to functions where needed, presented significant latency advantages in a heterogeneous FaaS platform such as the ones dealt with by FaDO. His thesis also explores latency-aware function placement, wherein the function completion times are compared over time and used to optimize function scheduling across the different FaaS targets.

This thesis will only concern itself with data-aware function placement and proposes an implementation where Possani proposed a strategy. However, aspects of the load-balancing mechanisms can be likened to latency-aware function scheduling, as the Caddy load-balancer can be configured to select upstream targets according to the `least\_conn`, or *Least Connections*, policy.

This policy causes the load-balancer to select the destination with the least active

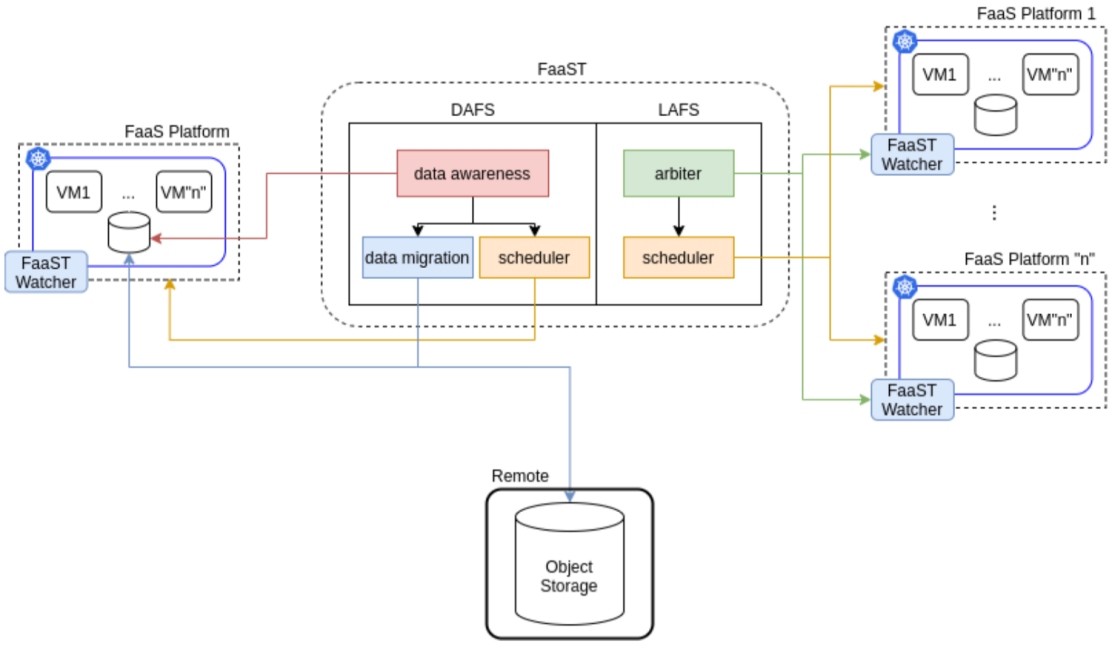


Figure 2.3.: The system architecture for FaaST [Pos20].

connections among a pool of upstreams. Hence, if functions run longer on one upstream, it will accrue more active connections and will not be picked as the upstream for a new function invocation. Functions will then more often be placed on the targets where they run faster.

This selection policy, of course, does not consider heterogeneity between the functions themselves and is not as fine-grained as a dedicated mechanism that would take more variables into account.

3. Methodology

Chapters 1 and 2 introduced FaDO, shedding light on its motivation and exploring the body of research that surrounds it. This chapter will explore the methodology that made FaDO possible, enumerating the requirements set for the project, explaining how the application is structured and what technologies it relies on, and describing the setup that made it possible to build it all.

A structured approach to the thesis was necessary to FaDO's successful development. Thus before jumping into the implementation process, it was essential to carefully specify the goals of this thesis in order to elaborate the project's core requirements. These requirements then led to an initial design which allowed for such decisions as what tools to use and what processes to develop. With this in place, it was possible to finalize the desired system architecture and specify a set of milestones to guide implementation efforts.

Implementation work then took up most of the duration of the project, and consisted in full-stack web application development, as it required creating and configuring development and productions environments and application deployment onto cloud resources, database design, backend server development and API consumption, and frontend client development.

Ultimately, the work resulted in a functional iteration of the FaDO application, which allowed for testing and performance evaluation. To this end, the application was benchmarked under different scenarios to understand FaDO's operating overhead and latency improvements.

3.1. Requirements

3.1.1. Functional Requirements

FaDO seeks to fulfill the following functional requirements:

- Maintain information about the different clusters of storage and FaaS deployments
- Locate the platforms' data objects
- Expose application data to clients

- Enforce user-defined constraints on data placement
- Forward function invocations to the appropriate FaaS deployments given the data requirements
- Dynamically automate the replication of partial data sets across clusters according to user-specified configurations and changes in the data

3.1.2. Non-Functional Requirements

Alongside the functional requirements defined above, FaDO also looks to fulfill the following set of non-functional requirements:

- Operate reliably and with high availability, as other systems will depend on the application
- Tolerate faults, especially from the various external services the application interacts with
- Accommodate scaling mechanisms
- Perform efficiently, adding as little latency as possible to function scheduling tasks

3.2. System Architecture

Following the philosophy of modular systems design and using the requirements established in section 3.1 as groundwork, we compose FaDO from the following parts:

- A database to store application state, user configurations, and information concerning the different resources in the platform
- A load balancer to terminate HTTPS traffic and reverse-proxy function invocations to the appropriate FaaS endpoints
- A backend server to implement all of FaDO's custom logic, exposing its functionality through an API and interacting with the database and load balancer, as well as communicating with and monitoring the platform's different storage deployments
- A frontend client to allow end-users to visually and intuitively interact with the system's data and affect changes through the backend server's API

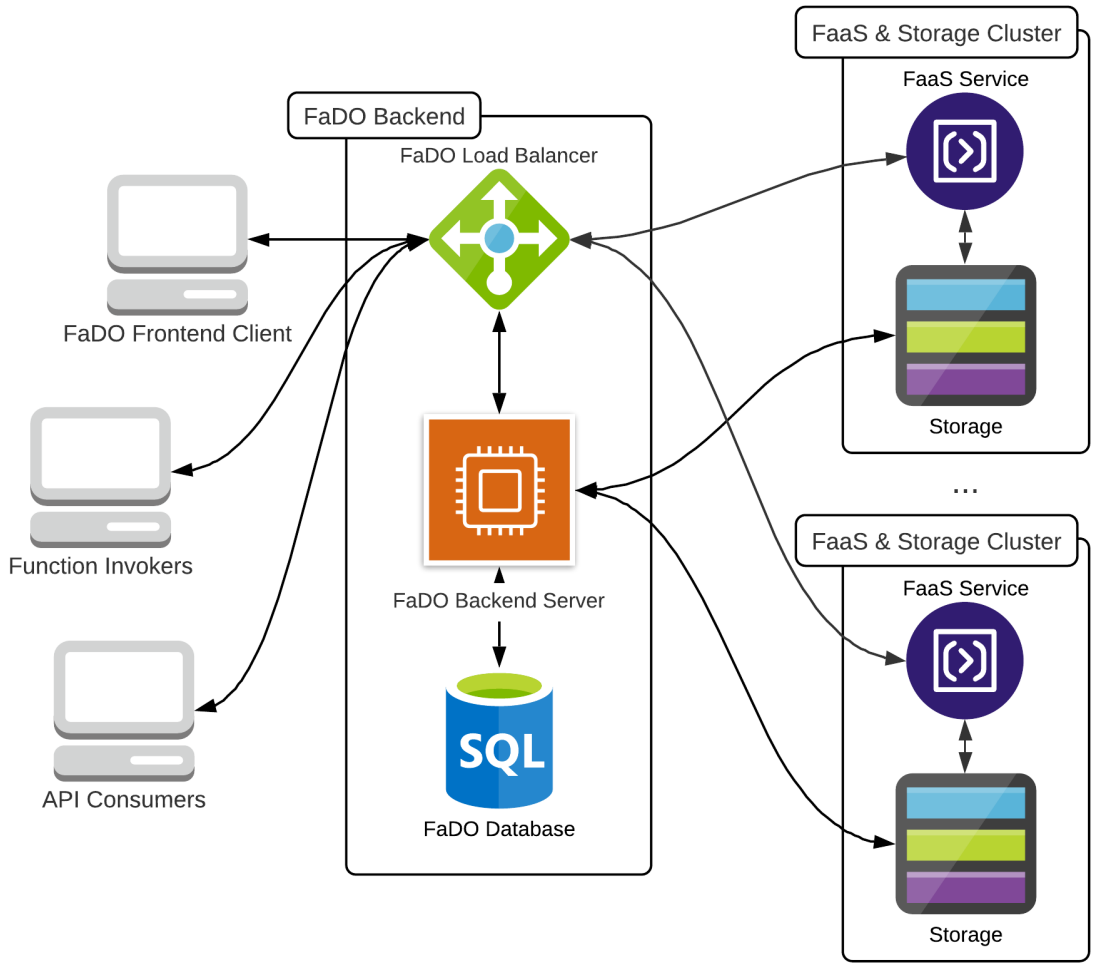


Figure 3.1.: FaDO's system architecture and its relationships with FaaS and Storage services in a heterogeneous cloud platform.

All tied together by the backend server, these four components make up the FaDO application and achieve the outlined requirements. While the database and load balancer can be run from existing technologies, the backend server and frontend client are, and must be, bespoke, as they implement FaDO's unique logic.

3.2.1. The Database

The database stores details concerning the different resources in the cloud platform, including FaaS service endpoints, storage deployments, the different storage buckets, and

their contained data objects. It also stores information on the different replication sets, detailing where the master buckets are located and where their replicas can be found. Furthermore, the database keeps current load balancing information configurations as well as user-defined policies.

Implementing this functionality from scratch would not be sensible, as multitudes of high-performance and highly reliable database technologies are available free of charge. Furthermore, most available options would be perfectly suited to FaDO's architecture. But, of course, only one can be selected, and the nature of FaDO's data must be considered to make an informed decision.

PostgreSQL

The data FaDO contends with is highly relational. Indeed, the application needs to track storage and FaaS deployment locations and relate them to data locations to configure the load balancer and send function invocations to the appropriate endpoints.

It was thus determined that a relational SQL Database Management System (DBMS) should be used over NoSQL alternatives such as MongoDB [Mone]. The final choice landed on PostgreSQL [Theb], a highly performant DBMS known for its reliability.

PostgreSQL is a well-documented and industry-standard open-source DBMS that has been under active development for the past three decades. Its track record is such that even the developers of MongoDB praise PostgreSQL for its performance when working with relational data [Monb].

This choice will ensure that the database can be maintained and extended long-term, allowing for fast and safe operations on the data. This makes it an ideal choice for the FaDO, as the application can expect reliable and fast operations on its data, increasing the application's overall robustness.

3.2.2. The Load Balancer

FaDO intends to provide transparent proxying of function invocations to achieve data-dependent function placement. At this time, the application only considers function invocations made through HTTP endpoint triggers. Hence, the application routes HTTP requests and expects data requirements to be passed through HTTP headers. The application then routes the request as-is to the selected upstream.

While forwarding function invocations is arguably the load balancer's primary task, it is also used as a general ingress for all of the application's traffic. Thus, the load balancer is responsible for the three following tasks:

- Load balancing function invocations to different FaaS upstreams based on information passed through HTTP headers

- Exposing the backend server's HTTP endpoint
- Terminating HTTPS traffic

As with the database, creating such a system from scratch would be prohibitively complex. This project thus outsources the load balancing functionality to a much more capable third-party solution. However, this system must allow for the above requirements while also allowing the backend server to configure it dynamically. Indeed, the load balancer cannot determine how to route function invocations independently, as this knowledge can only be derived from data stored in the database and must be interpreted by the backend server.

Caddy

Many options exist that can fulfill the application's load-balancing requirements, such as HAProxy and NGINX [Hap; F5]. Both are widely used, robust, and performant load balancers. However, Caddy Server [Staa] was preferred for this project.

While Caddy is a younger technology than the likes of HAProxy or NGINX, it remains a solid contender. It is written in Go, dynamically configurable through an administration endpoint using JSON objects, and provisions and maintains TLS certificates automatically.

This makes it an ideal choice for FaDO, as the backend server can easily configure it through its administration API. But not only that, the Caddy server can also be statically configured with the domain names used for both the backend server and the function invocations ingress, and it will seamlessly set up the endpoints for secured HTTPS traffic.

3.2.3. The Backend Server

FaDO's backend server is the most complex system developed for this project. Its complexity largely stems from its interactions with external systems, among which are:

- Storing data in, and retrieving it from, the database
- Creating the function invocation load balancing configuration and sending it to the load balancer
- Manipulating and monitoring the various storage deployments
- Automating storage bucket replications
- Accepting client requests to its API

To accomplish this reliably, the backend server must be put together with care using robust technologies. However, since the server's operations are complex, it is not always immediately apparent how they should be accomplished. Indeed, many quirks and caveats in the various interactions were only discovered after experimentation.

For this reason, the development of the backend server first consisted of a prototyping stage before transitioning to the development of FaDO's final iteration. NodeJS was the primary technology used to build prototypes, allowing for fast and flexible development, while the final server application was written in Go for robust performance.

NodeJS

NodeJS is a JavaScript runtime that was purpose-made to build network applications, and its large package ecosystem and Javascript's forgiving nature allowed for "quick-and-dirty" feature prototyping.

During the prototyping phase, the following packages were used in addition to NodeJS's built-in libraries:

- ExpressJS to expose the API via an HTTP server [Str]
- MinIO JavaScript SDK to interact with the MinIO storage clusters [Mini]
- Axios to send HTTP requests to such systems as the Caddy server [Sar]
- node-postgres for database operations [Car]

Go

Though NodeJS was a valuable tool for prototyping, the Go language was preferred when it came time to develop the FaDO server. Indeed, whereas Javascript is a scripting language with dynamic typing and NodeJS does not support concurrency, Go is a statically typed compiled language designed with highly reliable concurrent applications in mind. This made it a more attractive choice for building a performant, reliable, and maintainable application.

Alongside Go's standard library, the FaDO backend server is built using the following packages:

- Gorilla Mux for building an HTTP REST API [Too]
- MinIO Go client and admin SDKs for interacting with the MinIO storage clusters [Minh; Minc]
- pgx for database operations [Chr]

3.2.4. The Frontend Client

Though users can interface with FaDO through the backend server's API, it is a very unwieldy method that is not well suited to human-friendly and intuitive usage. Therefore, it was decided to build a client application that would allow users to interface with the system easily.

As FaDO is a network application, it was natural to build the client application as a frontend web application delivered by the backend server that runs inside a user's browser. This imposes some constraints on the application, as it must be built with the three foundational frontend web technologies: HTML, CSS, and JavaScript.

Frontend web development does, however, present a rich ecosystem of frameworks and libraries that ease the task of building complex browser applications.

ReactJS

The project uses the ReactJS framework, a powerful framework developed by Facebook that allows developers to build component-based web interfaces quickly [Fac]. The framework further allows for the use of JSX (JavaScript Syntax Extension), which allows for the programmatic representation of HTML within ReactJS JavaScript code.

On top of ReactJS and Javascript, the frontend client also uses the following packages:

- Axios for AJAX requests [Sar]
- Material-UI for UI components and styling [Mat]

3.3. Object Storage

FaDO's most significant external interactions take place with the storage services. Since it intends to operate on a heterogeneous cloud platform, it will ideally be agnostic to the different available storage technologies and allow for the usage of many different storage solutions.

However, this goal is not attainable before the conclusion of this work. Thus, this thesis will focus on the interaction with only one storage technology. And since FaDO is a proof-of-concept application aiming to demonstrate the viability of its function and data placement mechanisms, it was important to choose a storage technology representative enough of the solutions present in the market.

Furthermore, the storage solution must also allow granular data replication beyond reliably storing and fetching objects. This type of replication is not typical, so finding a

tool that would implement as many of the necessary features to achieve this out-of-the-box would be preferable, as it would reduce the server's responsibilities and decrease the number of potential points of failure for the proof-of-concept application.

Multiple storage options were then investigated, such as MongoDB with its file storage feature and GridFS support [Mone; Mona]. At first glance, it seems very fitting, as the technology supports both replication and sharding [Monc; Mond]. These features allow for automatic replication to secondary machines and data distribution across a heterogeneous platform. However, as implemented, these features do not allow the user enough control over how the data is distributed.

MinIO

MinIO was ultimately selected as the object storage solution. It was a known quantity, as previous work at the chair has already included it, such as Lucas Possani's master thesis [Pos20]. But more importantly, MinIO is an S3-compatible object storage technology that offers flexible bucket replication features [Mine; Mina]. Its compatibility with Amazon's widely popular S3 storage solution makes it representative of current cloud storage technologies, and its bucket replication features provide building blocks to implement FaDO's granular bucket replication mechanisms.

This thesis further leverages MinIO's `mc` command line tool [Ming; Minf] and its Javascript and Go language SDKs [Mini; Minh] to integrate the different MinIO deployments with FaDO.

3.4. FaaS Technologies

One of FaDO's most significant responsibilities is forwarding function invocations to different FaaS endpoints. Running FaaS services, while not directly necessary to FaDO's development and execution, is vital for testing and validating the application's operations. Many different FaaS technologies exist, such as those accessible through cloud providers like AWS's Lambda service [Amab] or open-source applications like OpenWhisk [Thea]. However, it was necessary to have fine control over the FaaS services and their execution environments for testing purposes.

OpenFaaS

For this reason, this thesis uses the FaaS resources already established during the course of Lucas Possani's thesis project and adjacent research at the chair. Hence, this thesis will rely on FaaS endpoints running the open-source technology OpenFaaS [Ope] that are already set up for testing.

3.5. Code Management and Application Deployment

Predictably, code management and versioning were an essential part of the development process. For this purpose, Git was used [Sof]. Git is a popular and powerful code versioning tool that allows many developers to manage and collaborate on a joint project.

Since the implementation work for this thesis did not involve more than one developer, Git's collaboration features were not leveraged. However, Git still allowed for effortless backup of the codebase and development from multiple locations.

In addition, Git was instrumental in deploying code to remote machines. And in doing so, it meshed very well with the other tool used for deployments: Docker [Docb].

Docker is a container technology that allows bundling an application's code with its dependencies. The Docker Engine can then run these containers consistently across different environments, allowing for predictable code execution, whether on a production server or the load development machine.

Most network applications have officially published Docker images, which are the blueprints that allow their execution as Docker containers. These images are handy, as they allow for straightforward application deployment and execution without the need for installing the software directly onto the machine. And so, this project relied on the Docker images released for PostgreSQL, Caddy, MinIO, and OpenFaaS [Doca; Docc; Docd; Docf].

The Docker command-line tool was invoked directly to start and stop individual Docker containers during development. However, Docker Compose was invoked instead on remote machines and for deployment purposes [Doce]. Docker Compose is a lightweight application released by Docker, Inc., used to orchestrate the execution of multiple Docker containers and provide facilities for them to communicate with each other.

3.6. Development and Deployment Environments

As FaDO is a rather complex network application, it was imperative to carefully set up its individual components and the environment it runs in to allow for local development and testing as well as deployment and execution on remote servers.

To achieve this, FaDO relies on environment variables and command-line options to dictate the behavior of the different systems that make up the application. These configurations are transmitted either on the command line when the system is invoked or through environment variables provided by the operating system and set by the user or Docker.

Furthermore, a VPN server was configured to expose the development environment to the internet and allow back-and-forth communications between FaDO and external MinIO deployments.

3.6.1. Environment Variables & Command Line Options

Environment variables are values that a program obtains from its execution environment. They are omnipresent in the background of our computer usage and allow programs to gain information specific to their environment that they can then use to modify their behavior.

On the other hand, command-line options are passed to a program as part of its invocation on a command-line prompt.

Both of these configuration methods are commonly used for network applications, and it is a vital mechanism that allows users to take powerful programs and quickly adapt them to their environment and run them without needing to modify them.

The systems that FaDO is built around, be it its internal components Caddy and PostgreSQL or external applications such as MinIO, all use these mechanisms to

```
func GetOptions() (listenHost string, listenPort string) {
    // Set variables from the environment.
    if listenHost = os.Getenv("LISTEN_HOST"); listenHost == "" {
        // If not present in the environment, set a default value.
        listenHost = "localhost"
    }
    if listenPort = os.Getenv("LISTEN_PORT"); listenPort == "" {
        listenPort = "8080"
    }

    // Set variables from command line options, specify fallback.
    flag.StringVar(&listenHost, "listen-host", listenHost,
        "The host the server listens for.")
    flag.StringVar(&listenPort, "listen-port", listenPort,
        "The port the server listens on.")

    flag.Parse()
    return
}
```

Listing 3.1.: Receiving input from the command line or the environment in Go.

configure their behavior [Stab; Thed; Minj]. These features thus make up part of FaDO's setup, and FaDO's backend server is designed to allow these configuration methods.

As these configuration methods are standard, programming languages will generally have built-in features that allow for these mechanisms. For example, in Go, it is straightforward to build these features using the `os` and `flag` packages, as is demonstrated in listing 3.1, while in NodeJS, the environment variables and command-line options can be directly accessed using the global variables `process.env` and `process.argv`, respectively.

Indeed, in listing 3.1 on lines 3 and 4, the `GetOptions` method attempts to retrieve the `LISTEN_HOST` and `LISTEN_PORT` environment variables. If these variables are not defined, the result will be an empty string. Thus, the program can check for this and set default values instead. On lines 12 and 14, the function will then look for values passed to the application through the command line. The third parameter to the `flag.StringVar()` function is the fallback value used if the option was not defined on the command line. In this manner, the program will configure its parameters, preferring command-line options to environment variables and setting hard-coded defaults if parameter values were not provided.

In this manner, FaDO is designed to condition its behavior based on parameters obtained from the environment or the command line. This dramatically increases the application's portability and simplified deployment tasks.

3.6.2. Standalone Container Services

The use of Docker containers was an integral part in building the local development and deployment environments, as it allows for bundling a program together with its dependencies and specifying various running parameters such as environment variables, TCP port mappings, and data volumes.

Docker containers were mainly run individually using the `docker run` command in the development environment. Convenience scripts were created to run the database, and the Caddy server, while the backend server was run natively to permit fast recompilation and restarts.

The contents of the `database/run-db.sh` script, as displayed in listing 3.2, serve to run the PostgreSQL database locally. The `--name` option specifies the container's name, and the `-p` option defines a port mapping. PostgreSQL's standard ingress port is TCP port `5432`, and here it is mapped to the local machines port `5454`. This is a convenient way to have multiple database instances running on the same machine without interference.

Listing 3.2's lines 4 and 5 define storage volumes. The first line defines a volume

```
1  docker run \
2    --name standalone-fado-db \
3    -p 5454:5432 \
4    -v pg_data:/var/lib/postgresql/data
5    -v /home/FaDO/schema:/docker-entrypoint-initdb.d \
6    -e POSTGRES_USER=fado \
7    -e POSTGRES_PASSWORD=password \
8    -e POSTGRES_DB=fado_db \
9    --rm \
10   postgres:13
```

Listing 3.2.: Running the containerized development database using Docker.

named `pg_data` that the Docker engine creates, containing the database's internal data, while the second line mounts the `/home/FaDO/schema` folder to the containers internal `/docker-entrypoint-initdb.d` folder. This is a special folder within PostgreSQL's docker container, which the database will use to initialize itself. Thus, the `/home/FaDO/schema` contains the database model definition files. The `--rm` command on line 9 causes the Docker engine to delete the container when the process is interrupted. This means that data is not persisted, and a fresh database is created each time the `run-db.sh` script is invoked.

The listing's lines 6, 7, and 8 define environment variables that tell the database what username and password to expect upon connection and what name it should use. Finally, the last line specifies the Docker image and version to use to create the Docker container. In this case, `postgres:13` designates PostgreSQL's official Docker image, and the project relies specifically on version 13 of the application.

3.6.3. Container Orchestration

When deploying the application to a remote server, where access is more constrained, running all components as standalone Docker containers using the `docker run` command would be too cumbersome and present difficulties. For instance, it would be incumbent on the user to ensure the services are monitored in case of failures and then restarted.

Docker Compose simplifies this significantly, as it runs on top of the Docker Engine and handles running and monitoring different container services according to a user-specified configuration.

Listing 3.3 shows a simple Docker Compose configuration that was used to run the FaDO backend remotely. The backend includes the FaDO server, the Caddy load

balancer, and the PostgreSQL database, and Docker Compose ensures that they are all executed properly.

The configuration defines three services named `database`, `caddy`, and `fado`, and defines different parameters for each. In the case of the database and the load balancer, the configuration species which Docker images and versions should be used to build the containers, while for the FaDO server, the configuration specifies a local folder from which to create the container.

The "`restart: always`" clauses specify that Docker Compose should always restart the services after a failure, and the `depends_on` enumeration on line 29 specifies that the FaDO server depends on the database and caddy to be running.

The `volumes`, `environment`, and `ports` enumerations correspond to the Docker options discussed in section 3.6.2.

With this file defined, it is very straightforward to control the orchestration using the `docker-compose` tool and perform tasks like starting and stopping the services or rebuilding containers.

```
1  version: "3.6"
2  services:
3    database:
4      image: postgres:13
5      restart: always
6      volumes:
7        - "db_data:/var/lib/postgresql/data"
8        - "/home/smithc/thesis/FaDO/database/schema:/docker-entrypoint-initdb.d"
9      environment:
10       POSTGRES_USER: "fado"
11       POSTGRES_PASSWORD: "password"
12       POSTGRES_DB: "fado_db"
13     ports:
14       - "5432:5432"
15   caddy:
16     image: caddy:2.4.5-alpine
17     restart: always
18     ports:
19       - 80:80
20       - 443:443
21       - 2019:2019
22     volumes:
23       - /home/smithc/thesis/FaDO/caddy/prod/data:/data
```

```
      - /home/smithc/thesis/FaDO/caddy/prod/config:/config
      - /home/smithc/thesis/FaDO/caddy/prod/Caddyfile:/etc/caddy/Caddyfile
  fado:
    build: ./backend
    restart: always
    depends_on:
      - database
      - caddy
    ports:
      - 9090:9090
    volumes:
      - /home/smithc/thesis/FaDO/backend/build:/app/build
    environment:
      FADO_DATABASE: postgres://fado:password@database:5432/fado_db
      FADO_SERVER_URL: https://server.fado
      FADO_LB_DOMAIN: lb.fado
      FADO_LB_PORT: "443"
      FADO_CADDY_ADMIN_URL: http://caddy:2019
volumes:
  db_data:
```

Listing 3.3.: Running the FaDO backend in a production environment with Docker Compose.

3.6.4. Exposing Local Services to the Internet

The most straightforward setup for development is to execute everything locally, including such services as the database and MinIO storage deployments. This allows for fast changes in the code by simply recompiling and relaunching a development build. However, it also comes with some downsides as it quickly became evident that remote production environments produced some unique circumstances that could not be replicated locally.

One of the problems that arose was the issue of timing when communicating with systems across the internet, such as distant MinIO deployments. In addition, the change in conditions helped reveal some weaknesses in FaDO's backend server code that would cause failures. It, therefore, became useful to develop the FaDO server alongside remotely deployed services, as the local environment was not sufficiently representative of FaDO's performance in production.

```
1 {
2   "srv0": { "listen": [":3000"], "routes": [ ... ] }
3 }
```

Listing 3.4.: Load balancing configuration in the Local environment.

```
1  {
2    "srv0": {
3      "listen": [":443"],
4      "routes": [
5        {
6          "handle": [{ "handler": "subroute", "routes": [ ... ] }],
7          "match": [{ "host": [ "server.fado.io" ] }],
8          "terminal": true
9        },
10       {
11         "handle": [{ "handler": "subroute", "routes": [ ... ] }],
12         "match": [{ "host": ["lb.fado.io"] }],
13         "terminal": true
14       }
15     ]
16   }
17 }
```

Listing 3.5.: Load balancing configuration in the production environment with two domain names on port 443.

Another issue arose with the Caddy load balancer and FaDO's interactions with it, as the load balancing configuration's JSON format changes shape between the local environment where the load balancer is receiving unencrypted HTTP traffic and the production environment where the load balancer is terminating secured HTTPS traffic.

Since FaDO's backend server manipulates the load balancer using its JSON configuration format, the change in the shape of the configuration between the two environments made it very difficult to develop the production-ready server locally. Listings 3.4 and 3.5 show the difference between Caddy's JSON configuration both in the development and production environments. The FaDO server must parse the configuration and insert updated route information on line 2 of listing 3.4 and line 11 of 3.5.

The routes defined on line 6 of listing 3.5 do not contribute to load balancing function

invocations but rather serve to expose the FaDO server's HTTP endpoints. This aspect of the configuration is pre-defined and must not be changed during FaDO operation.

Given these issues, it became desirable to expose the local development environment to the internet to execute the program under production circumstances while retaining the advantages of the local development environment, namely unfettered access to the local machine and fast compilation and restart times.

After some investigation, the use of a VPN server resolved to be the most practical solution. Therefore, Wireguard, an open-source VPN solution, was installed onto a remote virtual machine to achieve this. The Wireguard server on the remote machine creates a virtual private network (VPN) and exposes the development environment to

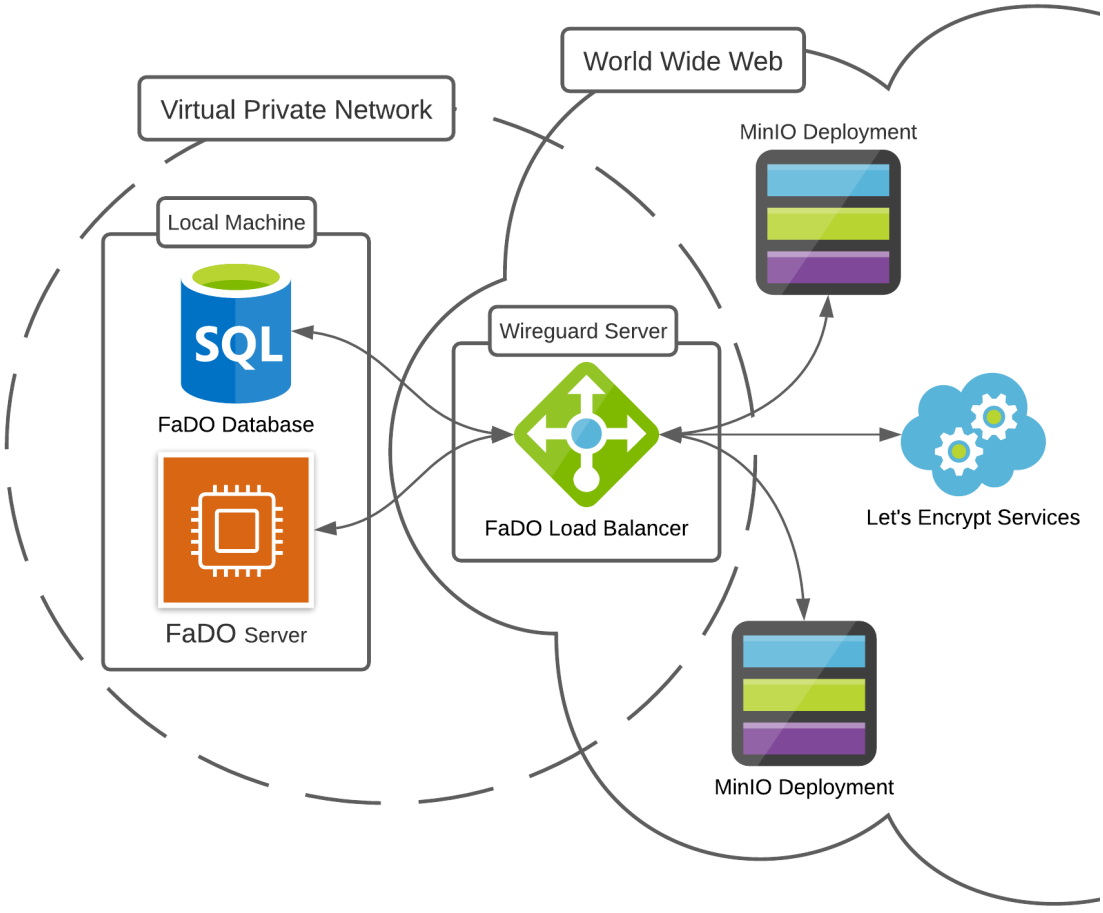


Figure 3.2.: Exposing the development environment to the internet using a Wireguard VPN server.

the internet using a Wireguard client installed on the local machine.

The Caddy load balancer is then moved to the Wireguard VM as illustrated in figure 3.2, where it is exposed to the internet while still being able to reach services running in the local environment. It can then request TLS certificates from Let's Encrypt and respond to challenges that confirm that the domain name included in the request does indeed point to the requesting machine. With the TLS certificates provisioned, the load balancer can then terminate HTTPS traffic and reverse proxy traffic back to the services running in the development environment.

3.7. Error Handling

Errors are an inevitable aspect of any software development endeavor, and there are multitudes of ways to deal with them. However, as the FaDO server is a complex web application, error handling should be done carefully to maximize reliability and facilitate bug fixing and maintenance.

Debuggers are a common tool used to root out problems in the code. Moreover, they are a superior tool to simple logging solutions. This is especially the case in large software projects with many collaborators. However, this was not such a project, as there was only one software developer writing the code. Because of this, a simple logging solution was devised instead of investing time in setting up a more powerful but less flexible solution.

This ended up being a much more practical solution, as when errors occurred, it was never complicated to understand the cause, and fixing them only required knowing where exactly they occurred in the code.

Go has a very straightforward error handling model. A function can return an error variable. If the value is `nil`, the function returned successfully, but if it is not `nil`, an error occurred, and the variable provided will give further details.

One shortcoming is that these errors can often be quite vague and do not indicate

```
2021/08/01 18:06:02 Error: exit status 1
    /app/mc/mc.go:23
    /app/mc/mc.go:67
    /app/mutations/buckets.go:181
    /app/mutations/storageDeployments.go:58
    /app/main.go:47
```

Listing 3.6.: Adding context to vague error logs.

where the error stems from. Without this information, it can be challenging to gain insight from logs, and it is thus necessary to attach contextual information when an error is printed to the logs. This can be done on a case-by-case basis, but this is a time-consuming practice that can be optimized.

Thus, some code was written to facilitate error logging and automatically attaching

```
type ServerError struct {
        Lines []string
}

func (se *ServerError) Error() string {
        return fmt.Sprintf("Error: %v", strings.Join(se.Lines, "\n          "))
}

func ProcessErr(err error, skips ...int) error {
        if err == nil { return nil }

        s := 1
        if skips != nil && len(skips) > 0 { s = skips[0] }
        _, file, line, _ := runtime.Caller(s)
        msg := fmt.Sprintf("%v:%v", file, line)
        switch v := err.(type) {
        case *ServerError:
                v.Lines = append(v.Lines, msg)
                return v
        default:
                return &ServerError{Lines: []string{err.Error(), msg}}
        }
}

func PrintErr(err error) error {
        if err = ProcessErr(err, 2); err != nil { log.Println(err.Error()) }
        return err
}
```

Listing 3.7.: Processing errors in Go leveraging interface mechanics and the runtime library.

contextual information to the output. The output generated by this code looks like the log in listing 3.6, where the original message would have only been "exit status 1". The code adds date and time information and source file locations where the error was encountered. Using this and going through the code, one can quickly find that the error occurred while trying to invoke the `mc` command-line tool and that the invocation failed.

Listing 3.7 shows the code that achieves this. It leverages Go's interface feature, where Go's `error` is simply an interface for a type that can describe itself with a function named `Error()` that returns a string. The code then defines a custom error type named `ServerError` compatible with the interface, which can be returned as an error.

The `ServerError` type is a simple Go `struct` with an array of strings named `Lines`. The `Error()` function it receives joins these lines together with newline characters.

Errors objects of type `ServerError` are created or processed by the `ProcessErr()` method. This function is used before an error is returned and will detect whether the given error is of type `ServerError` or not. If it is not, it will initialize a `ServerError` with the error message and the location, and if it is, it will add a line giving further location information.

The `ProcessErr()` function uses Go's `runtime` package to log the file and line number where the error was handled. To be noted, the `runtime.Caller()` function takes a *skip* value specifying which function call is referred to by the file and line numbers. If the *skip* value is 0, the file and line number correspond precisely to where the `runtime.Caller()` function was invoked. Passing a *skip* value of 1 to the function call on line 14 causes `runtime.Caller` to refer to where the `ProcessErr` function was invoked. And when `PrintErr()` passes a value of 2 to the function, it ensures that the coordinates it receives back point to where the `PrintErr()` function was invoked instead.

3.8. Evaluation

Evaluating FaDO began once the goals of the thesis had been reached and the application was sufficiently robust. Three core questions were then formulated to guide the testing and determine FaDO's performance:

- Is the design viable for fault-tolerant and reliable operation?
- Is the replication performance suitable for production scenarios?
- Is FaDO's data-sensitive scheduling robust and fast enough for demanding workloads?

This thesis will answer these questions by closely examining how FaDO is structured and what it depends on to maintain the correct state and benchmarking its replication performance and function completion times.

3.8.1. Replication Performance

FaDO's replication mechanisms are built on top of MinIO's *mc* command-line tool using the `mc mirror` directive. Thus, we will closely examine the tool through its documentation and behavior and test its performance by recording replication completion times under different scenarios.

The tests will seek to answer the following questions:

- How does mc's performance vary with different amounts of data?
- How do network conditions affect *mc*'s mirror performance?
- How does MinIO handle data mirroring between two buckets that are only partially out-of-sync?

3.8.2. Load Testing Function Invocations

To confirm FaDO's performance for data-sensitive function scheduling, measuring and recording function completion times under various scenarios is necessary. To this end, we will use the open-source tool *k6*, which is purpose-built for load-testing network applications [Gra].

Using this tool, we can apply varying loads to the load balancer and assess FaDO's overhead and its performance when placing functions according to their data and according to different scheduling policies. Broadly speaking, the tests will record performance using three load testing scenarios:

- Sending function invocations directly to a single FaaS endpoint.
- Sending function invocations to a single FaaS endpoint through FaDO.
- Send function invocations to multiple FaaS endpoints using FaDO's load balancing policies.

These measurements will provide insight into FaDO's overall performance, overhead, and the impact of its different load balancing policies.

4. Database Design

The first chapters introduced FaDO and the context surrounding it. Then, chapter 3 detailed the project's methodology and gave a thorough breakdown of the different components necessary to build the application.

This chapter will focus on FaDO's PostgreSQL data, describing the data it stores, what that data looks like, and how the application uses the DBMSs facilities to operate on the data effectively.

The first section will go over the data model, explaining the different conceptual items that FaDO uses and how these relate to real-world entities and values. The later sections describe how schema constraints connect related information and ensure data correctness and how views simplify complex data accesses.

4.1. Data Model

As shown in figure 4.1, the data model formalizes the conceptual entities that FaDO interacts with through a database schema. While this section will content itself with a summary of the schema's entities, an exhaustive definition of the schema is listed in appendix A. The entities this section summarizes are the following:

- Clusters
- FaaS deployments
- Storage deployments
- Master Buckets
- Replica Buckets
- Objects
- Policies

While some of these entities are easily recognizable as real *things*, such as a storage deployment, others are more abstract in nature, like a policy. The database schema uses tables and relationships to give concrete definitions to these entities and allows the application to handle and manipulate them.

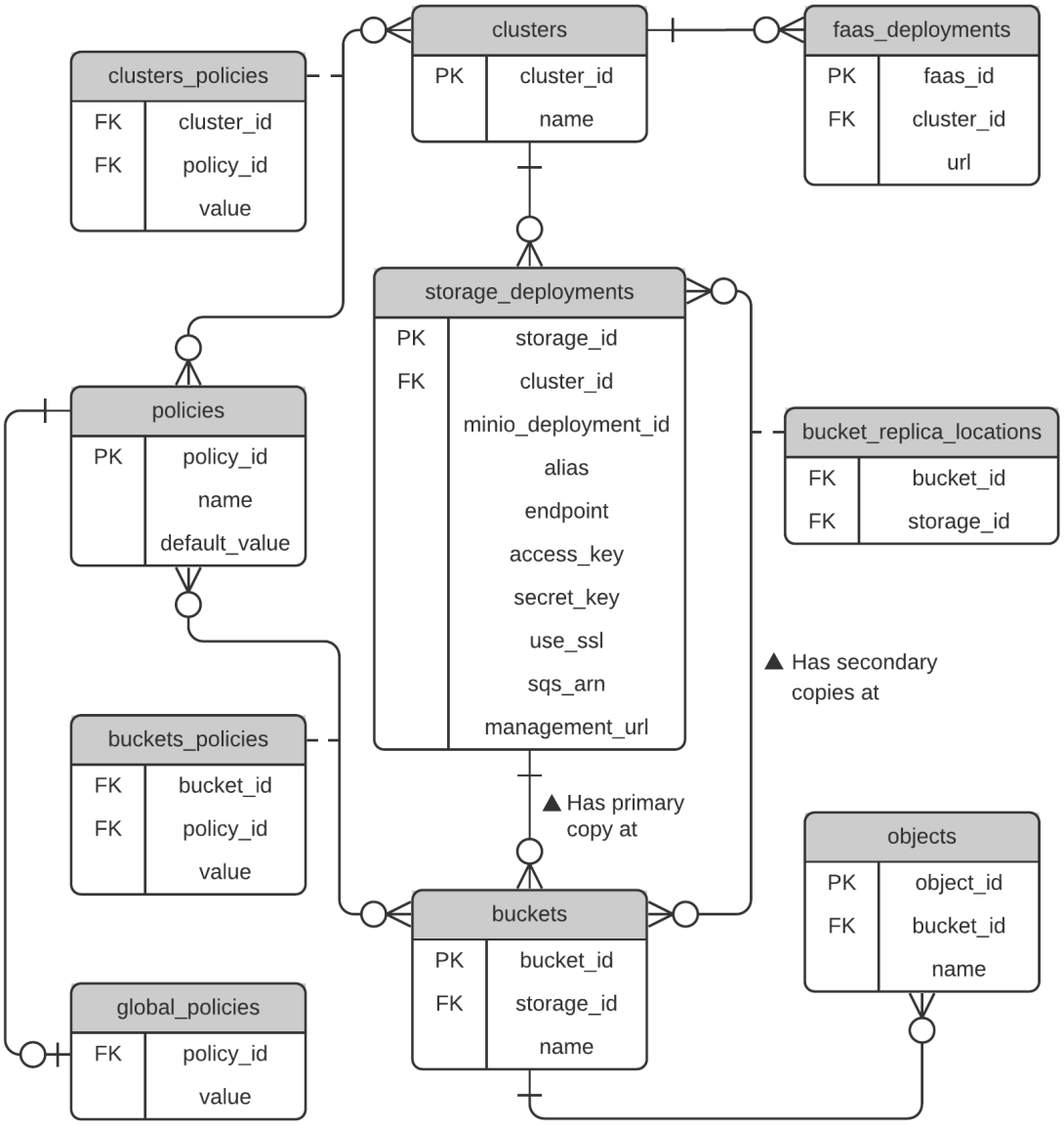


Figure 4.1.: FaDO's data model.

4.1.1. Policies

Policies are used in the data model to define specific settings. These can be global settings or specific to a cluster or a bucket. The schema defines the `policies` table, which lists the system's existing settings, giving them a name and a default value, and the `global_policies`, `clusters_policies`, and `buckets_policies` tables, which allow users to define policy values that should be applied globally, or to a given cluster or bucket. This value then overrides the default value set on the `policies` table.

The value columns are all defined to contain a JSON object. This allows the application to define values very flexibly, as the column can store different data types, such as a string, a number, or a more complex object.

These tables currently store global load balancing settings, *zone* policies to dictate where storage buckets can go, and bucket-specific overrides to manually define bucket replications and bucket load balancing routes.

4.1.2. Clusters

As formalized by the `clusters` table, clusters are records with an ID and a name representing colocated storage and FaaS services.

The records representing FaaS and storage deployments both refer to the cluster to which they belong, forming a one-to-many relationship between clusters and storage and FaaS deployments. Each storage or FaaS deployment belongs to one cluster, and one cluster can have multiple storage or FaaS deployments.

Furthermore, the `clusters_policies` join table links clusters and policies together and forms a many-to-many relationship where a cluster can have many associated policies and vice-versa.

Using these tables and relationships, FaDO users can associate colocated FaaS and storage deployments and apply cluster-wide policies. This allows the definition of zones, which are conceptual delimitations useful for determining storage bucket placement.

4.1.3. FaaS Deployments

The `faas_deployments` table defines the different FaaS services the application is aware of in the platform. It is a very simple table that only indicates the services' URLs and clusters. This is because, at this time, FaDO does not interact with the FaaS clusters beyond reserve-proxying traffic back to them.

4.1.4. Storage Deployments

The `storage_deployments` table tracks all MinIO deployments associated with the application. The table indicates the deployment's cluster and gathers the connection data provided by the user that the application needs to communicate with the services and other relevant metadata that the application gathers subsequently.

This additional metadata includes values like the `minio_deployment_id`, a unique ID that MinIO services generate for themselves and that help FaDO determine from which storage deployment a notification is coming.

4.1.5. Buckets

The `buckets` table lists all the storage buckets the application is aware of amongst the different storage deployments. These buckets are the containers for data objects and the grouping method used to organize data replication and load balancing.

The `buckets_policies` table allows the definition of bucket-specific settings. For example, this is used to define a bucket's *allowed zones*, determining which zones a bucket can be replicated to.

Buckets only directly refer to the one storage deployment that keeps their master copy. However, the `replica_bucket_locations` table also links the tables together to relate bucket replica locations. Through these means, the schema allows the application to determine a bucket's associated clusters through its related storage deployments.

As function invocations specify a storage bucket, the application can choose where to forward the requests by listing the different FaaS deployments in the bucket's associated clusters.

4.1.6. Objects

The `objects` table lists the data objects present on the different storage deployments. Every object belongs to a singular bucket, and their locations can be inferred by their bucket's master and replica locations. Tracking these entities is necessary so that users of the system can query what buckets they need to request when sending function invocations to the application.

4.2. Constraints

Constraints are used throughout the database schema to confine the data's behavior and increase its predictability. This contributes significantly to keeping the data within

the database correct. PostgreSQL's documentation provides an exhaustive reference of the DBMS's different constraints [Thef].

One constraint that was already touched upon in the previous sections is that of foreign key references. Indeed, a database table definition can define a foreign key reference on a column, ensuring that its value must reference another existing object. For instance, as is seen in listing 4.1, the `buckets` table references storage deployments, and the `storage_id` must be a valid storage deployment ID.

The `NOT NULL` and `UNIQUE` constraints, predictably, require that the column values be non-null and unique, respectively. For instance, from listing 4.1, we know the `storage_id` value on `buckets` must be defined. So for any given `buckets` row in the database, we know that the `storage_id` is defined and refers to an existing storage deployment. However, since it does not have the `UNIQUE` constraint, multiple rows can reference the same `storage_id`.

On the other hand, the table's `name` column must contain values that are both defined *and* unique. The same goes for `bucket_id` as the `PRIMARY KEY` constraint is shorthand for `NOT NULL UNIQUE`.

The `ON DELETE CASCADE` clause poses an additional constraint on the bucket row by instructing the database to delete the row if the storage deployment referred to also disappears, while the `DEFAULT` clause defines a default value for a column if none is provided upon insertion.

While these constraints improve the consistency of the data and help protect it from entering into a bad state, they also serve to better describe the nature of the data to the DBMS and help PostgreSQL optimize indexes and speed up data queries [Thec].

```
1 CREATE TABLE buckets (
2   bucket_id               serial        PRIMARY KEY,
3   storage_id              int           NOT NULL
4                                         REFERENCES storage_deployments
5                                         ON DELETE CASCADE,
6   name                    text          NOT NULL UNIQUE
7 );
```

Listing 4.1.: Buckets SQL Table Definition

4.3. Views

While it is straightforward to fetch data when we know its specific features, such as a primary key, it can be less evident to find data through indirect and extended relationships. Again, this is an area where PostgreSQL's relational nature comes in very handy as, using table relationships, we can construct queries that link and summarize data in potent ways.

Let us take the example of storage buckets and FaaS deployments. At first glance, these might not seem related. This is because the storage bucket only links to the storage deployments on which it is found, and the FaaS deployment only references the cluster to which it belongs. Nevertheless, if we traverse both the `storage_deployments`, `replica_bucket_locations`, and `clusters` tables, we can find which FaaS deployments are close to which buckets. This is precisely what happens when FaDO gathers the

```
CREATE VIEW buckets_faas_deployments AS
  SELECT x.bucket_id, x.bucket_name, array_agg(x.faas_id) AS faas_ids,
    array_agg(x.faas_url) AS faas_urls
  FROM (
    SELECT b.bucket_id, b.name AS bucket_name, fd.faas_id,
      fd.url AS faas_url
    FROM buckets b
      INNER JOIN replica_bucket_locations rbl
        ON rbl.bucket_id = b.bucket_id
      LEFT JOIN storage_deployments sd
        ON sd.storage_id = rbl.storage_id
      LEFT JOIN clusters c ON c.cluster_id = sd.cluster_id
      LEFT JOIN faas_deployments fd ON fd.cluster_id = c.cluster_id
    UNION
    SELECT b.bucket_id, b.name AS bucket_name, fd.faas_id,
      fd.url AS faas_url
    FROM buckets b
      LEFT JOIN storage_deployments sd ON sd.storage_id = b.storage_id
      LEFT JOIN clusters c ON c.cluster_id = sd.cluster_id
      LEFT JOIN faas_deployments fd ON fd.cluster_id = c.cluster_id
  ) AS x GROUP BY x.bucket_id, x.bucket_name;
```

Listing 4.2.: Definition of the `buckets_faas_deployments` view summarizing storage bucket and FaaS deployment pairs linked together by their clusters.

bucket_id	bucket_name	faas_ids	faas_urls
6	landscapes	{4,5}	{https://faas.cluster1.fado, https://faas.cluster2.fado}
7	measurements	{5}	{https://faas.cluster2.fado}
8	image-recognition	{6,4,5}	{https://faas.cluster3.fado, https://faas.cluster1.fado, https://faas.cluster2.fado}
9	ml	{7,6,5}	{https://faas.cluster4.fado, https://faas.cluster3.fado, https://faas.cluster2.fado}

Table 4.1.: Example result for a query on the `buckets_faas_deployments` view.

necessary data to configure the load balancer.

However, the queries that achieve these summary results can be quite complex, and for this reason, it can be somewhat impractical to formulate them within the server code itself, especially if it is done repetitively. This is why *views* have been formulated within the database's schema. These views allow the application to reuse complex queries [Theg] and cover the common use-cases where the application needs to construct linked data.

The example shown in listing 4.2 is one such view. First, it links storage buckets and FaaS deployments, starting from the `bucket` records and going through the `storage_deployments` and `clusters` tables to reach the `faas_deployment` records. Then, using the `UNION` clause, it includes both master bucket locations through a bucket's `storage_id` column and replica bucket locations via the `replica_bucket_locations` join table.

Once the linked records are gathered, the results are further processed using PostgreSQL's `array_agg()` function and `GROUP BY` clause. The final result is then a convenient list of buckets accompanied by arrays of FaaS Deployment IDs and URLs resembling the output listed in table 4.1.

5. Backend Server Implementation

Previous chapters introduced FaDO and how it is composed of a backend server, a database, a load balancer, and a frontend client, and chapter 4 laid out the data model and how the PostgreSQL DBMS is used to facilitate data operations.

This chapter will focus on the backend server and its implementation. It will first cover the backend server's various interactions with external systems, including the other FaDO components and MinIO storage deployments. It will then discuss the backend server's internal logic, organized into interdependent *mutations*, and how these are exposed over HTTP via an API. Finally, the chapter will mention various challenges that were faced during development.

5.1. Querying the Database

The application uses the *pgx* and *pgxpool* packages to interface with the database. These provide a Go driver and toolkit for interfacing with PostgreSQL instances and performing safe concurrent operations.

The *pgxconn* package provides facilities to create database connections and transactions, which come in the form of the `pgxconn.Conn` and `pgxconn.Tx` structures. Both these structures receive the `Query()` and `Exec()` methods needed by FaDO.

However, these functions are very general. Since many different queries are sent and many different results are received throughout the application, using these methods directly would lead to repetitive and hard-to-maintain code. Thus, the application requires some abstractions to deal with this, and FaDO's code includes a custom *database* package for this purpose.

As shown in listing 5.1, the package contains the `database.DBConn` interface, which represents a database connection and only expects the presence of `Query()` and `Exec()` methods. Both *pgxconn*'s `pgxconn.Conn` and `pgxconn.Tx` structures implement this interface since they both have suitable methods. This allows the `database.Query()` and `database.Exec()` methods to both receive either of these structures as their first parameter. This is the foundation for all the database abstractions in FaDO's *database* package, which build complex query methods that act on either a `pgxconn.Conn` or a `pgxconn.Tx`.

```
1  // In package database
2
3  type DBConn interface {
4      Query(context.Context, string, ...interface{}) (pgx.Rows, error)
5      Exec(context.Context, string, ...interface{}) (pgconn.CommandTag, error)
6  }
7
8  func Query(c DBConn, sql string, args ...interface{})(pgx.Rows, error) {
9      rows, err = c.Query(ctx, sql, args...)
10     return rows, util.ProcessErr(err)
11 }
12
13 func Exec(c DBConn, sql string, args ...interface{})
14         (pgconn.CommandTag, error) {
15     ct, err = c.Exec(ctx, sql, args...)
16     return ct, util.ProcessErr(err)
17 }
```

Listing 5.1.: The general database facilities from FaDO's *database* package.

5.1.1. Structures and Convenience Methods

Since *pgx* only provides very general functionality, we still run the risk of having repetitive hard-to-maintain code to interact with the database, as we need to interact with the same tables and views in different parts of the application. To mitigate this, the FaDO application defines types and functions within its *database* package built to handle the different database records.

Listing 5.2 shows the definition of the `database.BucketRecord` type, which represents records from the `buckets` database table. The annotations on lines 2 to 4 allow for JSON serialization and facilitate entity name translation between the server and the database and between the server and API clients.

However, the *pgx* package does not know these types and cannot use them directly. Therefore, the `database.ScanBucketRows()` method extracts the table columns from the `pgx.Rows` object received from the query. The order in which the columns are provided is dictated by the order in which they appear in the schema. With this knowledge, the `database.ScanBucketRows()` method can then fill in `database.BucketRecord` objects and return them as a convenient array.

The `database.QueryBuckets()` method thus provides a layer of abstraction over the

```go
type BucketRecord struct {
    BucketID int64 `json:"bucket_id"`
    StorageID int64 `json:"storage_id"`
    Name string `json:"name"`
}

func ScanBucketRows(rows pgx.Rows) (buckets []BucketRecord, err error) {
    for rows.Next() {
        var br BucketRecord
        err = rows.Scan(&br.BucketID, &br.StorageID, &br.Name)
        if err != nil { return buckets, util.ProcessErr(err) }
        buckets = append(buckets, br)
    }
    return
}

func QueryBuckets(conn DBConn, sql string, args ...interface{})
        (buckets []BucketRecord, err error) {
    rows, err := Query(conn, sql, args...)
    if err != nil { return buckets, util.ProcessErr(err) }
    defer rows.Close()

    buckets, err = ScanBucketRows(rows)
    if err != nil { return buckets, util.ProcessErr(err) }

    return
}
```

Listing 5.2.: The database abstraction methods to handle queries on the `buckets` database table from FaDO's *database* package.

raw `database.Query()` method and interprets the output for the caller. Thus, calling `QueryBuckets(conn, "SELECT * FROM buckets")` will yield an array of `BucketRecord` objects with all the bucket records in the database.

The *database* package defines these convenience functions for all tables and views in the database. It also defines convenience methods for querying single rows and inserting and updating objects.

5.1.2. Transactions

The FaDO database contains information about the different storage clusters and the Caddy server's load balancing configuration. However, these systems are all independent from one another, and the database's state may not always be in sync with all of them. Therefore, a cautious approach to data mutations must be taken. Database transactions contribute to this pursuit.

Database transactions wrap around a set of database operations treated as a consistent unit. This unit of instructions can either be *committed* or *rolled back*. All the changes resulting from its operations will be saved if a transaction is committed. However, if it is rolled back, all the changes will be reverted [Thee].

Programs can take advantage of this feature and roll back transactions whenever faults occur. In addition, if the database itself encounters an error during a transaction,

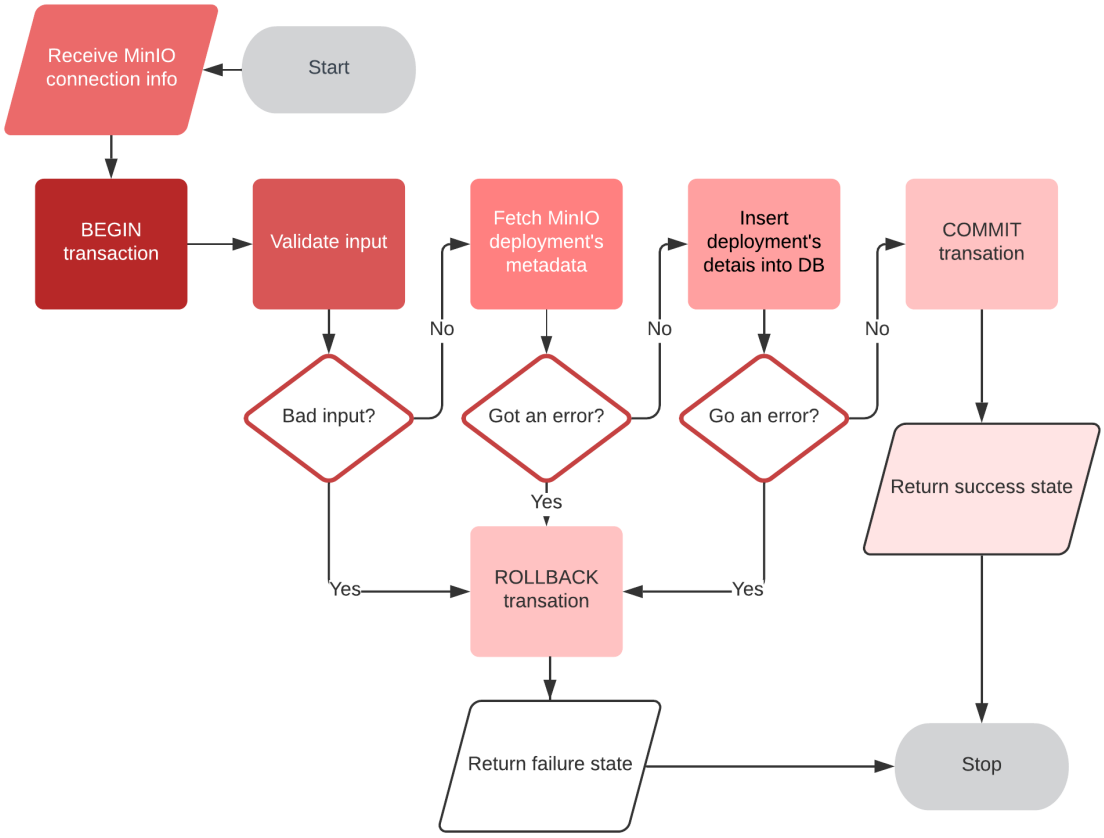


Figure 5.1.: Wrapping operations in a database transaction.

it will also automatically roll back the transaction. Thus programs can ensure changes are committed only when it is appropriate, significantly increasing their reliability.

Figure 5.1 shows the basic sequence of events involved in tracking a new storage deployment in FaDO. At first, the backend server receives input from a client giving the new storage deployment's connection information. At this point, a database transaction is started. The data is then validated and checked against the current database state. Following this, the server fetches various metadata from the storage deployment and then inserts the deployment's details into the database. If everything goes to plan, the transaction is committed, and a successful state is returned, but if an error was encountered along the way, the transaction is rolled back, and an error state is returned instead.

5.2. Configuring the Load Balancer

The Caddy server acting as FaDO's load balancer is responsible for three tasks:

- Reverse proxying and load balancing function invocations to the different FaaS deployments
- Exposing FaDO's backend server to the internet
- Provision TLS certificates and terminate HTTPS traffic

While the load balancing configuration is set and maintained by the FaDO server, the configuration that exposes the server itself is constant and set upon the Caddy server's initial start. To accomplish this, Caddy is provided with a *Caddyfile*. This configuration file follows Caddy-specific formatting and contains the information necessary to expose the FaDO server to the internet.

Listing 5.3 shows the contents of this file, where lines 6 through 12 instruct Caddy to provision a TLS certificate for the domain *server.fado* and reverse proxy the traffic for that domain to port 9090 on host *fado*. This hostname is defined by Docker Compose and associated with the FaDO server.

Line 2 instructs Caddy to operate in *debug* mode, which provides more insightful logging and proved helpful in understanding certain failures. Finally, lines 3 through 5 expose Caddy's administration endpoint, allowing the FaDO server to communicate with Caddy and modify its configuration.

However, this file only contains the configuration necessary for exposing the FaDO server itself. To act as the FaDO load balancer, the Caddy server must be configured to also listen on a different domain and perform header-based reverse proxying to the different FaaS endpoints.

```
{
    debug
    admin 0.0.0.0:2019 { origins * }
}

server.fado {
    reverse_proxy http://caddy:9090 {
        health_uri /healthz
        health_interval 10s
        health_timeout 2s
    }
}
```

Listing 5.3.: The initial Caddyfile configuration containing the configuration necessary to expose the backend server on host *server.fado* over HTTPS.

To accomplish this, the FaDO server adds settings to the configuration instructing the load balancer to listen for a dedicated domain used for FaDO function invocations. We will denote it as *lb.fado*. The resulting configuration looks like the JSON object shown in listing 3.5 from chapter 3.

5.2.1. Generating Bucket Routes

Once the load balancer is configured to receive function invocation requests, routes must be generated for each storage bucket. These routes specify a header match, a load balancing selection policy, and a set of upstream URLs.

Listing 5.4 shows the routes generated by FaDO for two storage buckets, *images* and *ml-models*. Both storage buckets are close to two FaaS endpoints, and so they each have two upstream URLs. The route for *images* specifies the *round_robin* selection policy, which will cause the load balancer to cycle through each upstream one after another. On the other hand, the route for *ml-models* uses the *least_conn* selection policy, standing for *least connections*. This policy causes the load balancer to send the next request to the upstream with the least open connections at that time.

```
[
  {
    "handle": [
      {
        "handler": "reverse_proxy",
        "load_balancing": { "selection_policy": {  "policy": "round_robin" } },
        "upstreams": [
          { "dial": "faas.hpc-cluster.fado:31112" },
          { "dial": "faas.cloud-cluster-2.fado:31112" }
        ]
      }
    ],
    "match": [
      {  "header": { "X-FaDO-Bucket": ["images"] } }
    ]
  },
  {
    "handle": [
      {
        "handler": "reverse_proxy",
        "load_balancing": { "selection_policy": { "policy": "least_conn" } },
        "upstreams": [
          { "dial": "faas.hpc-cluster.fado:31112" },
          { "dial": "faas.cloud-cluster-1.fado:31112" }
        ]
      }
    ],
    "match":[
      { "header": { "X-FaDO-Bucket": ["ml-models"] } }
    ]
  }
]
```

Listing 5.4.: Load balancing routes for *images* and *ml-models* storage buckets with multiple upstreams and different selection policies.

5.3. Interacting with MinIO Services

MinIO provides various tools to interact with their deployments. For example, users can interact with and manage them through a browser client or the *mc* command-line tool, and application developers can integrate MinIO's SDKs into their software to automate exchanges.

The FaDO server relies on MinIO's Go language libraries as much as possible. These are the MinIO Go client SDK for user-scoped actions such as interacting with objects and buckets and the MinIO Go administrator SDK for administration and management tasks.

Though most interactions are enacted using the SDKs, the application still depends on the *mc* command-line tool for two operations: data replication using the `mc mirror` command and setting up notification targets to receive the FaDO server bucket notifications.

This is an unfortunate dependency, as the command-line tool must be present and accessible to the program within its running environment, and the integration is not so tight, with the application having to formulate command-line arguments and parse its output, as well as trust the program to succeed or indeed fail, in a predictable manner.

However, with these tools and methods in place, the application can interact with the platform's various MinIO deployments to accomplish, broadly speaking, four different operations:

- Tracking storage deployments
- Creating and deleting buckets and objects
- Reacting to MinIO notifications
- Replicating data

5.3.1. Tracking a New Deployment

In order to track a new MinIO service in the application state, the server must go through several steps, broadly depicted in figure 5.2.

Before doing anything with a new storage deployment, the server needs to have the information necessary to connect to it. This includes an endpoint that specifies the hostname and port, an access key and a secret key, and an indication concerning the use of SSL. Additionally, the server also requires a name to be given to the deployment, which it will subsequently use for display purposes and setting a configuring a new alias with the *mc* command-line tool to facilitate future interactions with the deployment using the tool.

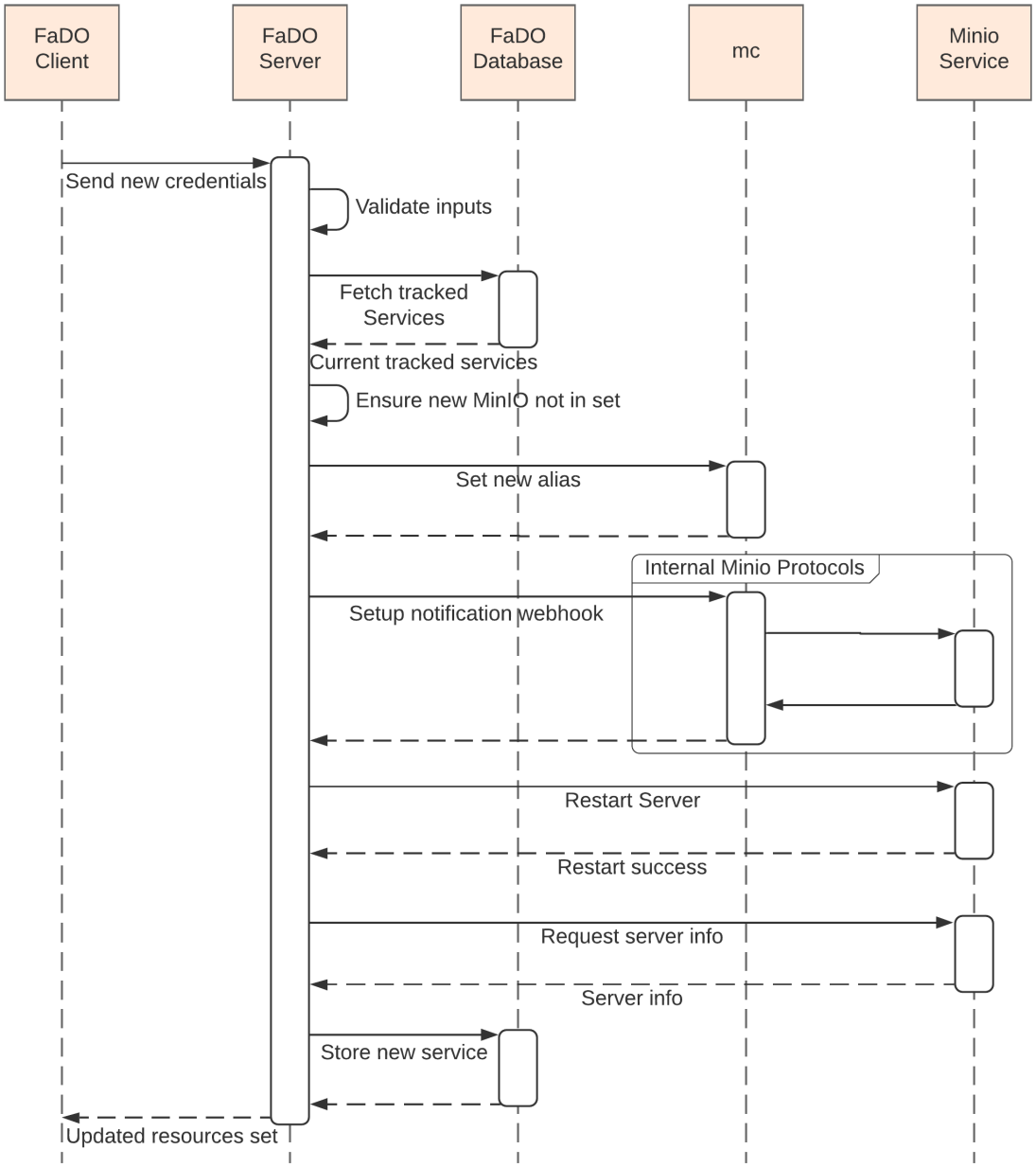


Figure 5.2.: Sequence of events when tracking a new MinIO deployment.

Once the connection information is given, the server can verify its apparent validity. And since, ultimately, the information concerning the new storage deployment will be kept in the database, the server will compare the new connection information against the current set in the database to ensure that a storage deployment is not tracked twice.

Once done, the application configures the *mc* command-line client with the new connection and sets up an alias according to the name provided by the user. It can then use the tool to configure the MinIO deployment with a new notification webhook that points back to the server. This will allow the application to set notification configurations on storage buckets and cause MinIO to send notifications back to the server when changes occur.

After this is set, the MinIO server must be restarted in order for the changes to take full effect. FaDO will trigger this and wait for the database to come back up to fetch the server's updated metadata.

The server extracts an internally generated deployment ID from this metadata and the server's *SQS ARN*, an identifier required for bucket notification configurations.

Once these steps have been successfully completed, the server can safely store the new information in the database.

5.3.2. Manipulating Buckets and Objects

One of the goals for FaDO is to provide a layer of abstraction on top of the storage deployments and allow users to interact with the platform's distributed data in a unified fashion. To this purpose, the application allows for the creation and deletion of storage buckets as well as the addition, downloading, and removal of storage objects.

All these operations are possible using MinIO Go SDK, using the package's `AddBucket`, `RemoveBucket`, `PutObject`, `Getobject`, `DeleteObject` functions.

As the server needs to monitor all storage buckets in the platform, it will add itself as a notification target for each newly created bucket to receive bucket notifications from MinIO whenever a bucket is changed.

The server's API further allows clients to manipulate data objects directly by acting as a go-between and using the MinIO SDK, essentially proxying objects between the client and the MinIO services.

5.3.3. Handling Notifications

For the server to leverage notifications, the different MinIO deployments must be configured to send them, and the server must have the means to receive and process them. To this purpose, the FaDO server has a dedicated API endpoint at the path `/api/notify` that is designed to handle MinIO notifications.

The MinIO service that originates the notification sends a JSON object in the request body. This JSON object can be parsed to extract the affected object's *key* and the storage deployment's `deployment_id`. The key is the object's path, including the bucket in which it is placed. Therefore, it is simple for the program to get both the storage bucket name and the object name. The `deployment_id`, which matches the database's `minio_deployment_id` column, is then used to identify which MinIO deployment sent the event so that FaDO recognizes exactly which object, in which bucket, and on which storage deployment has been affected.

If the bucket that originated the event is a master copy, FaDO will mirror the new contents to all the replica buckets and then update its information on the bucket's contents before reconfiguring the load balancer with the new knowledge.

5.3.4. Mirroring Buckets

One of MinIO's selling points is bucket replication and its bidirectional replication feature. This feature allows users to have two copies of a bucket on different and independent MinIO deployments and set up the services to sync and merge changes from one bucket to the other. This is a very attractive feature in FaDO's context, as it allows granular control over data by confining replication tasks to buckets and proposes an automated process.

```
1 mc mirror  --remove --overwrite minio1/ml minio2/ml
```

Listing 5.5.: Mirroring buckets using the `mc mirror` command.

Unfortunately, this feature is not suitable for FaDO, as for any given storage bucket, FaDO allows for only one replication target. Therefore, it is too limiting and does not allow the fine control over data, with more than one replication location, that FaDO aspires to offer.

Therefore, FaDO relies on the `mc mirror` command, a feature of the *mc* command-line tool that allows bucket mirroring and read replication between buckets on AWS compatible storage services.

However, this tool is much more bare-bones, as it only replicates data in discrete and explicit steps. The command does have a `--watch` option that causes the *mc* program to monitor a bucket for changes. However, mirror jobs can only be defined strictly for one source bucket and one target bucket at a time. Furthermore, as multi-threaded access to the *mc* command-line tool causes faults, the program reserves its use for one application thread at a time using a mutex lock. Therefore, using the `--watch` option is infeasible as it would monopolize the resource for a single replication job.

5.4. State Mutations

Since FaDO interacts with many different external services to accomplish its various tasks, the possible points of failure are quite numerous. Therefore, care is necessary to maximize reliability and consistency when writing the server code.

To this purpose, the FaDO server's code relies on its *mutations* package, which gathers all the various operations on the database, the load balancer, and the MinIO deployments, and organizes them into consistent and interdependent mutation functions.

As the dependency diagram from figure 5.3 shows, many functions cause a waterfall effect through the *mutations* package. This method of organization helps to ensure maximal code reuse and logical consistency without the same tasks or subtasks being accomplished multiple times in different ways.

For instance, if the `DeleteCluster()` method is called, then the `DeleteStorageDeployment()` method will also be called as part of the function's logic, and `DeleteStorageDeployment()` will call the `DeleteBucket()` method in the same way. But the `DeleteStorageDeployment()` and `DeleteBucket()` are also used directly to only delete a storage deployment, or to only delete a bucket.

There is some inevitable overlap of functionality. For instance with the `DeleteMasterBucket()` and `DeleteReplicaBucket()` mutations, and `DeleteObject`. Both lead to data deletion on a MinIO deployment, and bucket deletion also includes the deletion of its objects, so the deletion mutations for master and replica buckets do not depend on the object deletion mutation.

As for replication, both the `AddReplicaBucket()` and `ReplicateBucket()` methods rely on the `mc` package's `Mirror` function to trigger a mirror operation, but they obtain the data necessary for that step differently and so do not have any interdependency within the *mutations* package.

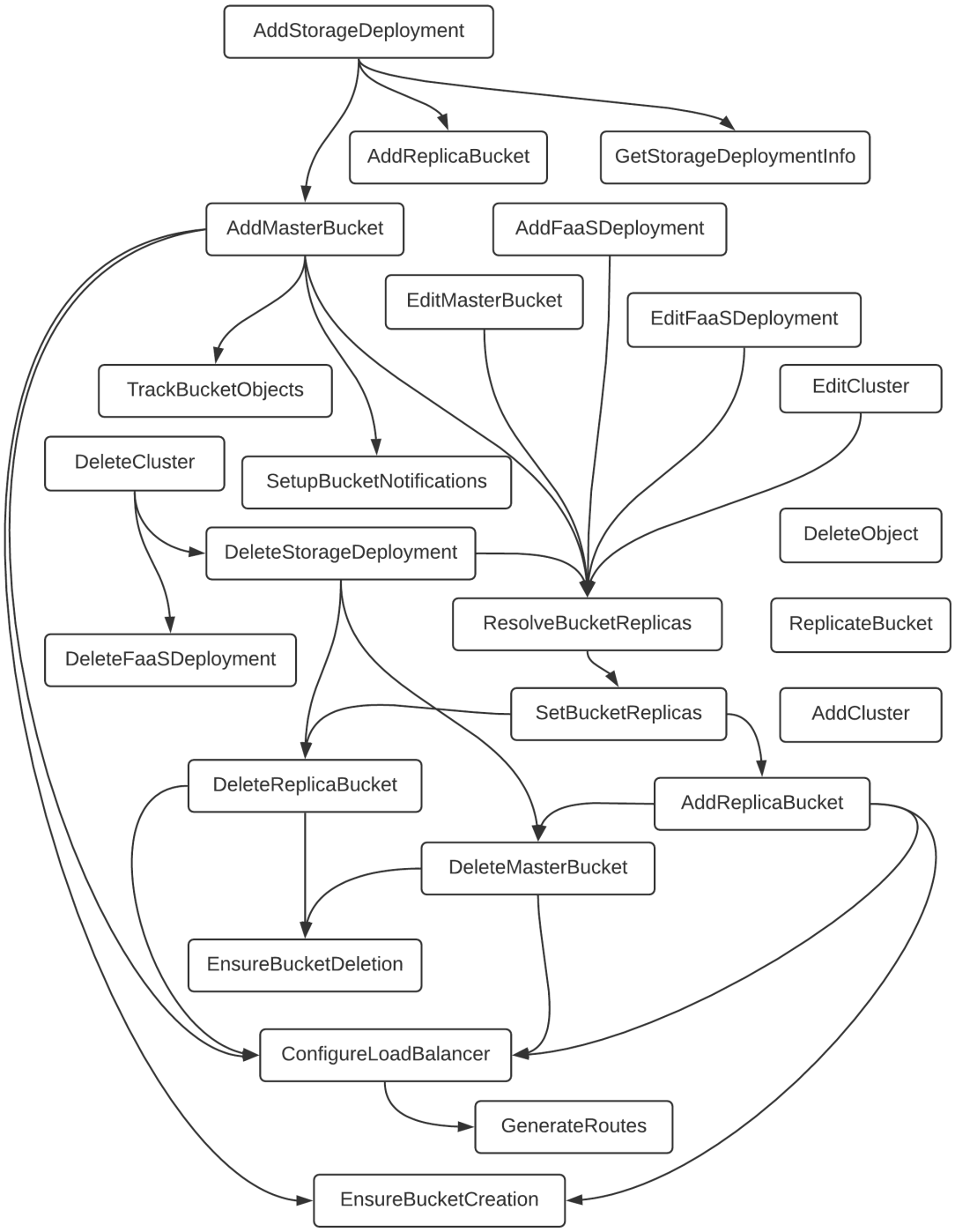


Figure 5.3.: FaDO's mutations dependency diagram.

5.5. HTTP Endpoints

5.5.1. API

The application lets users and clients interact with it through an API exposed over HTTP, allowing users to trigger the mutation functions described in the previous section. The different endpoints, along with the accepted HTTP methods, are as follow:

- `/api/resources`
  `GET` – Retrieve all current resources.

- `/api/clusters`
  `GET` – Retrieve all current resources.
  `POST` – Add a new cluster.

- `/api/clusters/{cluster_id:[0-9]+}`
  `PUT` – Update a cluster. Expects an integer cluster ID.
  `DELETE` – Delete a cluster. Expects an integer cluster ID.

- `/api/faas-deployments`
  `GET` – Retrieve all current resources.
  `POST` – Add a new FaaS deployment.

- `/api/faas-deployments/{faas_id:[0-9]+}`
  `PUT` – Update a FaaS deployment. Expects an integer FaaS deployment ID.
  `DELETE` – Delete a FaaS deployment. Expects an integer FaaS deployment ID.

- `/api/storage-deployments`
  `GET` – Retrieve all current resources.
  `POST` – Add a new storage deployment.

- `/api/storage-deployments/{storage_id:[0-9]+}`
  `DELETE` – Delete a storage deployment. Expects an integer storage deployment ID.

- `/api/buckets`
  `GET` – Retrieve all current resources.
  `POST` – Add a new storage bucket.

- `/api/buckets/{bucket_id:[0-9]+}`
  `PUT` – Update a bucket. Expects an integer bucket ID.
  `DELETE` – Delete a bucket. Expects an integer bucket ID.

- `/api/objects`
  `GET` – Retrieve all current resources.
  `POST` – Upload an object.
- `/api/objects?path={object_path}`
  `GET` – Download the object matching the `"bucket_name/object_name"` path.
- `/api/objects/{object_id:[0-9]+}`
  `GET` – Download an object. Expects an integer object ID.
  `DELETE` – Delete an object. Expects an integer object ID.
- `/api/load-balancer`
  `GET` – Retrieve all current resources.
- `/api/load-balancer/settings`
  `PUT` – Define the load-balancer's global settings.
- `/api/load-balancer/route-overrides`
  `PUT` – Define the load-balancer's overriden routes.
- `/api/notify`
  `*` – Handle MinIO bucket notifications. Accepts any method.

The API is organized following the rule of thumb that `GET` requests retrieve information, `POST` requests create an item, `PUT` requests update an item, and `DELETE` requests delete an item. To accelerate the development process and focus on more impactful features, all `GET` requests return the current list of resources. `POST`, `PUT`, and `DELETE` do the same after successfully completing their tasks.

5.5.2. Other HTTP Endpoints

The server also responds to two other endpoints:

- `/healthz`
  `GET` – Return HTTP status OK.
- `/*` (Catch-all fallback)
  `GET` – Serve the frontend single-page application.

The first item is a health-check endpoint used once deployed to monitor the server's health. If the endpoint ceases to respond correctly, the monitor knows that the server is unhealthy.

The second endpoint is used to serve FaDO's frontend client. It will return any existing frontend file corresponding to the request path or redirect to the `index.html` file.

6. Frontend Development

The previous chapters introduced FaDO and its different components and gave a detailed account of FaDO's backend components, including the database, the backend server, and the load balancer. This chapter will now focus on FaDO's frontend client, which gives users an intuitive visual interface to interact with the application.

This frontend client is a JavaScript single-page application. It is a static website served by the FaDO server next to the backend API, and users can access it through a browser.

Upon initial page load, the frontend application sends an HTTP GET request to the FaDO server's `/api/resources` endpoint to retrieve the current set of resources and settings stored in the database. Once the data is retrieved and parsed, the application can populate the website, allowing users to interact with up-to-date data.

The application is split into different pages dedicated to the different entities in the FaDO application's data. Users can access each page by clicking on links displayed throughout the client application or navigating to their URL in the browser. The Javascript application parses the URL and uses it to determine the appropriate view to display.

In addition to a main dashboard page, there is a page dedicated to each of the following:

- Clusters
- FaaS Deployments
- Storage Deployments
- Storage Buckets
- Data Objects
- The Load Balancer

We will refer to the first five as resource pages and the last as the load balancer settings page.

6.1. Resource Pages

The resource pages are all structured around a list of expansion panels that provide details concerning individual resources. As can be observed in figure 6.1, the collapsed items display summary information, such as, in this case, the bucket's name, the target number of replicas, and allowed zones. Once expanded, the user can see more detailed information, and so we can see that bucket *machine-learning* has its master copy on storage deployment *d2*, and is replicated to *d3*. The expanded item also lists the data objects stored in the bucket, in this case, three JPEG images, and exposes *edit* and *delete* buttons.

With these two buttons and the *add* button at the top right corner of the page, the user can send mutation requests to the backend FaDO server's API. All three buttons will cause a dialog to appear, such as the one visible in figure 6.2, that allows the user to type out information and submit it to the application.

When the user submits data, the application sends an AJAX request to the appropriate API endpoint. Once the request resolves, the application receives back an updated set of data that reflects the changes the user acted. Upon receiving the updated data, the

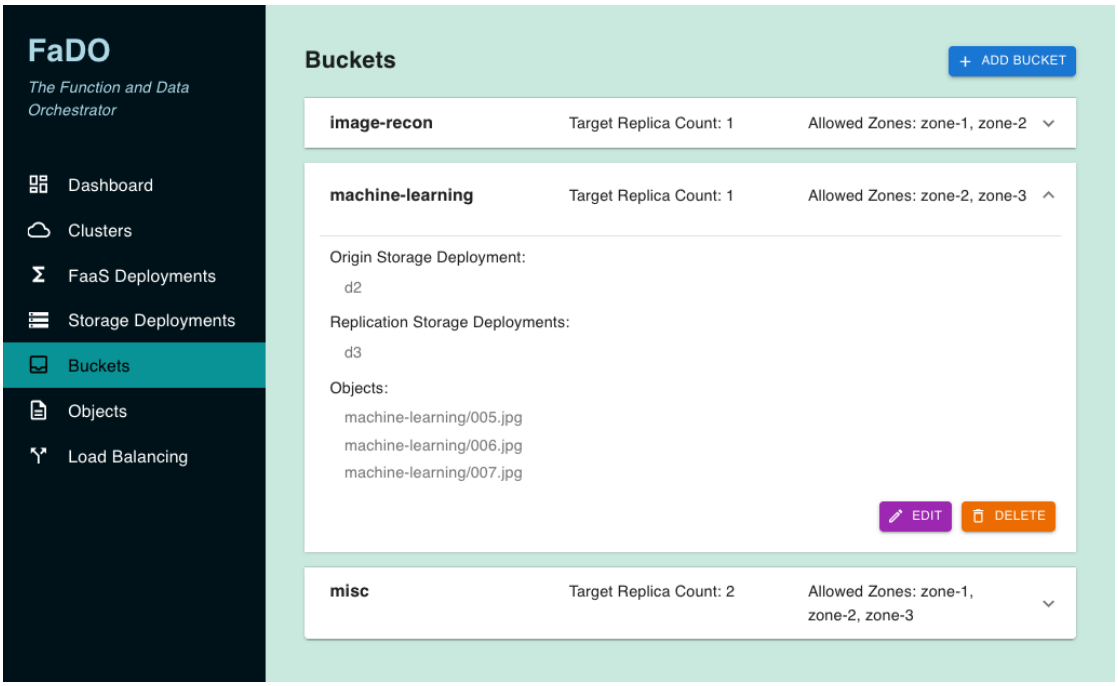


Figure 6.1.: Listing the platform's storage buckets in FaDO's frontend client.

application can immediately update the data shown to the user without requiring a page reload.

The user can thus use the different resource pages to achieve the following:

- Track and monitor or stop tracking storage deployments
- Store or delete FaaS endpoint information
- Organize the above resources into clusters and define cluster policies
- Create and remove storage buckets and configure their replication across the platform
- Manually override FaDO's bucket replication behavior
- Upload, download and delete data objects across the platform

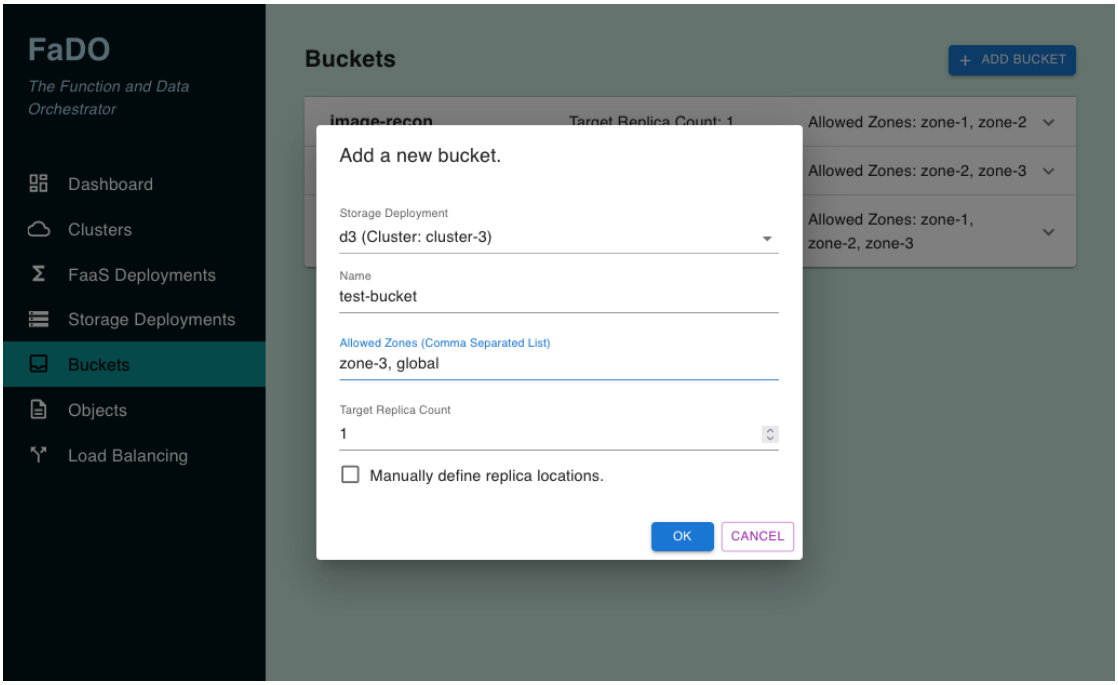


Figure 6.2.: Adding a new storage bucket using FaDO's frontend client.

6.2. Load Balancer Settings Page

Seeing figure 6.3, the load balancer settings page does not look entirely unlike the resource pages, with the bottom half of the page displaying a list of expansion panels. However, this page is entirely dedicated to configuration values governing the load balancer's behavior and how FaDO determines this configuration.

The *view raw configuration* button at the top right corner of the page displays the Caddy server's raw JSON configuration set by the backend server, as can be seen in appendix B's figure B.22, while the panel in the top half of the page displays global settings such as the ingress domain and port, the load balancing policy that FaDO uses by default, and the specific header name that is used to convey a function's required storage bucket.

Users can modify the global load balancing settings by clicking on the *edit* button in the bottom right corner, which opens up a dialog with a form allowing users to modify these global values, as shown in appendix B's figure B.23. Once the form is submitted, the backend server uses the new values to update the configuration and sends the new data back to the client.

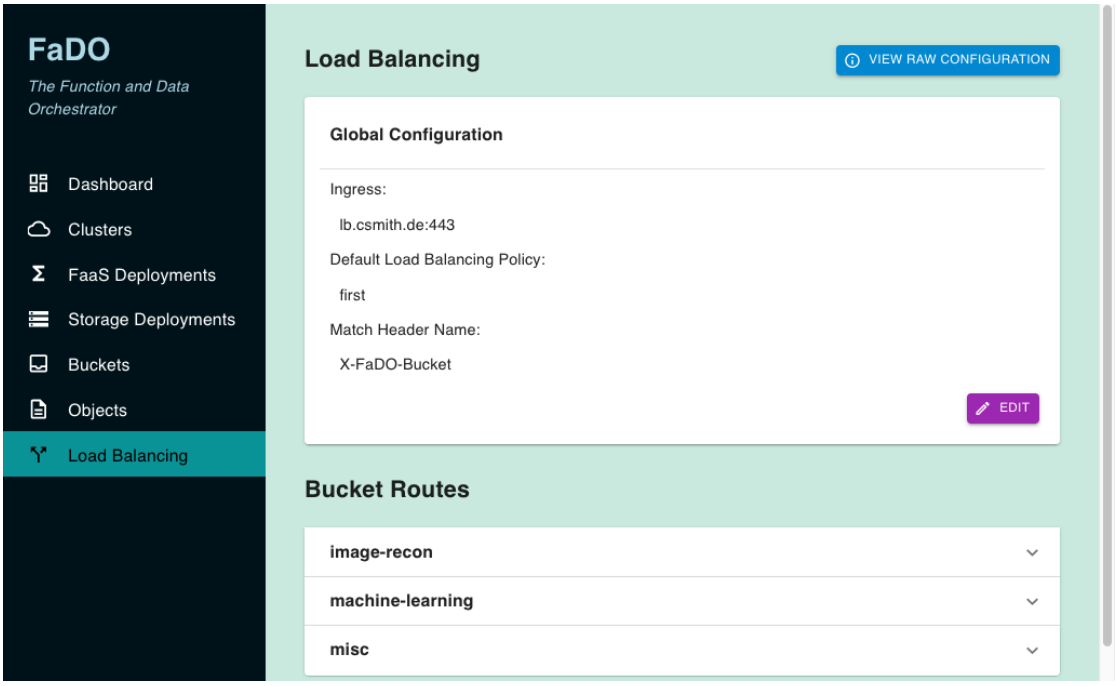


Figure 6.3.: Displaying the load balancer's global settings in FaDO's frontend client.

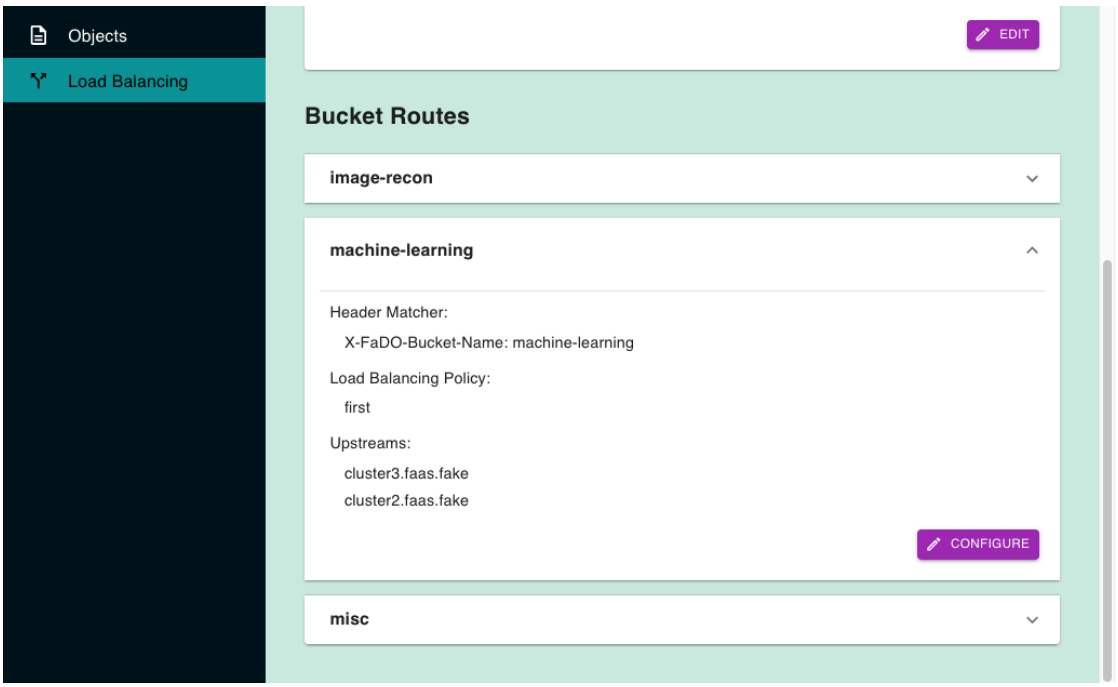


Figure 6.4.: Displaying the load balancer's route settings in FaDO's frontend client.

The list of expansion panels at the bottom half of the page shows routing information for the platform's different storage buckets. A bucket's route is active when the header mentioned above matches the bucket's name. As shown in figure 6.4, the expanded panel then details the list of upstreams the request will be forwarded to and the load balancing policy used to choose the target upstream for a given request. These upstreams correspond to the different FaaS endpoint URLs associated with the bucket.

As appendix B's figure B.25 shows, clicking on the *configure* button at the bottom right corner of the panel reveals a dialog allowing users to submit a custom route configuration to the backend server's API. These route configurations override FaDO generated configurations for those routes, allowing users to manually specify the set of FaaS endpoints and load balancing selection policy for any given storage bucket.

7. Results

The previous chapters presented FaDO, first giving an overview of its structure before detailing the database's design, the backend server's implementation, and the frontend client's organization. This chapter will evaluate the application through a qualitative assessment and measurements obtained. The first section will detail the testing environment. The chapter will then consider FaDO's replication and storage policy mechanisms before evaluating FaDO's replication performance and load balancing capabilities. Finally, the chapter will discuss the FaDO's viability as a network application.

7.1. Testing Environment

A controlled testing environment was prepared for the testing phase. It includes four computing clusters, a control plane server, and a local machine. The clusters and the control plane form FaDO's cloud platform, while the local machine is valuable in testing MinIO's *mc* client, which FaDO relies on for replications, in additional conditions. The resources were as follows:

1. A local machine connected to the internet through a domestic provider. It will be named **local machine**.
2. A virtual machine provisioned from a private cloud and intended as the platform's control plane. It will be referred to as the **control plane**.
3. A high-performance computing (HPC) cluster provisioned from a private cloud. It is an HPC node and is the platform's most powerful cluster. It will be called **HPC cluster**.
4. A cloud cluster comprises three virtual machines provisioned from a private cloud provider. Though not as performant as the HPC cluster, it is is still very capable. It will be referred to as the **cloud cluster 1**.
5. A cloud cluster similar to cloud cluster 1, but less powerful, which will be referred to as the **cloud cluster 2**.

6. A cloud cluster provisioned from AWS, with a *t2.medium* instance VM running the storage server and functions running on AWS Lambda with 1024MB memory. It is unique from the other clusters, as AWS will automatically scale the function to cope with heavy workloads. It will be called **AWS Cluster**.

The control plane hosts FaDO's backend, which is to say, the backend server, the Caddy load balancer, and the PostgreSQL database. The platform clusters from the private cloud provider run a MinIO storage deployment and an OpenFaaS server, while the AWS cluster runs a MinIO storage deployment alongside AWS Lambda functions. Table 7.1 lists hardware and software characteristics for each listed resource, indicating CPU and memory properties and the presence or lack of a GPU. Details concerning AWS Lambda are not listed as the service does not specify them.

Resources	**Processor**	**H/W Spec.**	**Software**	**GPU**
Local Machine	Apple M1 2.10GHz - 3.20GHz	8 CPUs 16GiB mem.	mc	No
Control Plane	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	2 CPUs 4GiB mem.	FaDO, mc k6	No
HPC Cluster	Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz	88 CPUs 251GiB mem.	OpenFaaS, MinIO	Yes
Cloud Cluster 1	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	20 CPUs 370GiB mem.	OpenFaaS, MinIO	Yes
Cloud Cluster 2	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	4 CPUs 8GiB mem.	OpenFaaS, MinIO	No
AWS Cluster (Storage)	Intel Scalable CPU Up to 3.3GHz	2 vCPUs 4GiB mem.	MinIO	No

Table 7.1.: Hardware specifications for the testing platform's different resources.

Additionally, each FaaS service is prepared with two functions:

- An image processing function at path `/function/image` that receives a bucket and object name. It fetches the data from its local MinIO service and manipulates the image before finishing. It will be referred to as the ***image*** function.
- An information function at path `/function/nodeinfo` that does not take any input and does not access MinIO, but simply returns information concerning the current cluster. It will be referred to as the ***nodeinfo*** function.

7.2. Data Replication and Storage Policies

Beyond letting users override storage configurations and manually define where a master bucket should be replicated and which FaaS upstreams a bucket's load balancing route should contain, FaDO allows users to control replication and load balancing through storage configurations. These policies include buckets' allowed zones and their target replica counts. For example, if a bucket does not have allowed zones defined, it will be replicated indiscriminately onto other storage deployments. However, if a set of zones has been defined, FaDO will ensure that the bucket is only replicated to storage deployments within clusters that share a zone with the bucket. Additionally, if zones change on a bucket or a cluster or a bucket's target replica count is altered, FaDO will ensure that replica buckets are created and deleted where needed.

A scenario was devised to confirm that FaDO's replication and policy mechanisms work wherein gradual changes are made using FaDO's frontend client, and the results of the backend server's operations are recorded by using MinIO's *mc* tool with the `mc ls` command to list buckets and objects and fetching Caddy's configuration. The steps took place as follows:

1. The HPC cluster and both cloud clusters are set up. Each belongs to the zone *germany*, while the HPC cluster and cloud cluster 1 also belong to zone *high-performance*.

   The output from `mc ls` shows there are currently no buckets on the platform:

   ```
   $ mc ls minio-hpc --recursive # List all data on the HPC cluster
   $ mc ls minio-cc1 --recursive # List all data on cloud cluster 1
   $ mc ls minio-cc2 --recursive # List all data on cloud cluster 2
   ```

   And, through the absence of a list of routes in the map on line 3, Caddy's configuration shows there are no buckets to route to[1]:

```
1 {
2   "handle": [
3     { "handler": "subroute" }
4   ],
5   "match": [{ "host": ["lb.fado"] }],
6   "terminal": true
7 }
```

[1] Artificial hostname *lb.fado* used instead of privately owned domain.

2. A bucket named *rep-policy-demo* is created, and an object named *image.png* is added to it. The bucket is set with the allowed zone *germany* and the target replica count 0.

   Output from `mc ls` shows the bucket is only created on the HPC cluster:

```
$ mc ls --recursive minio-hpc
[2021-12-06 19:29:34 CET] 9.5MiB rep-policy-demo/image-1.png
$ mc ls --recursive minio-cc1
$ mc ls --recursive minio-cc2
```

   Caddy's configuration now contains the following bucket route, which will direct function invocations requiring the bucket solely to the HPC cluster[2]:

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" } },
      "upstreams": [{ "dial": "faas.hpc-cluster.fado:31112" }]
    }
  ],
  "match":[
    { "header": { "X-FaDO-Bucket": ["rep-policy-demo"] } }
  ]
}
```

3. The *rep-policy-demo* bucket's target replica count is changed from 0 to 2.

   Output from `mc ls` shows the bucket has been replicated to the other two clusters:

```
$ mc ls --recursive minio-hpc
[2021-12-06 19:29:34 CET] 9.5MiB rep-policy-demo/image-1.png
$ mc ls --recursive minio-cc1
[2021-12-06 19:30:54 CET] 9.5MiB rep-policy-demo/image-1.png
$ mc ls --recursive minio-cc2
[2021-12-06 19:30:53 CET] 9.5MiB rep-policy-demo/image-1.png
```

   And the load balancing route now contains the three clusters' upstreams[3,4]:

[2]Artificial hostname *faas.hpc-cluster.fado* used instead of IP address for clarity.
[3]Artificial hostname *faas.cloud-cluster-1.fado* used instead of IP address for clarity.
[4]Artificial hostname *faas.cloud-cluster-2.fado* used instead of IP address for clarity.

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" } },
      "upstreams": [
        { "dial": "faas.hpc-cluster.fado:31112" },
        { "dial": "faas.cloud-cluster-1.fado:31112" },
        { "dial": "faas.cloud-cluster-2.fado:31112" }
      ]
    }
  ],
  "match":[
    { "header": { "X-FaDO-Bucket": ["rep-policy-demo"] } }
  ]
}
```

4. A second image named *image-2.png* is added to the *rep-policy-demo* bucket.

   Output from `mc ls` shows that the new image has been mirrored to all the replicas:

```
$ mc ls --recursive minio-hpc
[2021-12-06 19:29:34 CET] 9.5MiB rep-policy-demo/image-1.png
[2021-12-06 19:32:27 CET] 9.5MiB rep-policy-demo/image-2.png
$ mc ls --recursive minio-cc1
[2021-12-06 19:30:54 CET] 9.5MiB rep-policy-demo/image-1.png
[2021-12-06 19:32:27 CET] 9.5MiB rep-policy-demo/image-2.png
$ mc ls --recursive minio-cc2
[2021-12-06 19:30:53 CET] 9.5MiB rep-policy-demo/image-1.png
[2021-12-06 19:32:27 CET] 9.5MiB rep-policy-demo/image-2.png
```

   And the load balancing route has not been altered:

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" } },
      "upstreams": [
        { "dial": "faas.hpc-cluster.fado:31112" },
        { "dial": "faas.cloud-cluster-1.fado:31112" },
```

```
        { "dial": "faas.cloud-cluster-2.fado:31112" }
      ]
    }
  ],
  "match":[
    { "header": { "X-FaDO-Bucket": ["rep-policy-demo"] } }
  ]
}
```

5. The *rep-policy-demo* bucket's allowed zones are set to no longer contain zone *germany*, and only contain zone *high-performance*.

   Output from `mc ls` shows the bucket has been removed from cloud cluster 2, which is not in the *high-performance* zone:

```
$ mc ls --recursive minio-hpc
[2021-12-06 19:29:34 CET] 9.5MiB rep-policy-demo/image-1.png
[2021-12-06 19:32:27 CET] 9.5MiB rep-policy-demo/image-2.png
$ mc ls --recursive minio-cc1
[2021-12-06 19:30:54 CET] 9.5MiB rep-policy-demo/image-1.png
[2021-12-06 19:32:27 CET] 9.5MiB rep-policy-demo/image-2.png
$ mc ls --recursive minio-cc2
```

   And the load balancing route now only lists the upstreams for the HPC cluster and cloud cluster 1:

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" } },
      "upstreams": [
        { "dial": "faas.hpc-cluster.fado:31112" },
        { "dial": "faas.cloud-cluster-1.fado:31112" }
      ]
    }
  ],
  "match":[
    { "header": { "X-FaDO-Bucket": ["rep-policy-demo"] } }
  ]
}
```

These recordings confirm the correctness of FaDO's replication and storage policy mechanisms. The backend server ensures that buckets are only replicated to allowed locations, reacting to changes to keep the replicated data up-to-date and the replica buckets consistent with user policies. In this manner, users can define location restrictions and replica count targets to control how their data is distributed across the platform.

7.3. FaDO's Replication Performance

FaDO's replication performance depends entirely on MinIO's *mc* command-line tool and its `mc mirror` directive. Therefore, to assess FaDO's replication performance, it was useful to isolate the command line tool and measure its performance directly. To accomplish this, a bucket was created on the HPC cluster and fully replicated to cloud cluster 2 with data amounts ranging from 10MB to 150MB. The test was then run under the two following scenarios:

1. Use `mc mirror` from the control plane to replicate the bucket's entire data, from 10MB to 150MB.

2. Use `mc mirror` from the local machine to replicate the bucket's entire data, from 10MB to 150MB.

All data objects used throughout the tests were precisely 10 MB in size, and the scenarios advanced in 10 MB increments. The scenarios were executed twenty times, and the resulting graph in figure 7.1 relates the average mirror completion times.

Figure 7.1 shows a sizeable discrepancy between invoking `mc mirror` on the control plane and on the local machine. Indeed, when on the control plane, transferring 150MB took just shy of 2 seconds, whereas on the local machine, it was 15 times longer at about 30 seconds. This demonstrates what MinIO refers to as "client-side replication", where the *mc* client first downloads data to its location from the source before uploading it to the destination [Minb].

The control plane benefits from low latency and high bandwidth links between itself and the clusters. On the other hand, the local machine is on a domestic internet connection. The local machine then simulates a lower quality link, such as might be expected in IoT contexts, and the results demonstrate how dramatic the network link quality can be on replication performance.

This can be quite an unfortunate circumstance, as the replication performance between two storage deployments does not depend on the quality of the link directly between them but rather on the lowest quality link between them and the control plane.

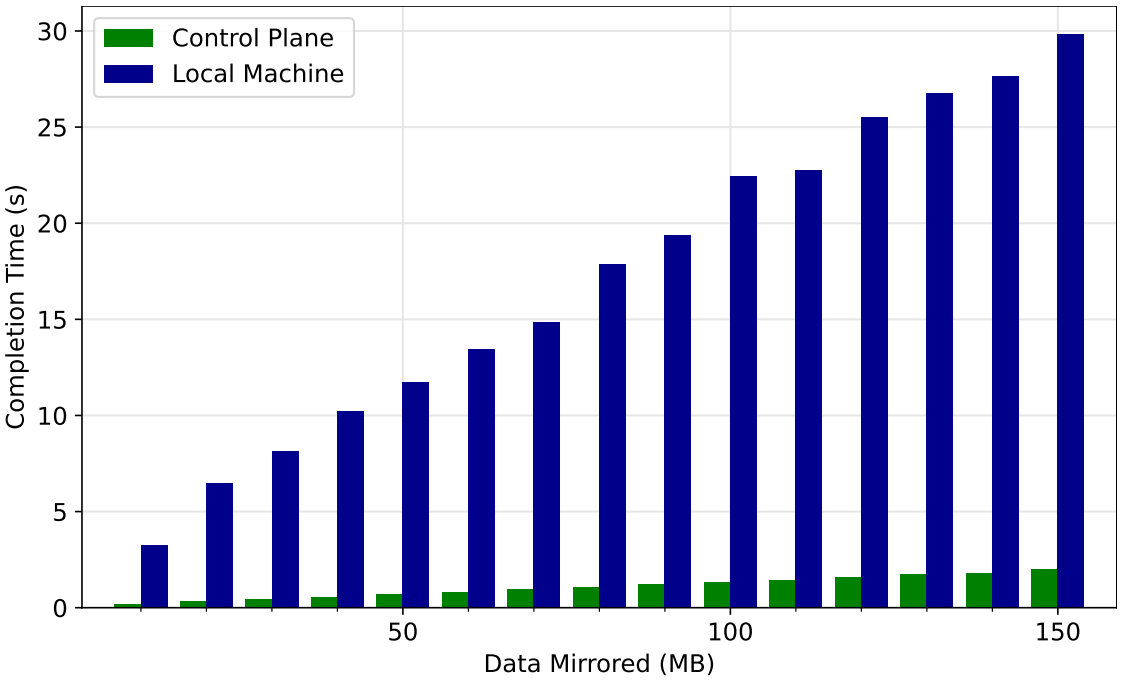


Figure 7.1.: Invoking `mc mirror` from the control plane and the local machine to mirror different data amounts from the HPC cluster to cloud cluster 2.

This is notably the case when testing replication from the local machine, as the quality of the link between the HPC cluster and cloud cluster 2 is excellent.

Using MinIO's replication facilities does, however, come with some advantages. Listing 7.1 shows a sequence of events where data is successively copied to a primary bucket on the HPC cluster and mirrored to a secondary bucket on cloud cluster 2. In this scenario, the secondary bucket contains some of the primary's content and only receives the new data at each step. The output from the `mc mirror` command shows that the MinIO client only mirrors new data. This is because MinIO's mirroring tool works similarly to Andrew Tridgell and Paul Mackerras' *rsync* utility [Mind; TM], where the tool calculates the difference between the two buckets and only transfers the data necessary to make their content match.

This is a significant advantage to FaDO, as it dramatically reduces the amount of data the application must transfer for replication tasks. It also protects the application from many large replication jobs that mutex lock the *mc* tool for long periods.

These results confirm MinIO's claims that their `mc mirror` facility operates similarly to Andrew Tridgell and Paul Mackerras' *rsync* utility [Mind; TM], transferring only the missing data to the mirror destination. This is a highly desirable feature for FaDO as it

greatly optimizes replication tasks and limits the amount of data transferred around the platform.

Despite this advantage, using the MinIO command-line client for replication tasks is a significant flaw in FaDO's design. The issues that it creates can be summarized in four points:

- Data is forced through the control plane, creating a central bottleneck
- User requirements and regulations might conflict with data invariably going to the control plane
- Data transfers greatly increase the load on the backend server
- The backend server is a central point of failure for all replication tasks

A potential solution to these issues may extend FaDO's design by introducing "storage adapter" components. These would be deployed next to the platform's different storage

```
$ mc --quiet cp 001.dat minio-hpc/mirror-incr
`001.dat` -> `minio-hpc/mirror-incr/001.dat`
Total: 0 B, Transferred: 10.00 MiB, Speed: 1.76 MiB/s
$ mc --quiet mirror minio-hpc/mirror-incr minio-cc2/mirror-incr
`minio-hpc/mirror-incr/001.dat` -> `minio-cc2/mirror-incr/001.dat`
Total: 0 B, Transferred: 20.00 MiB, Speed: 1.27 MiB/s
$ mc --quiet cp 002.dat minio-hpc/mirror-incr
`002.dat` -> `minio-hpc/mirror-incr/002.dat`
Total: 0 B, Transferred: 10.00 MiB, Speed: 757.56 KiB/s
$ mc --quiet mirror minio-hpc/mirror-incr minio-cc2/mirror-incr
`minio-hpc/mirror-incr/002.dat` -> `minio-cc2/mirror-incr/002.dat`
Total: 0 B, Transferred: 20.00 MiB, Speed: 1.74 MiB/s
$ mc --quiet cp 003.dat minio-hpc/mirror-incr
`003.dat` -> `minio-hpc/mirror-incr/003.dat`
Total: 0 B, Transferred: 10.00 MiB, Speed: 1.19 MiB/s
$ mc --quiet mirror minio-hpc/mirror-incr minio-cc2/mirror-incr
`minio-hpc/mirror-incr/003.dat` -> `minio-cc2/mirror-incr/003.dat`
Total: 0 B, Transferred: 20.00 MiB, Speed: 1.37 MiB/s
```

Listing 7.1.: Incrementally copying data into a bucket and mirroring it to a replica using the *mc* command-line tool.

deployments and responsible for direct operations on the data. In addition, they could expose a storage API allowing the backend server to access data when needed while taking responsibility away from the backend server for replication tasks. Being deployed close to the different storage services, these storage adapters could achieve distributed and decentralized data replication, avoiding a central bottleneck and using the best link possible for any given replication task.

Different storage adapters could even be created to handle different storage technologies. While this would make their design and operations more complex, they would also make FaDO more compatible with multi-cloud platforms where users might be constrained to different storage technologies.

7.4. Load Balancing

The *k6* utility was used to test FaDO's load balancing capabilities. The program is designed specifically for load testing applications and was the perfect tool to put FaDO through its paces.

To run tests with *k6*, the tool must be provided with a test script, which is a file written in JavaScript that tells the utility what to do. This file will contain the necessary instruction to apply load to the target application, such as sending HTTP requests. The utility will then simulate users with *virtual users* (VUs), where each VU is a process that loops through the script [Gra]. If not otherwise configured, k6 will generate one VU and simply loop through the script until the test is over.

Load testing scenarios control the amount of active VUs over time by breaking down a test into several stages, where each stage has a defined time duration and a target VU count. The *k6* utility will use these scenarios to decide when to ramp VUs up and down and by how much. This will consequently increase and decrease the load on the application, respectively, and define the resulting load pattern.

The two load patterns used for testing FaDO, as shown in figure 7.2, are as follows:

- **Load pattern 1**: this pattern is the most demanding of the two. It starts with a low number of VUs, and thus a low load. It maintains this low load for about 5 minutes before quickly ramping up to a high load with about 80 VUs. After this, it decreases, periodically spikes back up until it plateaus at around 20 VUs.
- **Load pattern 2**: this pattern is much less taxing on the application, keeping a relatively low VU count with minor spikes for the majority of the test. It does, however, have a significant spike in load towards the end.

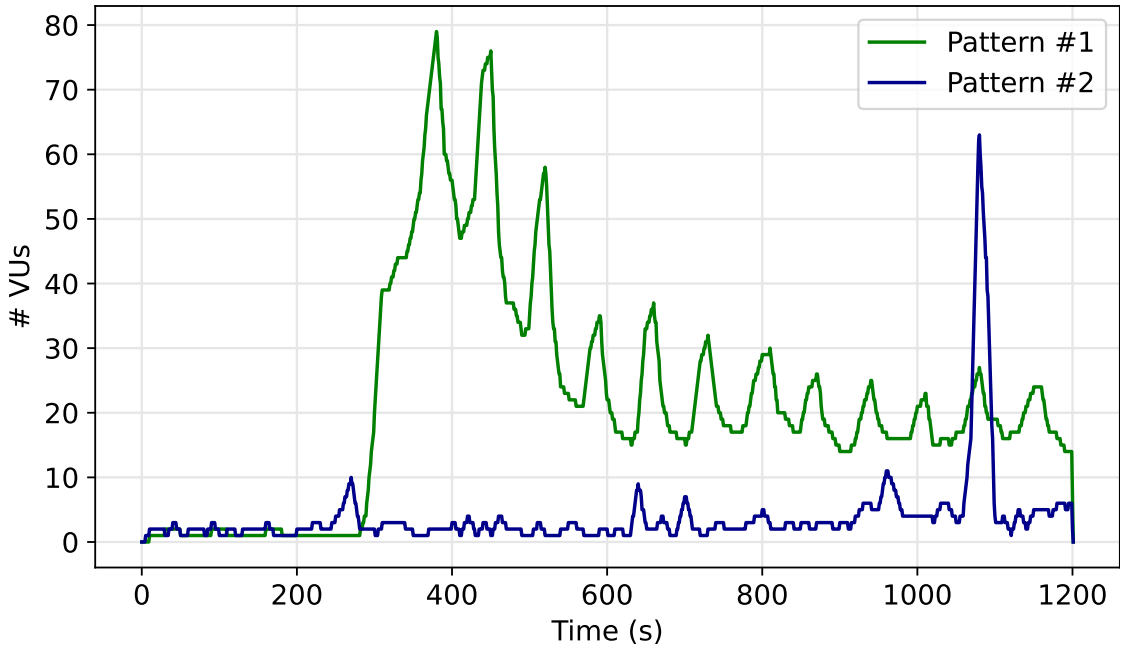


Figure 7.2.: The two load patterns used with *k6* visualized as the number of virtual users over time.

All tests were conducted using both patterns. However, this section will focus on the first, while results for the second pattern can be found in appendix C. These tests can be further split into three categories:

- **Direct connection tests**: these tests send function invocations directly to a single cluster. These determine a cluster's expected performance.
- **FaDO connection tests**: these tests send function invocations to a single cluster through FaDO's load balancer. These determine a cluster's performance through FaDO.
- **Policy tests**: these tests leverage FaDO's load balancing policies to distribute function invocation between the HPC cluster and cloud clusters 1 and 2.

While the FaDO connection test can be considered a load balancing policy, where only one upstream is selected, the policy tests refer more specifically to the following upstream selection policies:

- ***Least Connections***: the load balancer sends the incoming request to the upstream with the least open connections.

- ***Round Robin***: the load balancer cycles through the list of upstreams.
- ***Random***: the load balancer picks a random upstream for each incoming request.

Figure 7.3 relates the measurements recorded for the first load balancing pattern applied to the HPC cluster's OpenFaaS endpoint in both the direct connection and the FaDO connection tests. The load pattern is clearly distinguishable from the figure. As it charts the number of requests completed over time, this would suggest that they stay proportional to the number of active VUs during the test and that performance is not being throttled.

The same tests conducted on the three other clusters yield the same results, indicating that the load balancer does not significantly impact request throughput, and thus on the cluster performances.

The data visualizations further confirm these deductions in figure 7.4. The charts relate the cumulative distribution functions (CDFs) for each cluster's recorded response times, comparing direct and FaDO connection tests.

The X-axes relate the recorded response times, while the Y-axes indicate their probabilities. For instance, the AWS cluster recordings show a steep ascent right around

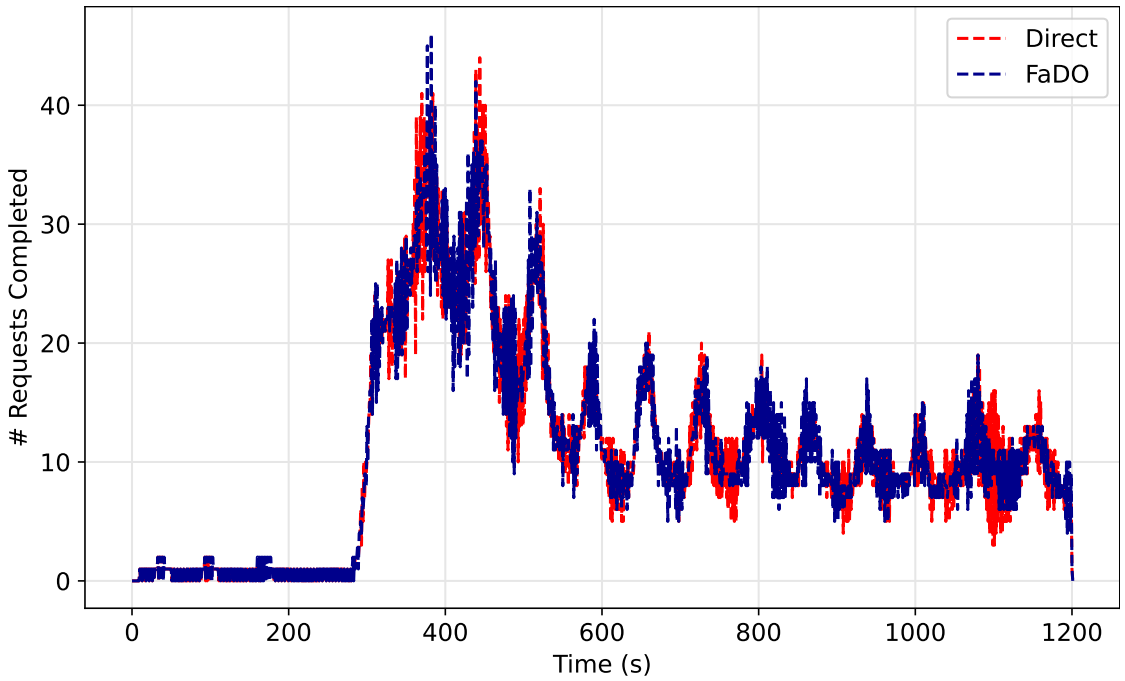


Figure 7.3.: The number of requests completed over time when connecting to the HPC cluster directly and through FaDO. (Load Pattern 1)

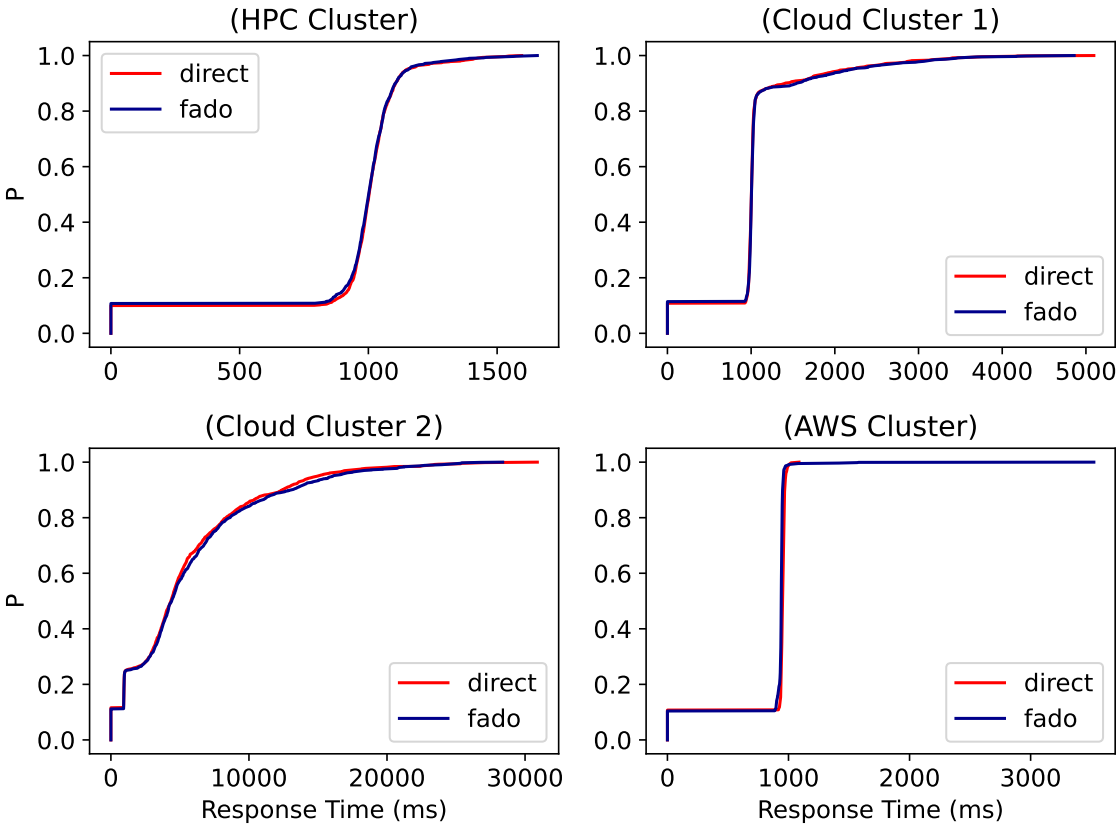


Figure 7.4.: The cumulative distribution functions for the response times recorded when connecting to the different clusters either directly or through FaDO. (Load Pattern 1)

the 1000 millisecond mark. This indicates that the majority of requests took 1000 milliseconds or less.

The direct connection and FaDO connection tests yield virtually the same curve in all four cases. So, not only do these two tests yield very similar numbers of requests over time, but they also have virtually the same response time distribution. This substantial evidence indicates that FaDO's load balancing does not add significant overhead to function invocations.

Figure 7.4's charts further provide insight into the clusters' performances. Indeed, it seems immediately apparent that cloud cluster 2 is not as performant as the others, with a broader distribution of response times. On the other hand, the HPC and AWS clusters appear to be the most performant, with a much tighter response time distribution and virtually all requests complete in about 1000 milliseconds. Finally, though not quite

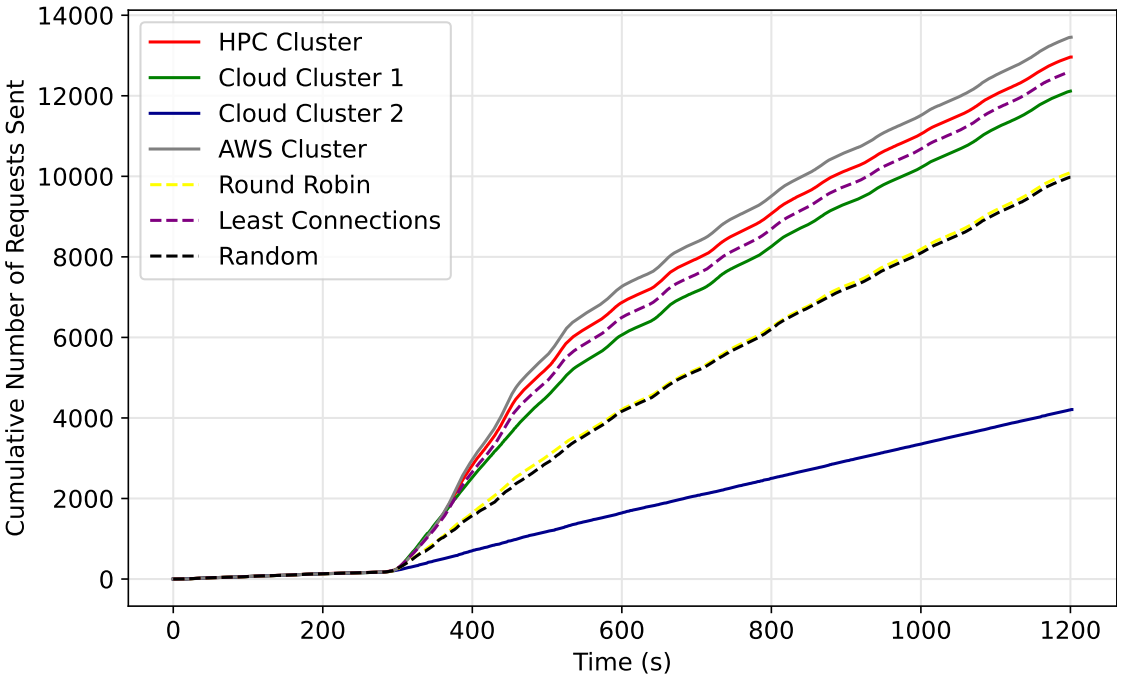


Figure 7.5.: The cumulative sum of completed requests recorded when connecting through FaDO directly to the different clusters or load balancing the requests between the HPC cluster, cloud cluster 1, and cloud cluster 2. (Load Pattern 1)

matching the HPC or AWS clusters, the first cloud cluster is also very good, with about 85% of its requests completed in around 1000 milliseconds.

These deductions from the data align with what we know about the clusters. Indeed, the AWS Lambda functions will scale to handle heavy loads, while the HPC cluster and cloud cluster 1 are both very powerful nodes, with the HPC cluster expected to perform best.

Figure 7.5 gives another view on the cluster performances, relating the cumulative number of requests sent over time. Having established that FaDO's overhead is not significant, the chart now only shows measurements obtained through FaDO. Unsurprisingly, it also shows that the AWS cluster is the most performant, followed closely by the HPC cluster and then the first cloud cluster. The second cloud cluster is, however, far outperformed.

This figure also puts the load balancing policy tests into context. These measurements result from testing FaDO's load balancing with the *least connections*, *round robin*, and *random* selection policies.

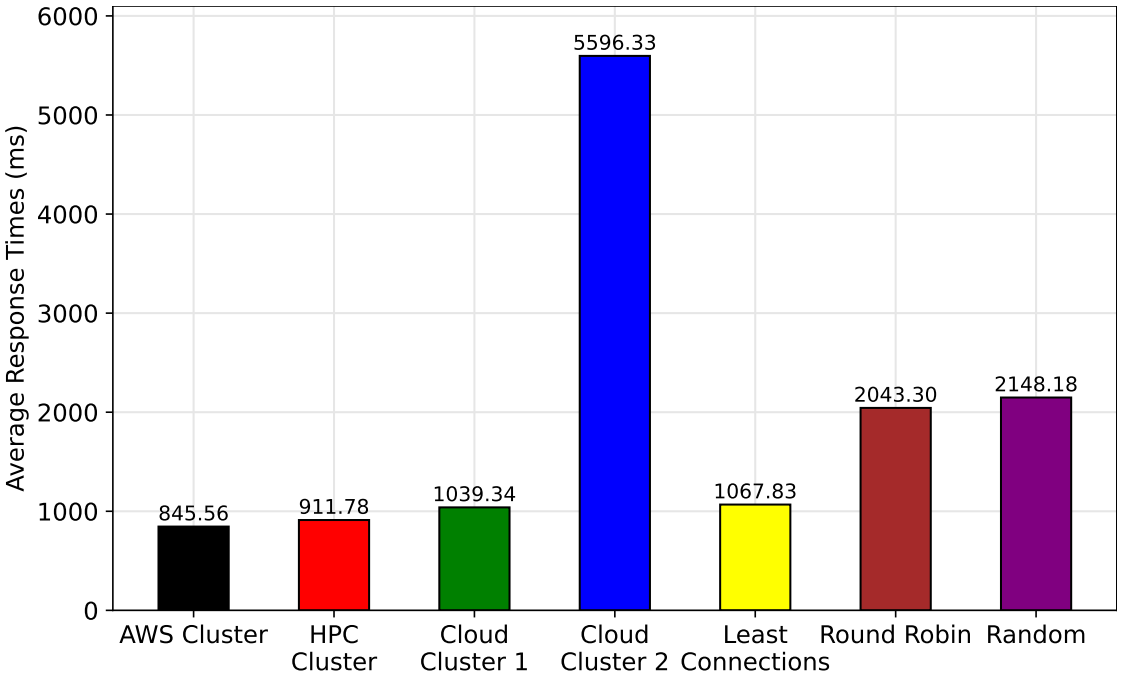


Figure 7.6.: The average response times obtained when connecting through FaDO directly to the different clusters or load balancing the requests between the HPC cluster, cloud cluster 1, and cloud cluster 2. (Load Pattern 1)

The load balancing tests had the load balancer send requests to the HPC and the two cloud clusters. Unfortunately, due to a technical limitation with Caddy, the AWS cluster could not be included as Caddy cannot handle mixed HTTP and HTTPS upstreams. Therefore, the AWS cluster is out of scope when evaluating the different policies.

The *least connections* policy that sends requests to the upstream with the smallest amount of open connections is very performant. In fact, its performance sits between that of the HPC cluster and first cloud cluster and far outperforms the second cloud cluster. Evidently, its policy favors the two most powerful clusters at the second cloud cluster's expense. The *round robin* and *random* policies, however, are virtually identical and notably worse. Though they do still outpace the second cloud cluster.

Figure 7.6 gives a summary of the average response times observed for each policy, be it forwarding requests to a single upstream or using a selection policy to choose from a set of upstreams. The results show that the best throughput is obtained when requests are sent directly to the most powerful cluster. However, the *least connections* policy achieves very comparable performance while also distributing function invocations to other clusters. The *round robin* and *random* policies, while slower than the *least*

connections policy, are effective at mitigating the impact of a slow upstream like the second cloud cluster.

7.5. Reloading Caddy's Configuration

FaDO listens for changes affecting storage buckets and updates the load balancing configuration every time one is created or deleted. Unfortunately, this causes the application to repeatedly reconfigure the load balancer through its configuration API exposed over HTTP.

When the Caddy server receives a new configuration, it will restart its HTTP server to apply the changes. However, it can not do this while handling connections and will wait for an idle period to affect the change.

Caddy reloads very fast, only taking a couple of milliseconds. However, a test was written to assess the impact many reloads in a period would have on the load balancer's throughput. A script was set up to send a reconfiguration request to Caddy every second while *k6* sent function invocations to the *nodeinfo* function using load patterns 1 and 2.

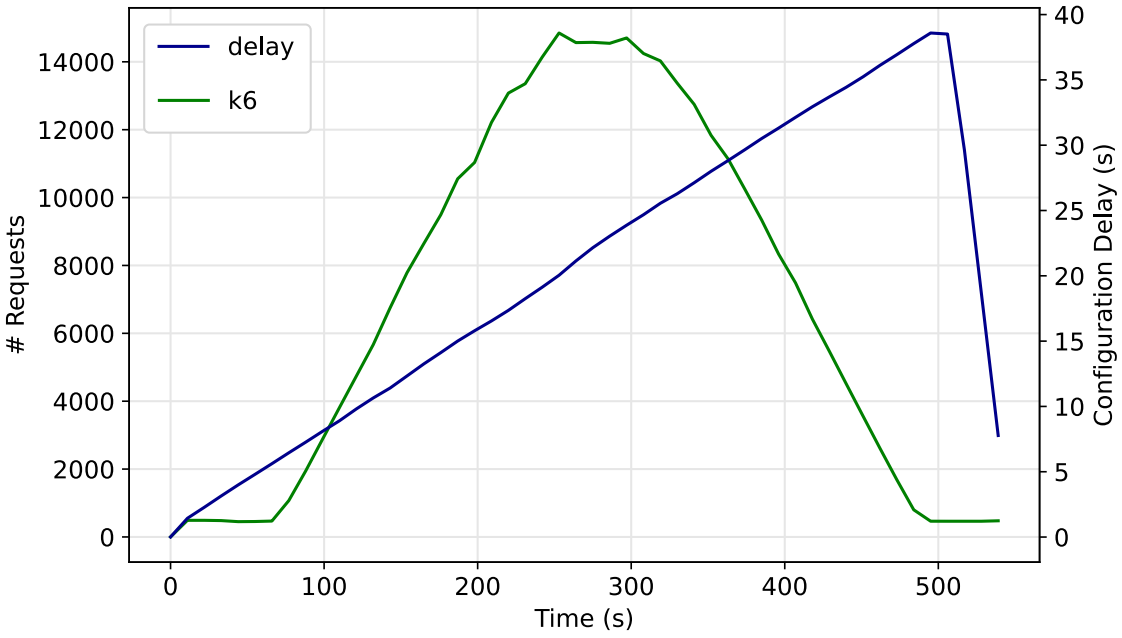


Figure 7.7.: The impact of load on the load balancer's configuration delay observed during a simple k6 test script while concurrently sending reconfiguration requests at 1 second intervals.

The results were unexpected, as Caddy ceased to respond on its administration endpoint. This meant that FaDO could no longer communicate with Caddy, causing failures and necessitating restarting Caddy to fix the issues. However, Caddy was still capable of load balancing requests and was otherwise still operational, so it would never restart independently.

Furthermore, testing showed that the configuration reload time is very sensitive to the load balancer's load. For example, figure 7.7 shows the configuration reload times recorded during a simple load test that ramps up to 30 VUs before going back down to one. The configuration delay increases linearly and does not start recovering until the very end of the test.

These results indicate a limitation on the Caddy administration endpoint, where it cannot handle requests at a high frequency and will cause significant delays in reconfiguration while under load. This indicates that FaDO's load balancing configuration mechanisms are a flaw in its design, as it will cause significant delays and possibly failures. These characteristics are not compatible with latency-sensitive and high-throughput applications.

7.6. Deployment Readiness

While there are issues with its design, FaDO has the makings of a robust network application. Indeed, it is built on top of robust technologies such as the Go language, and it composes itself from highly reliable parts, including the Caddy server and PostgreSQL. Furthermore, it is designed ready for cloud deployment, be it with a simple Docker Compose configuration or using technologies like Kubernetes [Kub]. The application is modular, and its different components can be distributed onto different machines seamlessly. The backend server's HTTP endpoint allows for heartbeat monitoring to detect failures, and its reliance on the database to keep state means that it can be replicated without issue.

However, the application's current implementation poses many problems. One obvious shortcoming stems from the lack of time and manpower necessary to build a highly robust software system of FaDO's nature and is the lack of support for edge cases leading to failures. For instance, the application does not monitor storage deployments or FaaS deployments and cannot tell if they are available or not.

Other issues are more explicit. For example, this is the case with FaDO's reliance on MinIO's *mc* command-line tool, which forces the backend server to interface with the shell, forming a somewhat imperfect integration. This problem is further compounded because the command-line tool does not allow for concurrent invocations, making mutex locks necessary and slowing down the application under load. Alternatives

must therefore be found for bucket mirroring and bucket notification setup tasks to remove the *mc* command-line tool from FaDO's dependencies.

A more straightforward problem lies in the load balancer's conflicting purposes, where the load balancer exposes both the function invocation endpoint and the backend server. However, through its API, the backend server can be called to modify the load balancer's configuration, which triggers the load balancer to restart its HTTP handler. The backend server will wait for the load balancer to respond with its new configuration after it restarted, thus leaving an HTTP connection open. But the load balancer cannot restart its API while a connection is still open, and the application enters into a deadlock. The result is that client connections with the backend server hang and fail to respond with up-to-date data. This issue was patched by responding to requests before reconfiguring the load balancer and accepting that clients need to make a subsequent call to the API to receive the new load balancing configuration. However, it would be solved by having two separate Caddy servers handle the function invocations and the backend server's API.

And beyond those, FaDO's integration with the load balancer exposes a significant point of failure, where too much load may cause the backend server to no longer be able to communicate with the Caddy server.

While these issues are unfortunate, this thesis never intended for FaDO to be an industry-ready product. Instead, it intends to showcase a proof-of-concept application capable of data-aware function scheduling and data placement on a heterogeneous cloud platform. FaDO achieves this and proposes a viable design.

8. Conclusion

This thesis presents the Function and Data Orchestrator (FaDO), a proof-of-concept application that achieves data-aware function placement and automates granular data replication across a heterogeneous serverless cloud platform. The application leverages the cloud's *storage bucket* paradigm, using buckets as the units of replication and the metric for selecting function invocation destinations. Users of the application can thus organize their data into different storage buckets and ensure functions are scheduled close to their data by defining a special HTTP header specifying the required bucket on the function invocation requests.

FaDO intends to fulfill the demands placed on high-throughput network applications subject to heavy workloads. To accomplish this, the application is constructed from four components:

- A PostgreSQL **database** that stores application state, including the distribution of resources and data across the platform and user-defined policies and configurations
- A Caddy Server **load balancer** that manages TLS certificates and terminates HTTPS connections, reverse-proxies function invocations to sets of upstream FaaS endpoints, and exposes the backend server's API
- A bespoke **backend server** written in Go that ties FaDO's different components together and interacts with the platform's different resources, orchestrating storage bucket replication to distribute the data across the platform according to user-defined policies, configuring the load balancer to reverse-proxy function invocation to the FaaS endpoints closest to their requested storage buckets, and allowing user configuration and data manipulations through its API
- A JavaScript single-page **frontend web application** that allows FaDO's users to interact with the data and influence the system's behavior

The frontend client is FaDO's built-in method allowing users to consume the backend server's API. It is served by the backend server itself and is accessible through a web browser by navigating to the server's dedicated HTTPS endpoint prepared by the load balancer. The web application presents FaDO's data visually across multiple web pages

and allows users to submit data to the server using forms, dialogs, and buttons. In this way, users can accomplish the following tasks:

- List the platform's data, including information on the clustered resources, storage bucket and data object locations, and load balancing configuration
- Track FaaS and storage services
- Organize colocated FaaS and storage services into clusters
- Manage storage buckets and data objects across the platform
- Configure bucket replication behavior by specifying the desired number of replicas and defining which replication zones a cluster is in and which replication zones a bucket is allowed to exist in
- Override replication decisions by manually specifying the set of storage deployments a bucket should be replicated to
- Configure the load balancer's default upstream selection policy and the HTTP header name used to determine the requested storage bucket
- Override the load balancing route settings for a given storage bucket by manually specifying the set of upstreams and selection policy

Load testing results indicate that the application is capable of high-performance data-aware function scheduling through HTTP header matching. Indeed, Caddy's reverse-proxying adds negligible overhead to function invocations and effectively optimizes completion times using the "least connections" load balancing policy where incoming requests are sent upstream with the least open connections.

Moreover, the application successfully automates granular bucket-scoped replication on MinIO storage deployments. Storage buckets are either read/write master copies or read replicas, and changes affecting a master copy cause its MinIO service to send notifications back to FaDO. Upon receiving these notifications, FaDO identifies which data was affected and propagates the changes to all concerned bucket replicas using MinIO's command-line client. This client behaves similarly to the popular *rsync* tool and avoids sending a bucket's entire content by identifying the differences between out-of-sync buckets and transmitting only necessary data.

The replication mechanisms abide by user-defined policies and attempt to create the desired number of replicas for a given bucket while respecting zonal constraints imposed on its data. In so doing, these mechanisms spread data around the cloud platform and allow users to distribute function invocations across different compute

resources while also allowing them to impose location constraints on their data, such as can be necessitated by privacy laws like the General Data Protection Regulation (GDPR).

In conclusion, this thesis presents FaDO, a proof-of-concept application that confirms the viability of systems like the FDN [Jin+21] for data-aware function scheduling in multi-FaaS cloud platforms. Furthermore, this project proposes an atypical mechanism for granularly replicating data across a platform containing multiple storage services and implements a virtual storage layer where data from multiple locations can be manipulated through a unified interface.

However, FaDO does show some weaknesses. Indeed, it relies on MinIO and its command-line tool and centralized replication mechanisms, preventing concurrent transfers and forcing data through the control plane. Also, its integration with the Caddy load balancer is prone to breaking under load. What is more, the load balancing method, while performant, is also somewhat limited, imposing homogeneous function paths on users and preventing them from defining a mixed set of HTTP and HTTPS upstreams. Thus, there is much opportunity for further work to improve upon FaDO's current design and implementation.

8.1. Future Work

While FaDO is a successful prototype, the short timeframe for its implementation inevitably means that the application is not up to the standards of industry network applications. Indeed, many edge cases remain unaddressed, which leads to unexpected behavior and failures. Therefore, an obvious avenue for future work is solidifying FaDO internal logic to produce a more fault-tolerant system.

Another straightforward direction lies in increasing the amount of information FaDO gathers from its cloud platform and using it for more powerful and nuanced decision-making. At this time, the application only keeps minimal metadata concerning storage deployments and none concerning FaaS deployments beyond an HTTP endpoint. However, the application could collect such information as storage usage to implement storage quota policies or hardware details to limit demanding functions to capable machines.

Beyond these options, it will be useful to address FaDO's current limitations. Namely its reliance on MinIO as a storage technology and its simplistic function invocation load balancing.

8.1.1. FaDO and Storage Technologies

The application is currently only capable of interacting with MinIO storage services. This is because much of the backend server logic is specifically dedicated to integrating with MinIO deployments, and the application relies on MinIO's command-line client to execute replication tasks.

Now, MinIO does claim to be compatible with AWS S3 storage technologies and therefore suggests the possibility of an S3-compatible FaDO. However, this would still pose challenges for multi-cloud use-cases, as users might be constrained to non-S3 object-storage technologies.

To tackle multi-cloud use-cases and support different storage technologies, FaDO's architecture could be extended to include *storage adaptors*. Different storage adaptors would be created for different storage technologies, and they would hide the specificities of the different solutions behind a unified storage API.

This strategy would remove code from the backend server specific to particular storage technologies, making the program more general and diminishing its responsibilities and thus also its points of failure. It would also allow for decentralized bucket replication, avoiding any central bottleneck and thus improving upon FaDO's current use of MinIO's client-side replication facilities.

Such an architecture would, however, make replication tasks more complex, as it would no longer be possible to rely on tools like MinIO's command-line utility for efficient and easy bucket mirroring.

8.1.2. FaDO and Load Balancing

Thanks to Caddy, FaDO's load balancing mechanism is robust and performant. However, it is also very simplistic. Upstream selection is solely made using an HTTP header to match a storage bucket, and requests are reverse-proxied as-is to the target FaaS platforms. This means users must organize their functions to be invoked at a consistent path across the different FaaS services and can only influence function placement by specifying the required storage bucket.

Future work on FaDO's load balancing could extend the decision-making process to include more metrics than simply the target storage bucket. For instance, function invocations could include hardware requirements or zonal policies. Furthermore, the load balancer could account for the same function's different trigger paths on different FaaS endpoints.

While the configuration generation logic can be extended to add features like path rewriting and multi-variate matching, the resulting configuration would be dramatically more complex as it needs to be exhaustive. Moreover, such a configuration removes

any possibility for dynamic behavior.

A possible remedy would be to implement custom load balancer control plane extensions. This is a common feature among technologies like Caddy and NGINX, and Caddy allows this by allowing users to load custom compiled modules written in Go. These mechanisms would allow for much more complex scheduling mechanisms and remove the need for FaDO to send unreliable configuration requests.

The FDN's proposal includes monitoring different aspects of the platform, including function completions on the different FaaS servers. This information is used to create behavioral models useful for fine-tuning the scheduling process. Load balancer control plane extensions may be a valuable tool for creating a more capable scheduler and monitoring function behavior.

A. Database Schema

A.1. Tables

```
CREATE TABLE policies (
  policy_id              serial       PRIMARY KEY,
  name                   text         NOT NULL UNIQUE,
  default_value          jsonb        NOT NULL
);
```

Listing A.1.: Database schema definition for the `policies` table.

```
CREATE TABLE global_policies (
  policy_id              serial       NOT NULL UNIQUE
                                      REFERENCES policies
                                      ON DELETE CASCADE,
  value                  jsonb        NOT NULL
);
```

Listing A.2.: Database schema definition for the `global_policies` table.

```
CREATE TABLE clusters (
  cluster_id             serial       PRIMARY KEY,
  name                   text         NOT NULL UNIQUE
);
```

Listing A.3.: Database schema definition for the `clusters` table.

```
CREATE TABLE clusters_policies (
  cluster_id             int          NOT NULL REFERENCES clusters
                                      ON DELETE CASCADE,
  policy_id              int          NOT NULL REFERENCES policies
                                      ON DELETE CASCADE,
  value                  jsonb        NOT NULL,

  UNIQUE (cluster_id, policy_id)
);
```

Listing A.4.: Database schema definition for the `clusters_policies` join table.

```
CREATE TABLE faas_deployments (
  faas_id                serial       PRIMARY KEY,
  cluster_id             int          NOT NULL REFERENCES clusters
                                      ON DELETE CASCADE,
  url                    text         NOT NULL UNIQUE
);
```

Listing A.5.: Database schema definition for the `faas_deployments` table.

```
CREATE TABLE storage_deployments (
  storage_id             serial       PRIMARY KEY,
  cluster_id             int          NOT NULL REFERENCES clusters
                                      ON DELETE CASCADE,
  minio_deployment_id    text         NOT NULL UNIQUE,
  alias                  text         NOT NULL UNIQUE,
  endpoint               text         NOT NULL UNIQUE,
  access_key             text         NOT NULL,
  secret_key             text         NOT NULL,
  use_ssl                boolean      NOT NULL,
  sqs_arn                text         NOT NULL,
  management_url         text         NOT NULL DEFAULT ''
);
```

Listing A.6.: Database schema definition for the `storage_deployments` table.

```
CREATE TABLE buckets (
  bucket_id              serial       PRIMARY KEY,
  storage_id             int          NOT NULL
                                      REFERENCES storage_deployments
                                      ON DELETE CASCADE,
  name                   text         NOT NULL UNIQUE
);
```

Listing A.7.: Database schema definition for the `buckets` table.

```
CREATE TABLE buckets_policies (
  bucket_id              int          NOT NULL REFERENCES buckets
                                      ON DELETE CASCADE,
  policy_id              int          NOT NULL REFERENCES policies
                                      ON DELETE CASCADE,
  value                  jsonb        NOT NULL,

  UNIQUE (bucket_id, policy_id)
);
```

Listing A.8.: Database schema definition for the `buckets_policies` join table.

```
CREATE TABLE replica_bucket_locations (
  bucket_id              int          NOT NULL REFERENCES buckets
                                      ON DELETE CASCADE,
  storage_id             int          NOT NULL
                                      REFERENCES storage_deployments
                                      ON DELETE CASCADE,

  UNIQUE (bucket_id, storage_id)
);
```

Listing A.9.: Database schema definition for the `replica_bucket_locations` join table.

```
CREATE TABLE objects (
  object_id              serial        PRIMARY KEY,
  bucket_id              int           NOT NULL REFERENCES buckets
                                       ON DELETE CASCADE,
  name                   text          NOT NULL,

  UNIQUE (bucket_id, name)
);
```

Listing A.10.: Database schema definition for the `objects` table.

A.2. Views

```
CREATE VIEW buckets_faas_deployments AS
  SELECT x.bucket_id, x.bucket_name, array_agg(x.faas_id) AS faas_ids,
    array_agg(x.faas_url) AS faas_urls
  FROM (
    SELECT b.bucket_id, b.name AS bucket_name, fd.faas_id,
      fd.url AS faas_url
    FROM buckets b
      INNER JOIN replica_bucket_locations rbl
        ON rbl.bucket_id = b.bucket_id
      LEFT JOIN storage_deployments sd
        ON sd.storage_id = rbl.storage_id
      LEFT JOIN clusters c ON c.cluster_id = sd.cluster_id
      LEFT JOIN faas_deployments fd ON fd.cluster_id = c.cluster_id
    UNION
    SELECT b.bucket_id, b.name AS bucket_name, fd.faas_id,
      fd.url AS faas_url
    FROM buckets b
      LEFT JOIN storage_deployments sd ON sd.storage_id = b.storage_id
      LEFT JOIN clusters c ON c.cluster_id = sd.cluster_id
      LEFT JOIN faas_deployments fd ON fd.cluster_id = c.cluster_id
  ) AS x GROUP BY x.bucket_id, x.bucket_name;
```

Listing A.11.: Database schema definition for the `buckets_faas_deployments` view linking storage buckets to FaaS endpoints.

```
CREATE VIEW existing_bucket_locations AS
  SELECT bucket_id, array_agg(storage_id) AS storage_ids
  FROM replica_bucket_locations
  GROUP BY bucket_id;
```

Listing A.12.: Database schema definition for the `existing_bucket_locations` view summarizing buckets' storage service locations.

```
CREATE VIEW bucket_replications AS
  SELECT rbl.bucket_id, b.name AS bucket_name,
    b.storage_id AS src_storage_id, src_sd.alias AS src_storage_alias,
    src_sd.minio_deployment_id AS src_minio_deployment_id,
    rbl.storage_id AS dst_storage_id, dst_sd.alias AS dst_storage_alias,
    dst_sd.minio_deployment_id AS dst_minio_deployment_id
  FROM replica_bucket_locations rbl
  LEFT JOIN buckets b ON b.bucket_id = rbl.bucket_id
  LEFT JOIN storage_deployments src_sd
    ON src_sd.storage_id = b.storage_id
  LEFT JOIN storage_deployments dst_sd
    ON dst_sd.storage_id = rbl.storage_id;
```

Listing A.13.: Database schema definition for the `bucket_replications` view listing all storage bucket master-replica replication pairs.

A.3. Initial Data

```
INSERT INTO policies (name, default_value)
VALUES
  ('lb_policy',             '"round_robin"'),
  ('lb_match_header',       '"X-FaDO-Bucket"'),
  ('lb_upstreams',          '[]'),
  ('lb_routes',             '{}'),
  ('lb_route_overrides',    '{}'),
  ('caddy_config',          '""'),
  ('replica_locations',     '[]'),
  ('target_replica_count',  '0'),
  ('zones',                 '[]');
```

Listing A.14.: SQL command inserting the initial policy data.

B. FaDO Client UI

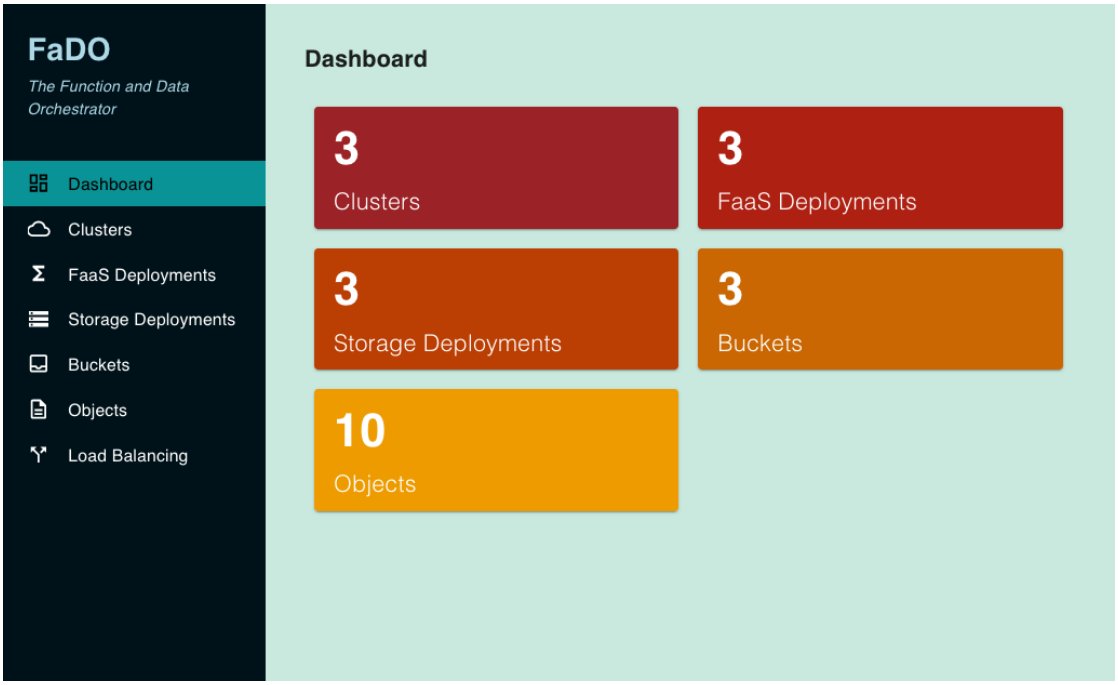


Figure B.1.: The FaDO frontend client's dashboard page.

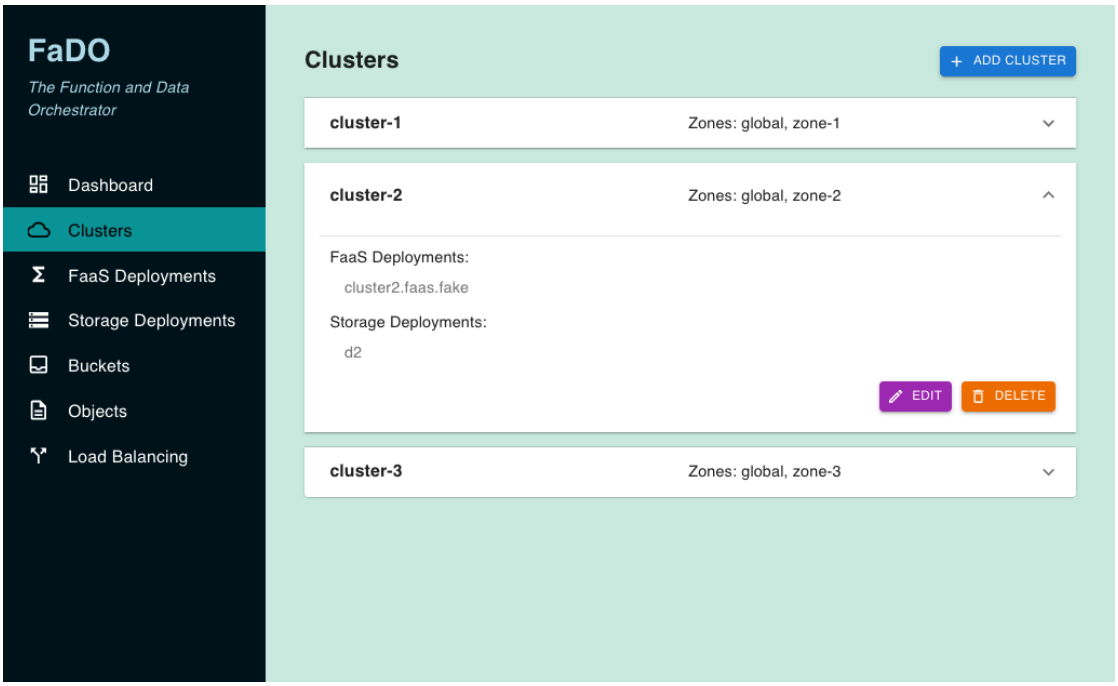


Figure B.2.: Listing the platform's clusters in FaDO's frontend client.

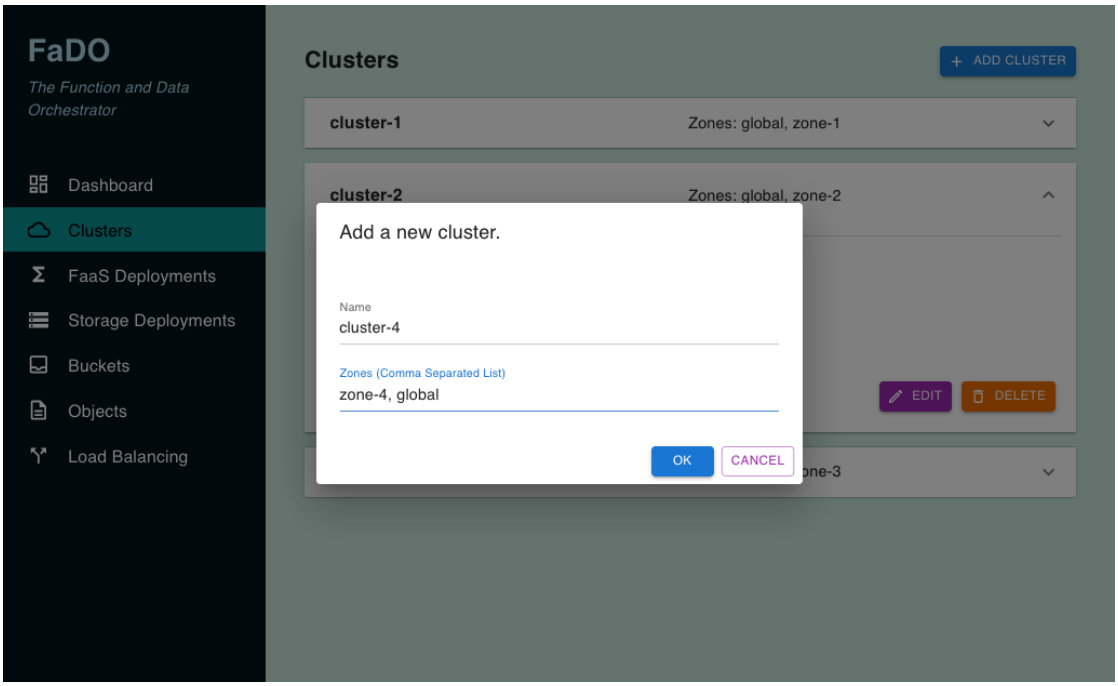


Figure B.3.: Adding a new cluster using FaDO's frontend client.

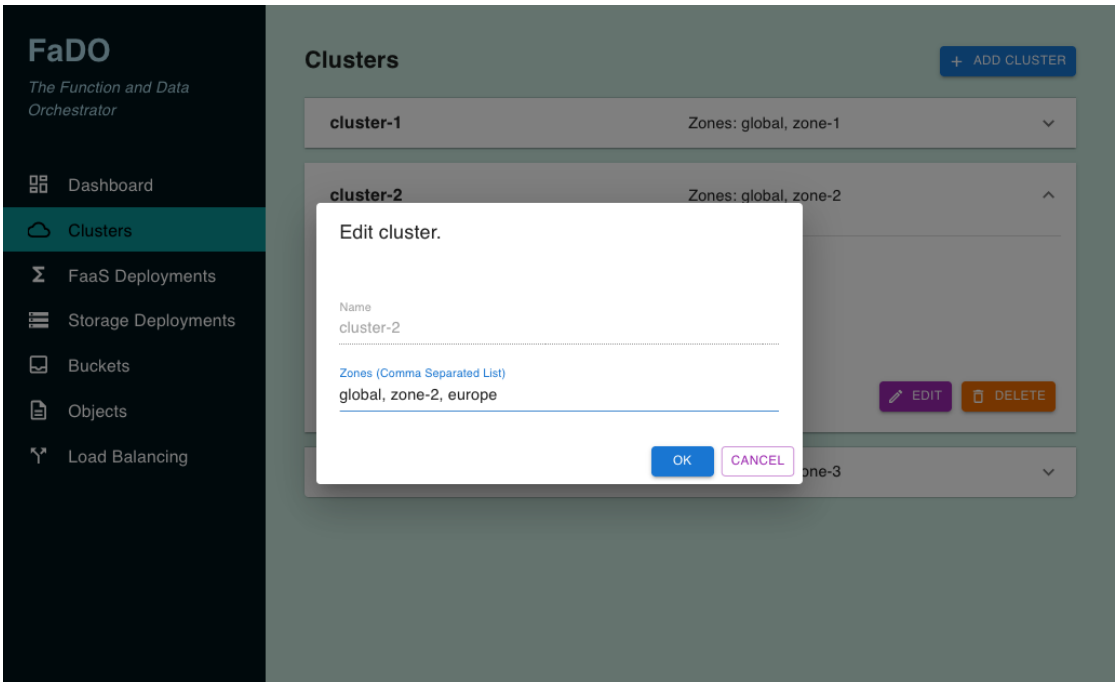


Figure B.4.: Editing a cluster using FaDO's frontend client.

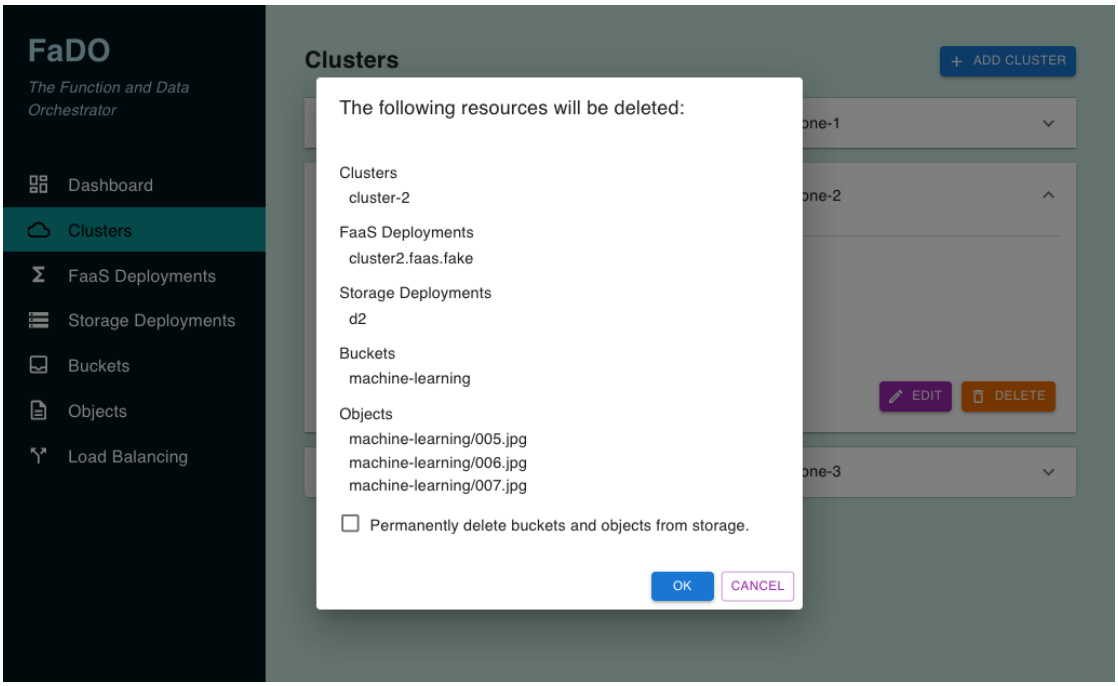


Figure B.5.: Deleting a cluster using FaDO's frontend client.

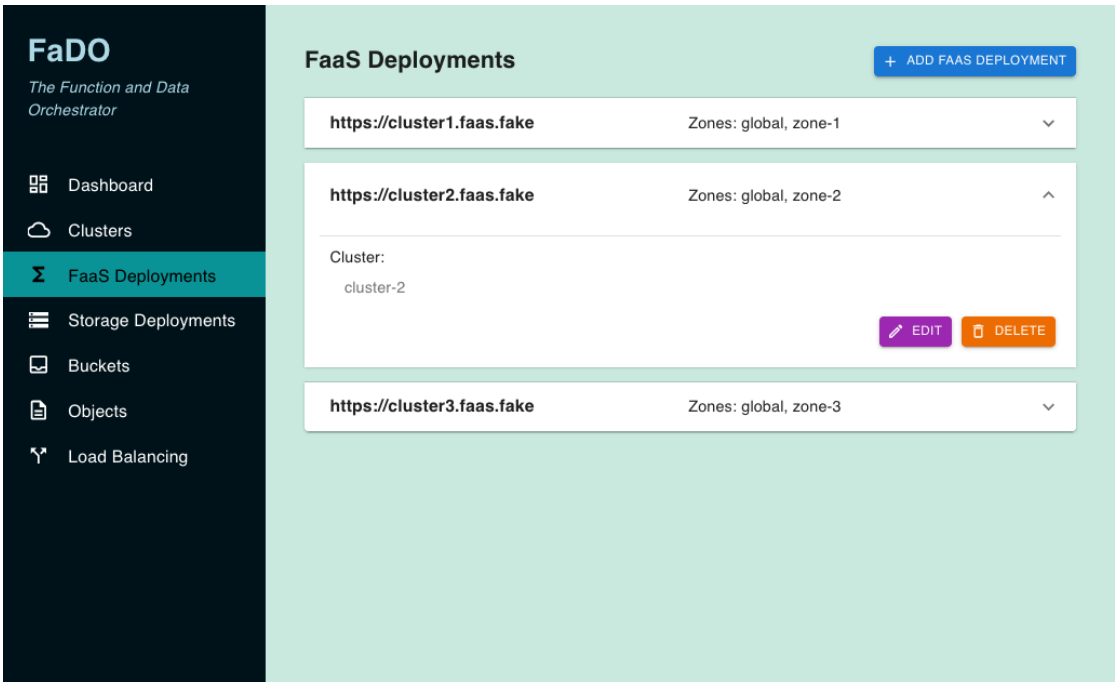


Figure B.6.: Listing the platform's FaaS deployments in FaDO's frontend client.

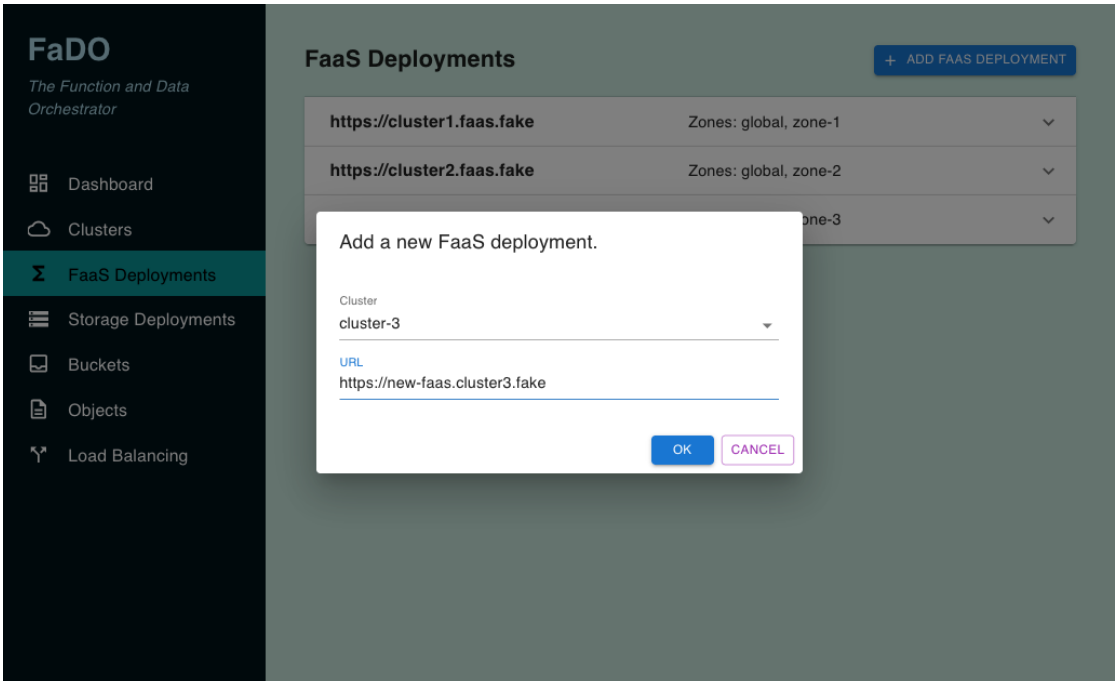


Figure B.7.: Tracking a new FaaS endpoint using FaDO's frontend client.

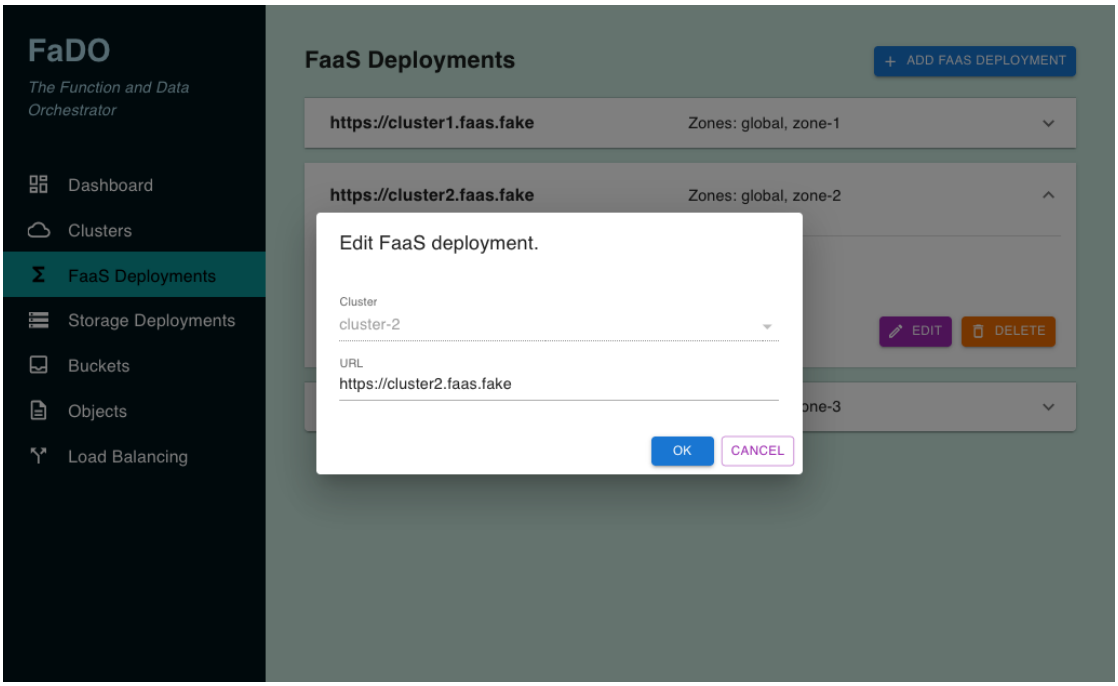


Figure B.8.: Editing a FaaS endpoint using FaDO's frontend client.

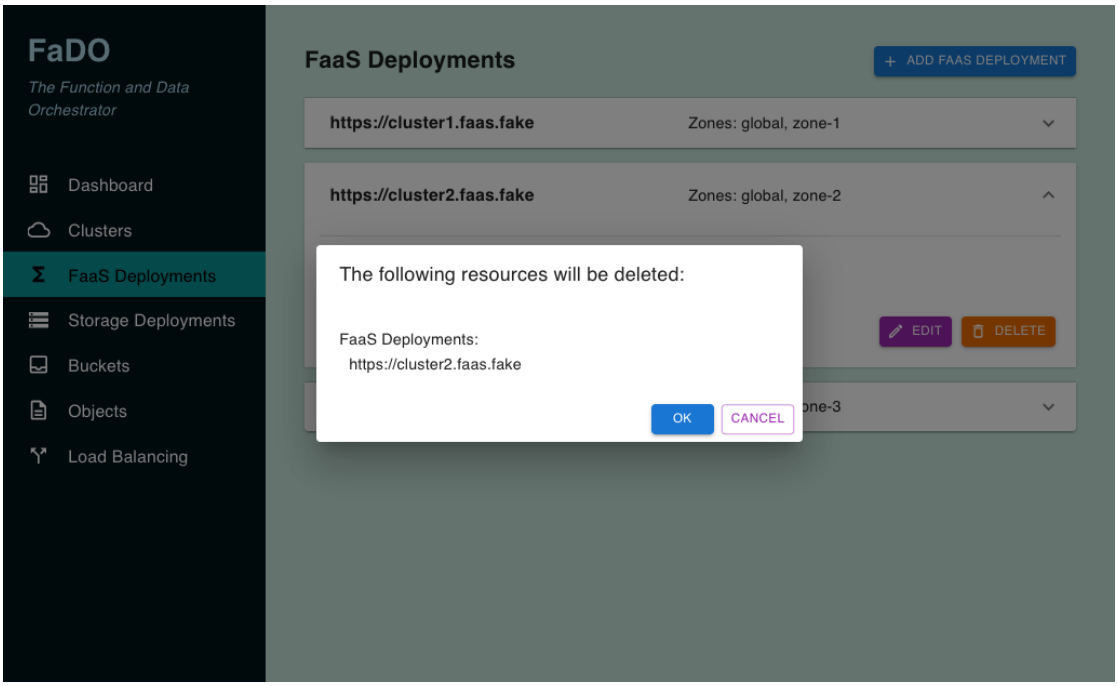


Figure B.9.: Deleting a FaaS endpoint using FaDO's frontend client.

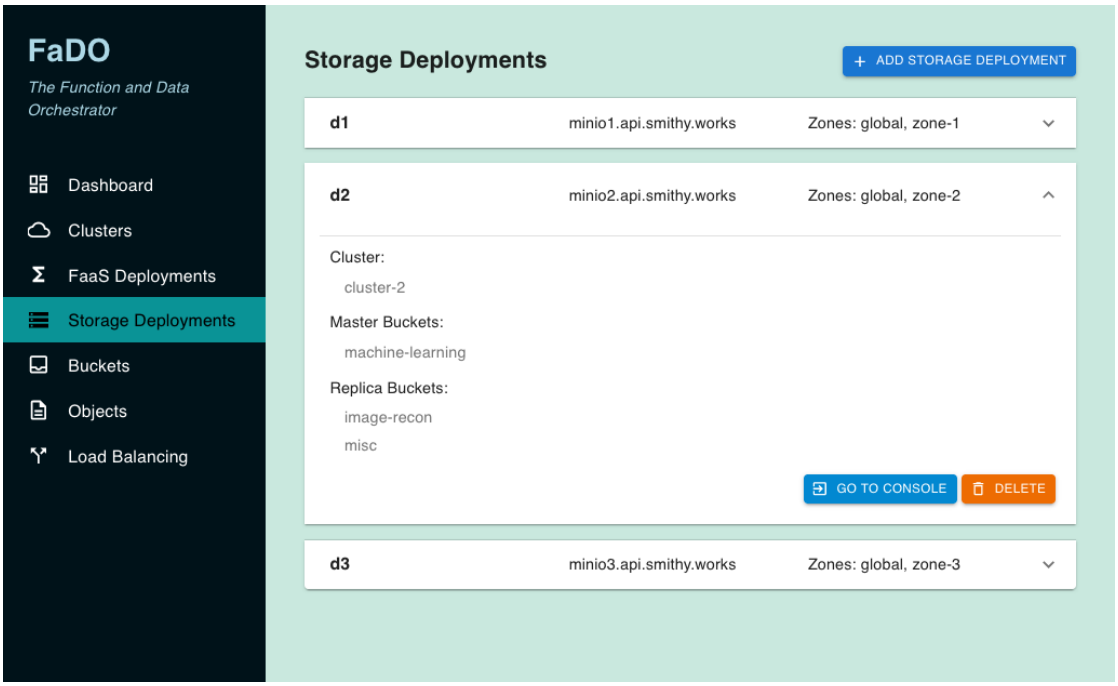


Figure B.10.: Listing the platform's storage deployments in FaDO's frontend client.

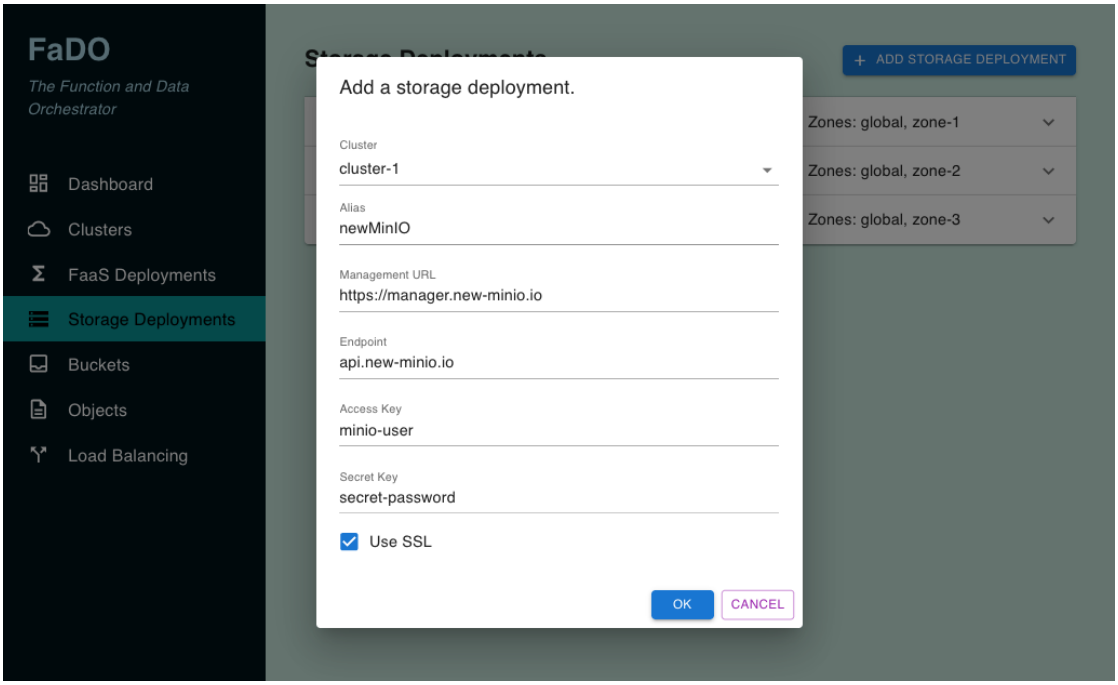


Figure B.11.: Tracking a new storage deployment using FaDO's frontend client.

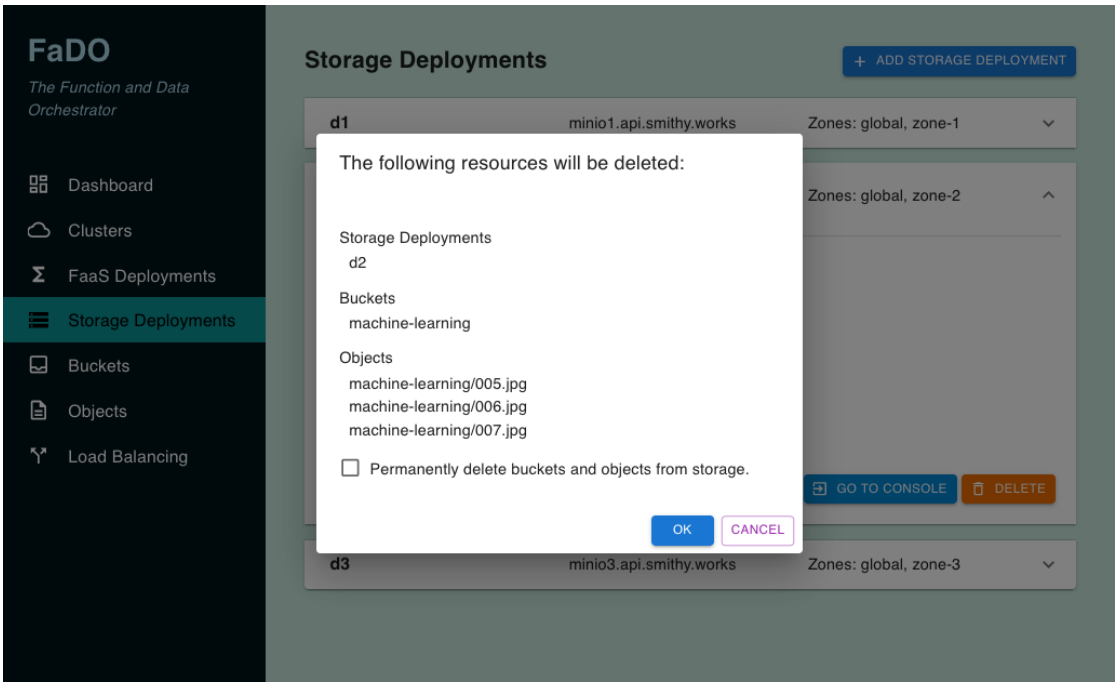


Figure B.12.: Removing a storage deployment using FaDO's frontend client.

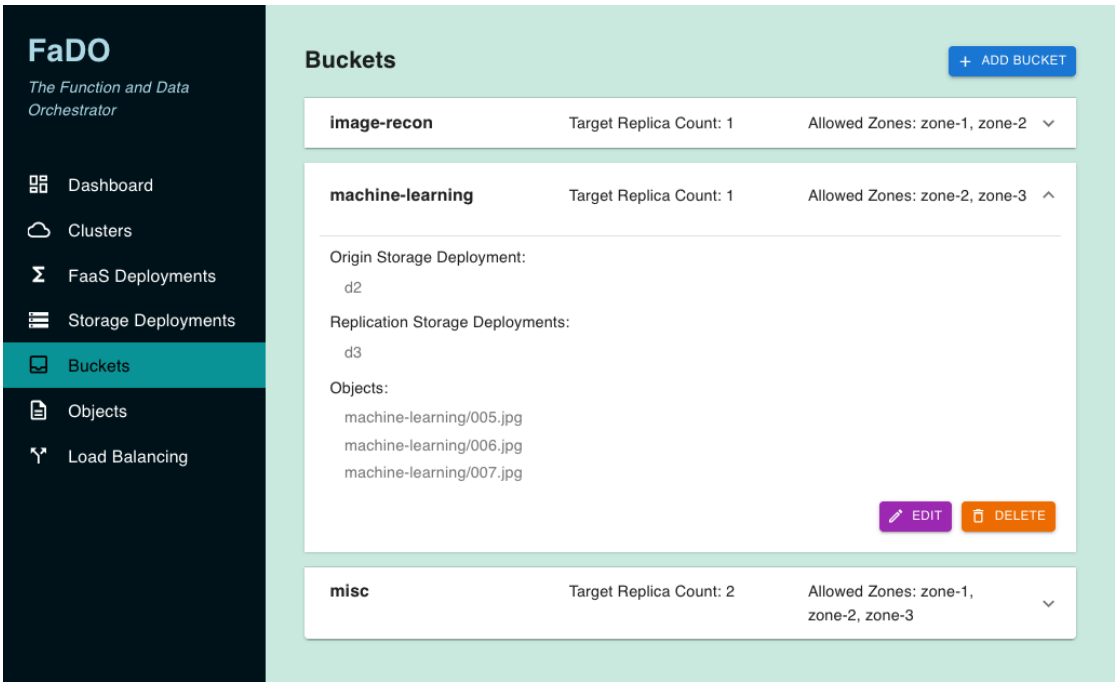


Figure B.13.: Listing the platform's storage buckets in FaDO's frontend client.

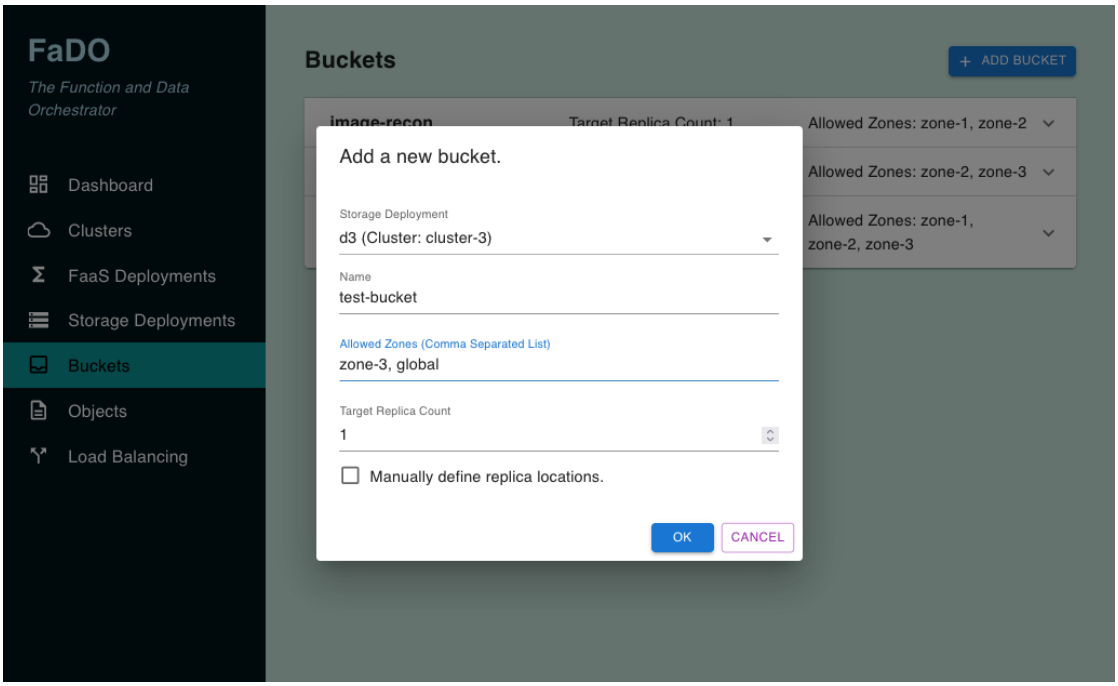


Figure B.14.: Adding a new storage bucket using FaDO's frontend client.

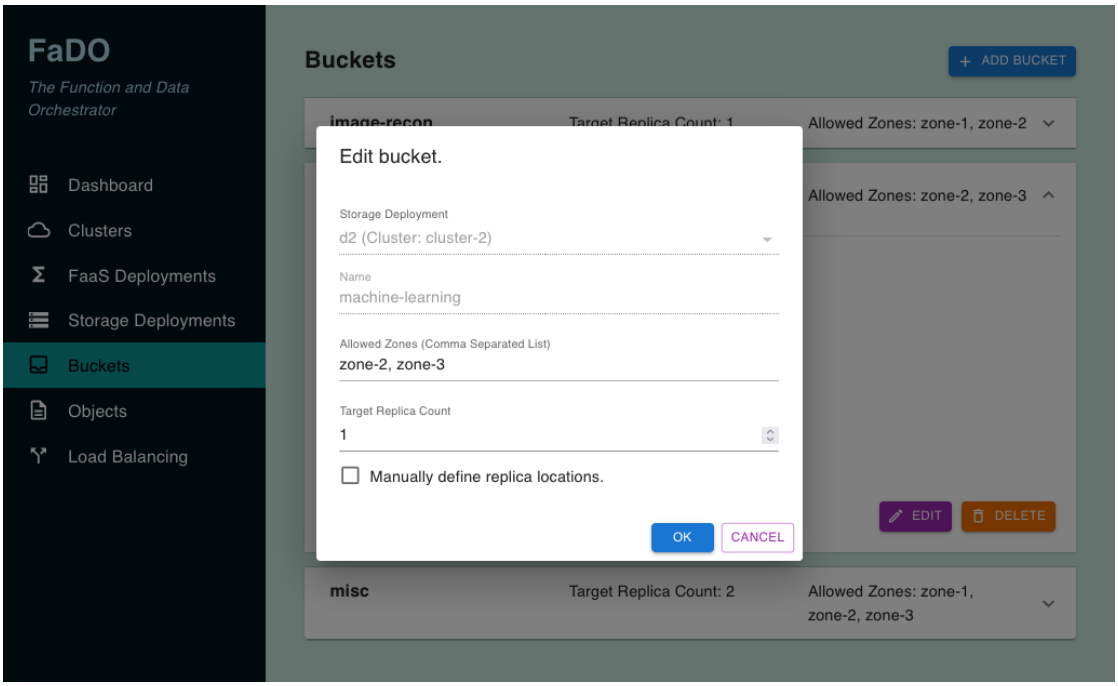


Figure B.15.: Editing a storage bucket using FaDO's frontend client.

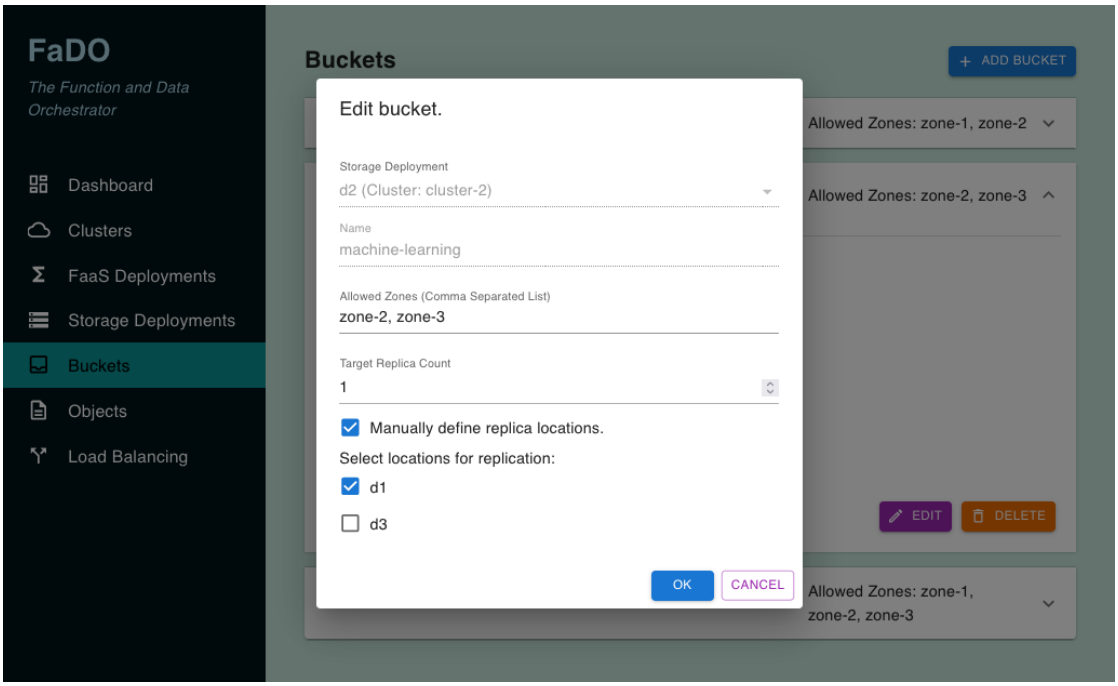


Figure B.16.: Overriding a storage bucket's replication using FaDO's frontend client.

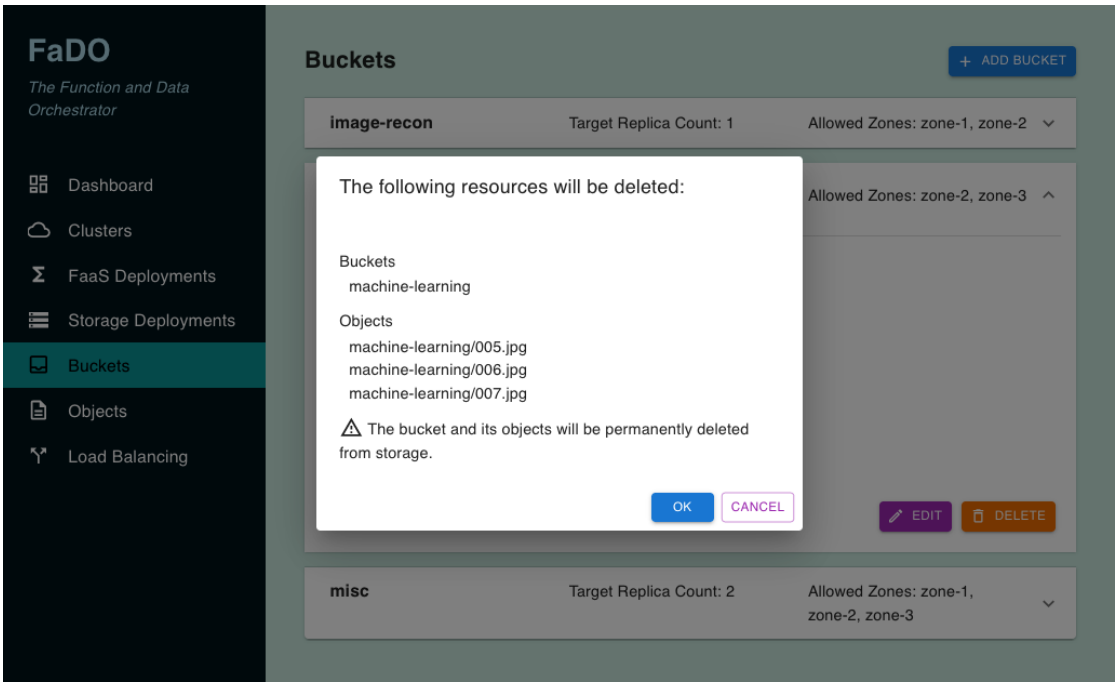


Figure B.17.: Deleting a storage bucket using FaDO's frontend client.

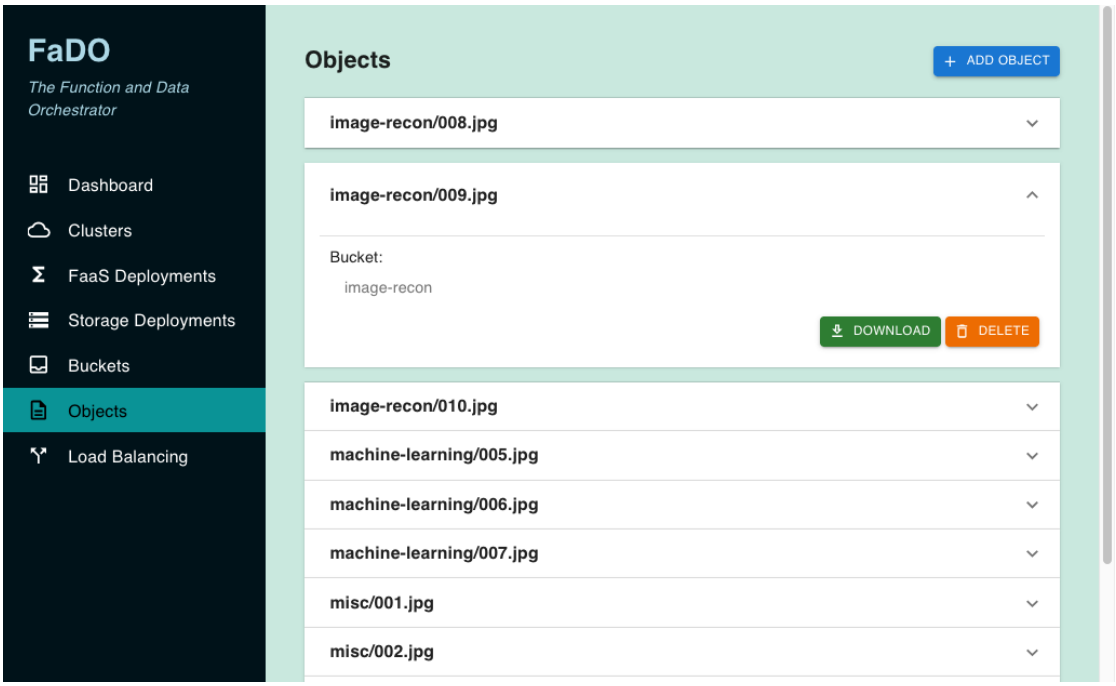


Figure B.18.: Listing the platform's data objects in FaDO's frontend client.

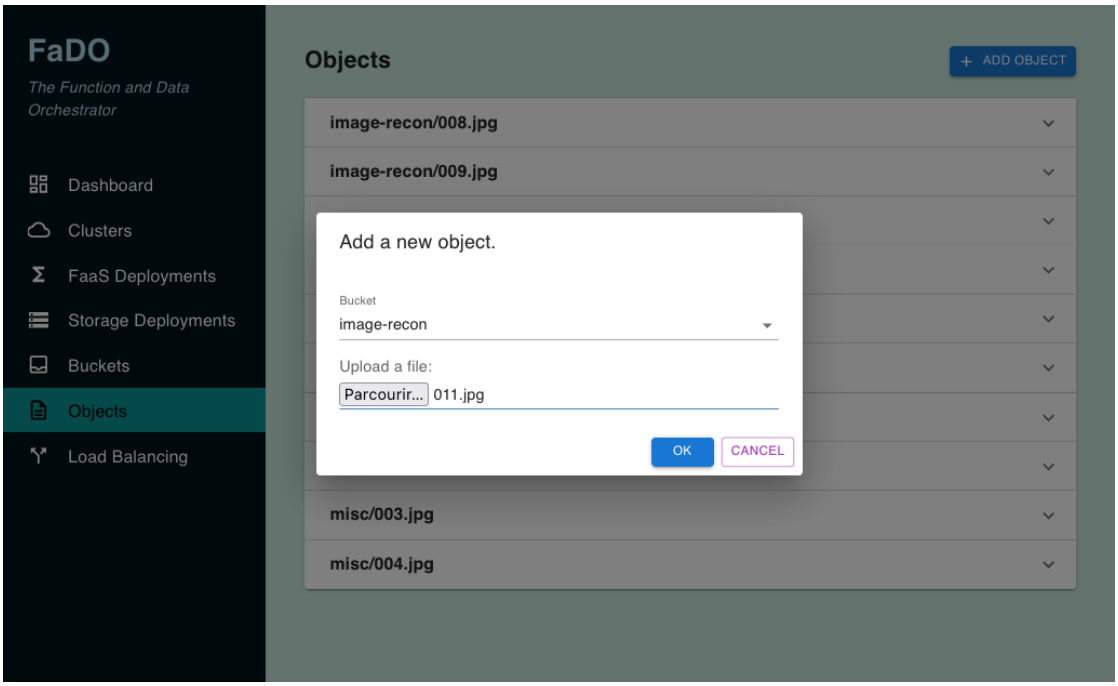


Figure B.19.: Uploading a data objects using FaDO's frontend client.

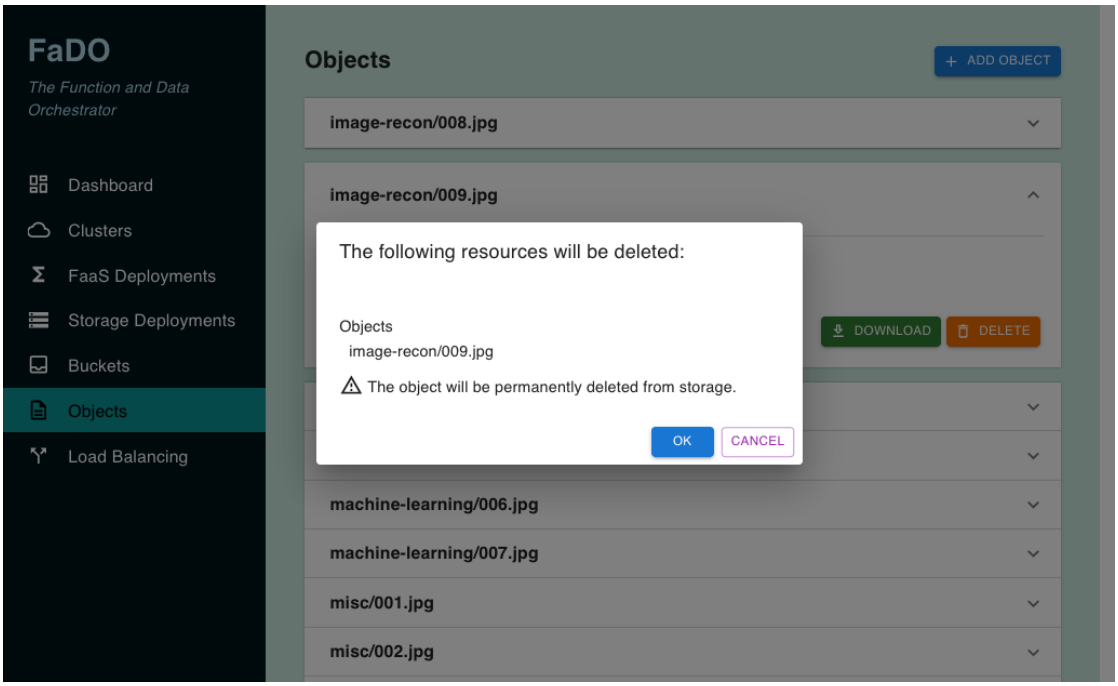


Figure B.20.: Deleting a data objects using FaDO's frontend client.

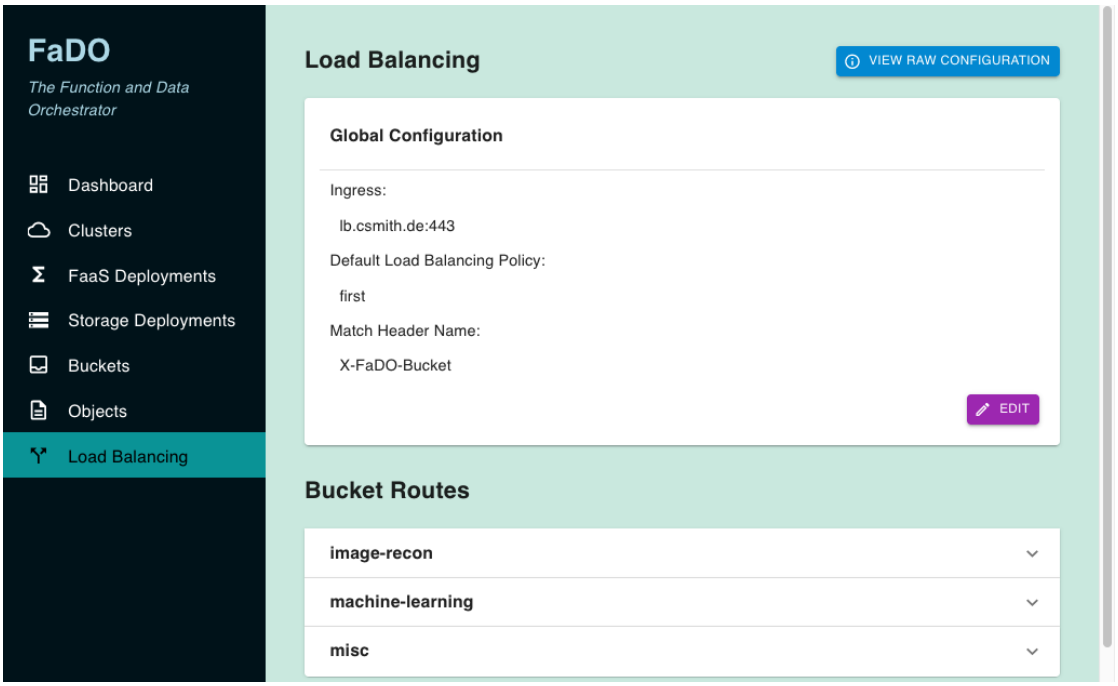


Figure B.21.: Displaying the load balancer's global settings in FaDO's frontend client.

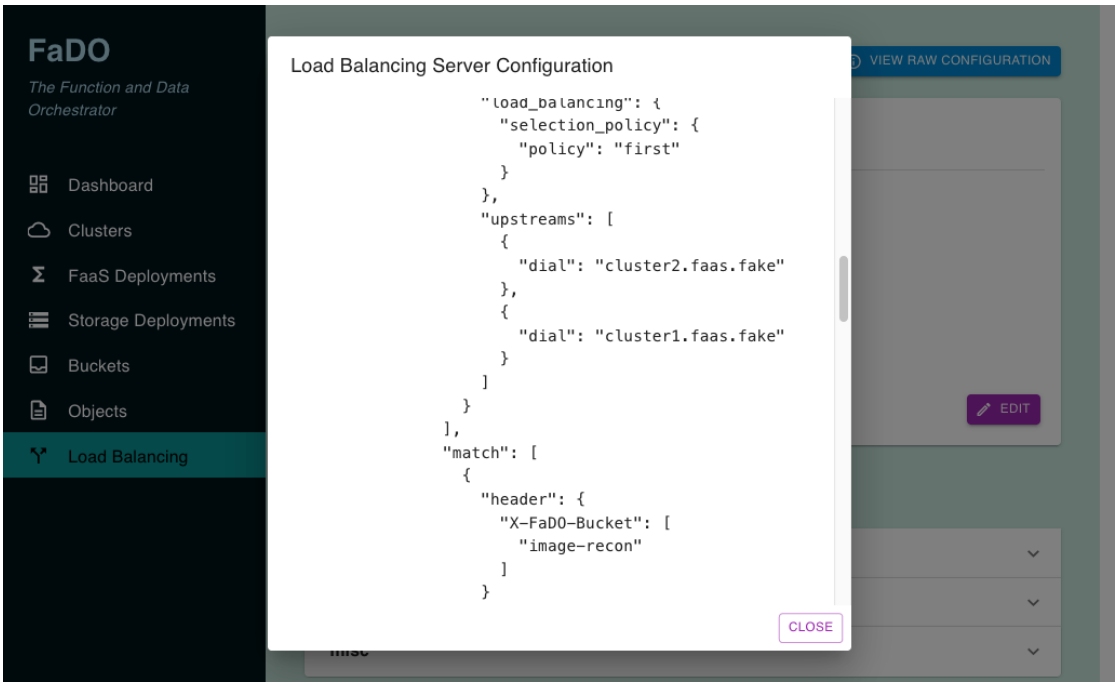


Figure B.22.: Viewing the load balancer's JSON configuration in FaDO's frontend client.

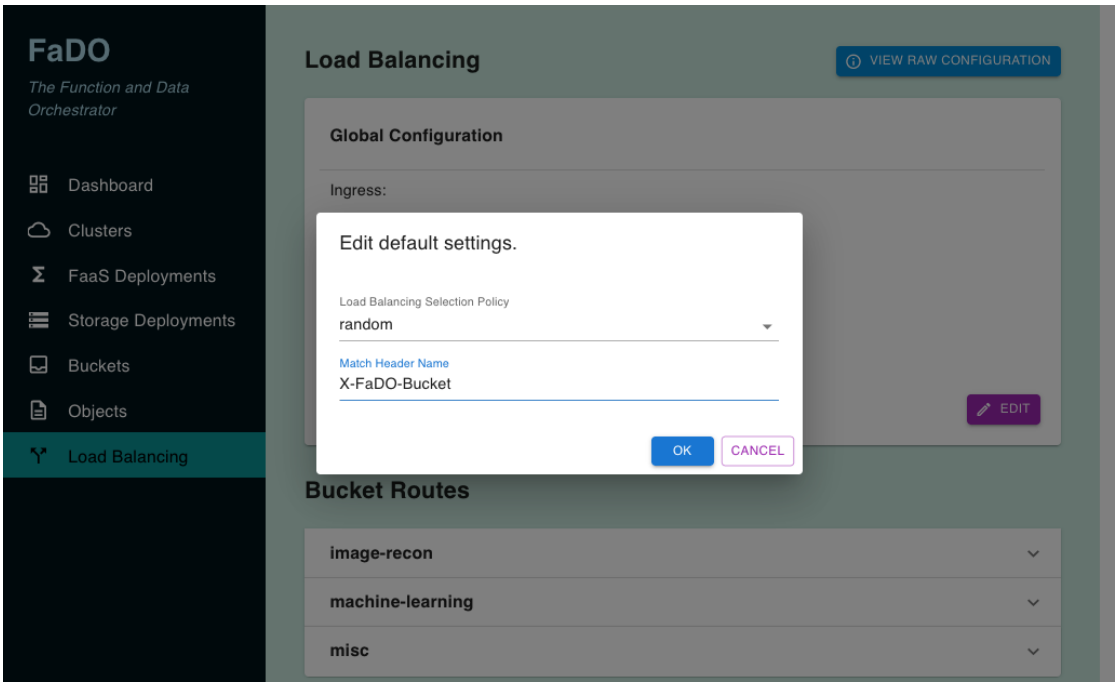


Figure B.23.: Editing the load balancer's global settings in FaDO's frontend client.

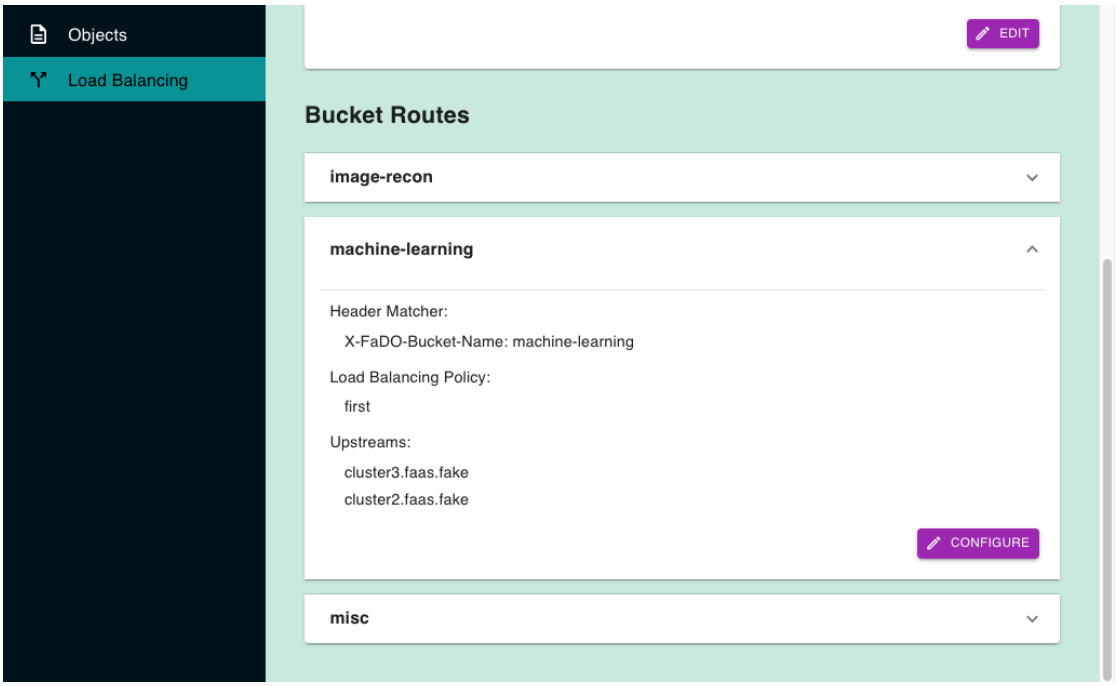


Figure B.24.: Displaying the load balancer's route settings in FaDO's frontend client.

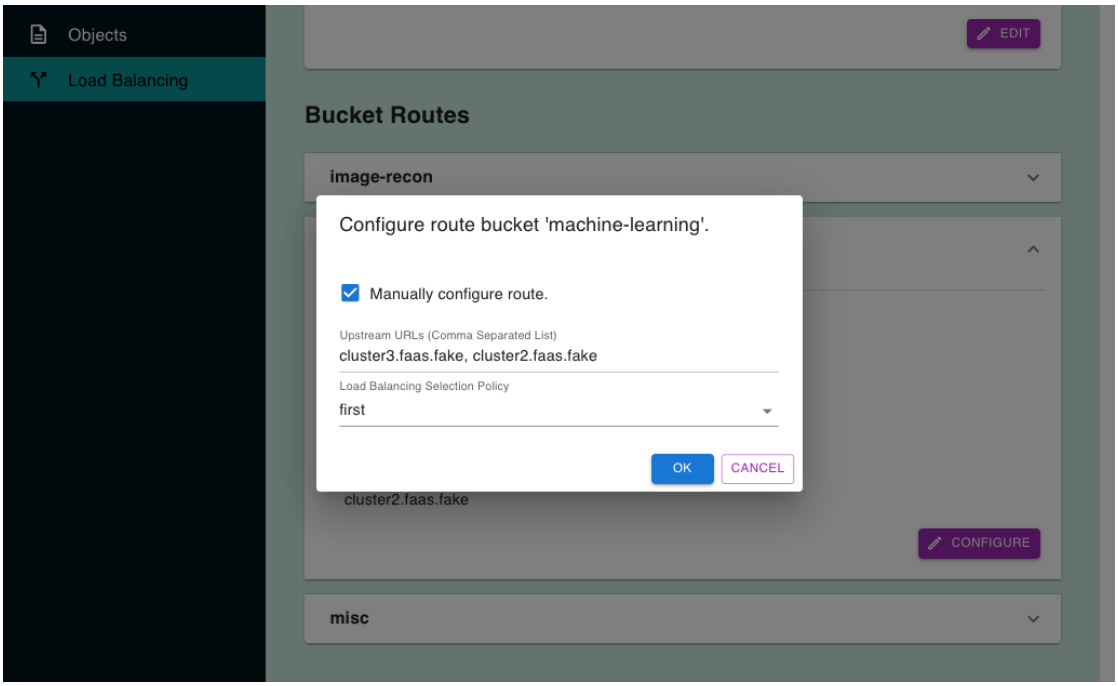


Figure B.25.: Overriding a storage bucket's load balancing route settings in FaDO's frontend client.

C. Supplemental Results

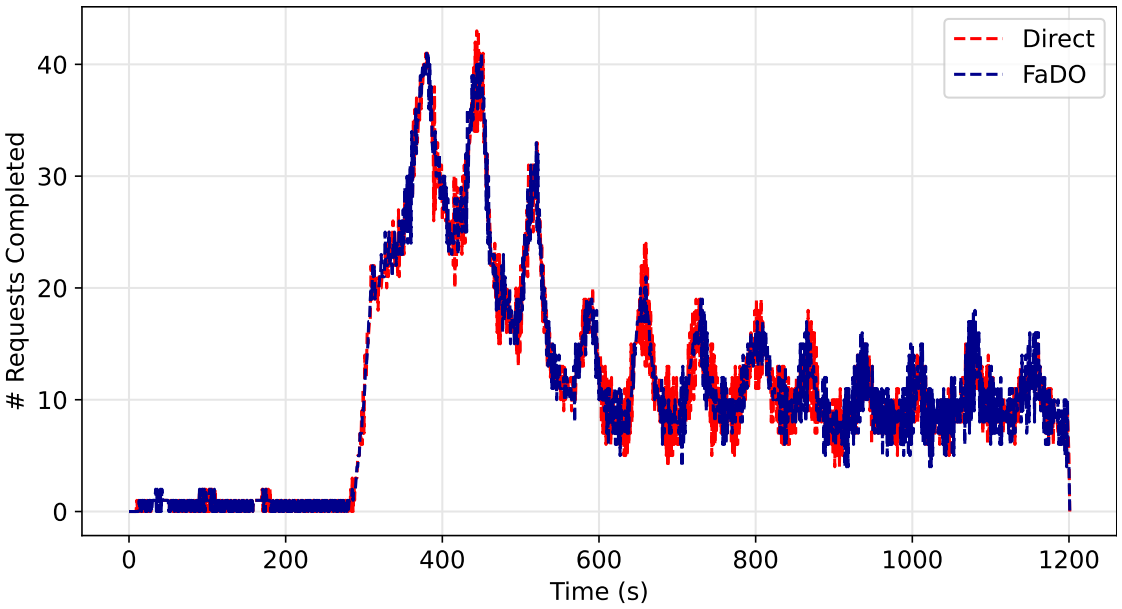


Figure C.1.: The number of requests completed over time when connecting to the AWS cluster directly and through FaDO. (Load Pattern 1)

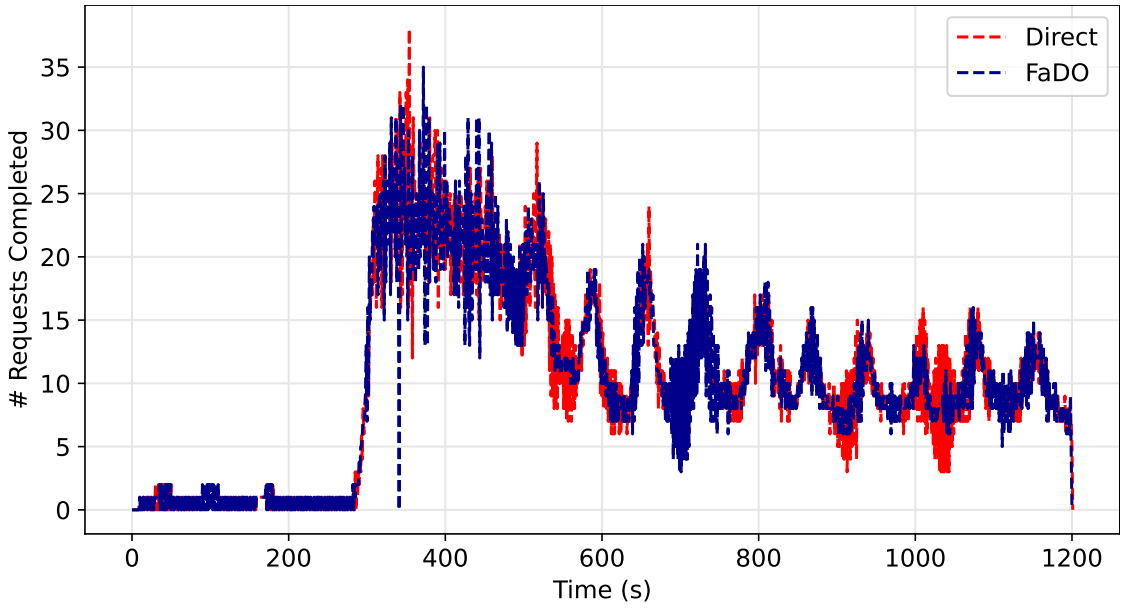


Figure C.2.: The number of requests completed over time when connecting to cloud cluster 1 directly and through FaDO. (Load Pattern 1)

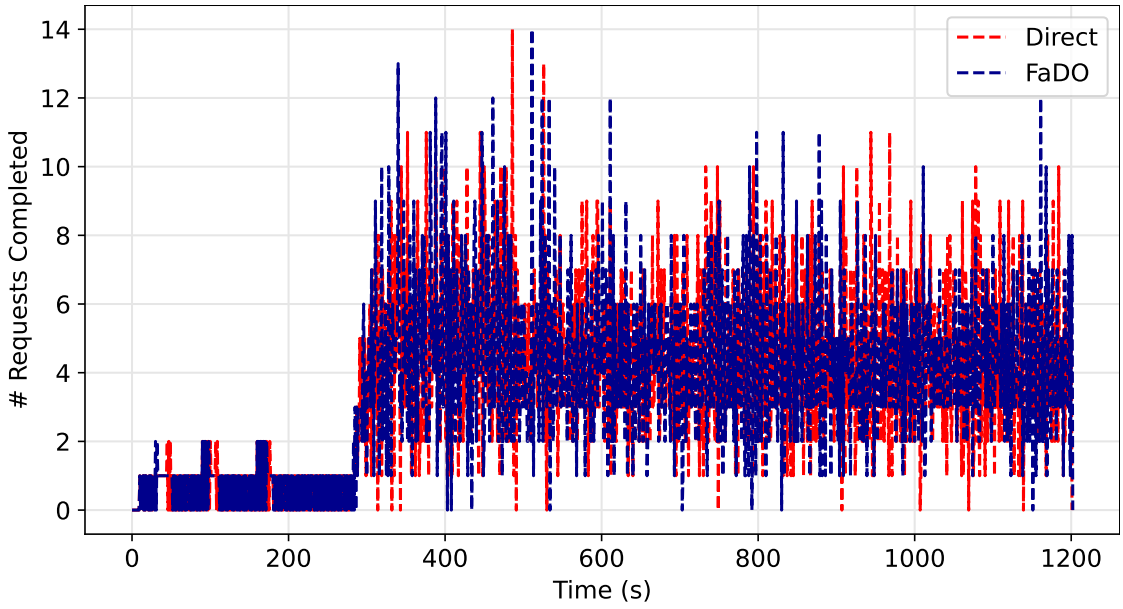


Figure C.3.: The number of requests completed over time when connecting to cloud cluster 2 directly and through FaDO. (Load Pattern 1)

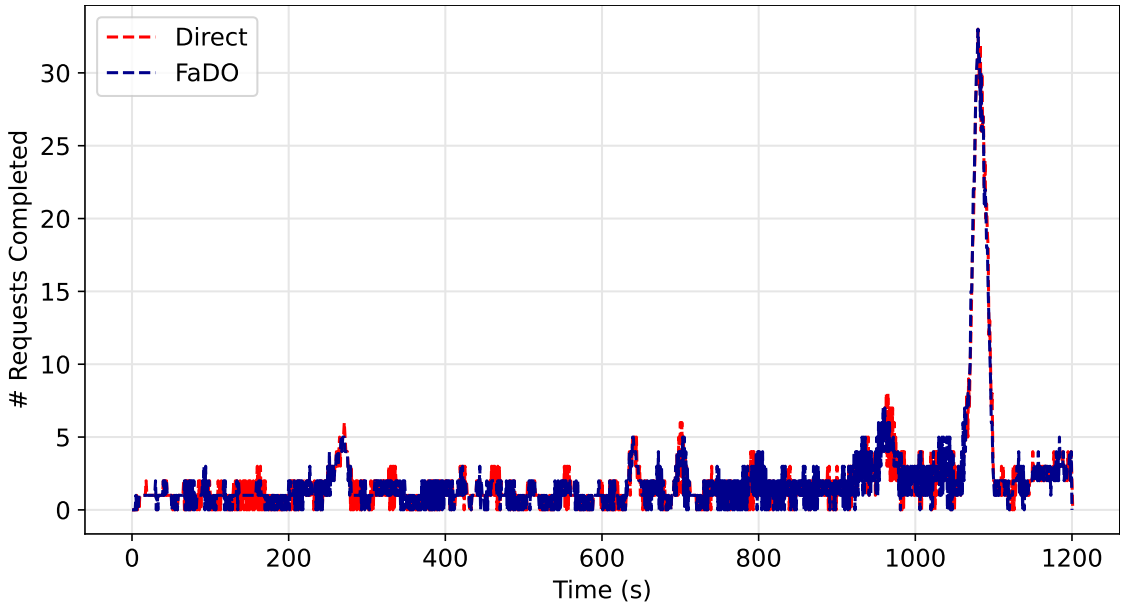


Figure C.4.: The number of requests completed over time when connecting to the AWS cluster directly and through FaDO. (Load Pattern 2)

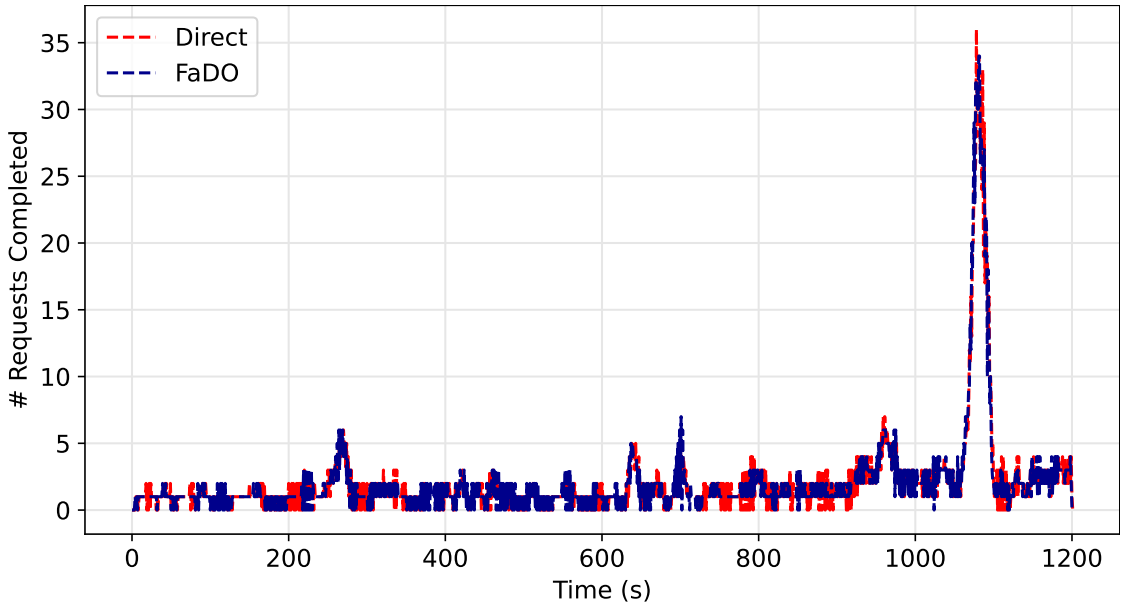


Figure C.5.: The number of requests completed over time when connecting to the HPC cluster directly and through FaDO. (Load Pattern 2)

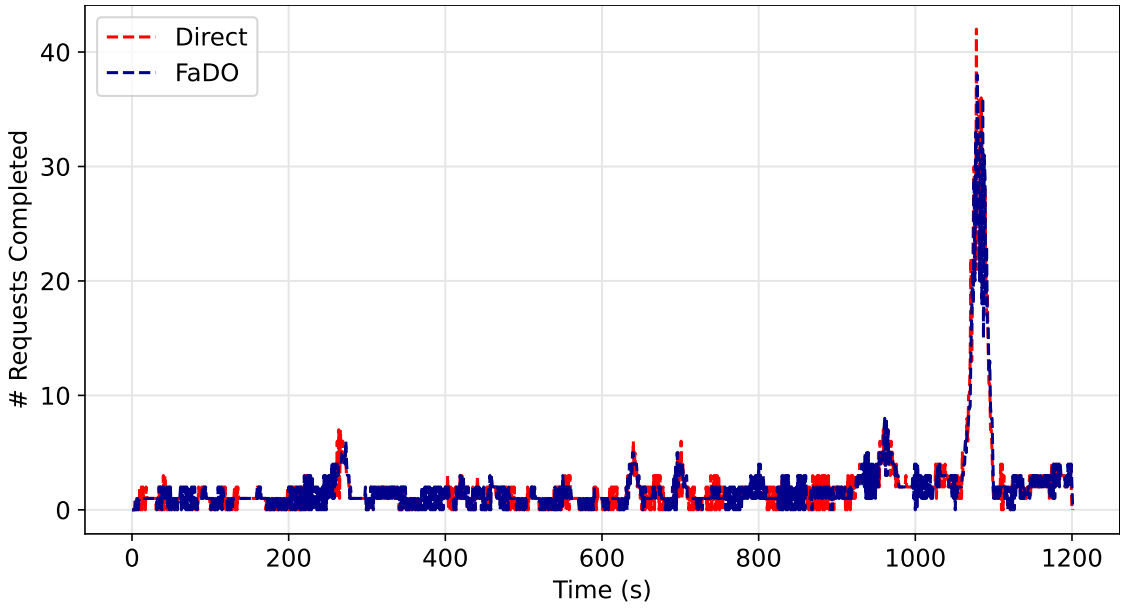


Figure C.6.: The number of requests completed over time when connecting to cloud cluster 1 directly and through FaDO. (Load Pattern 2)

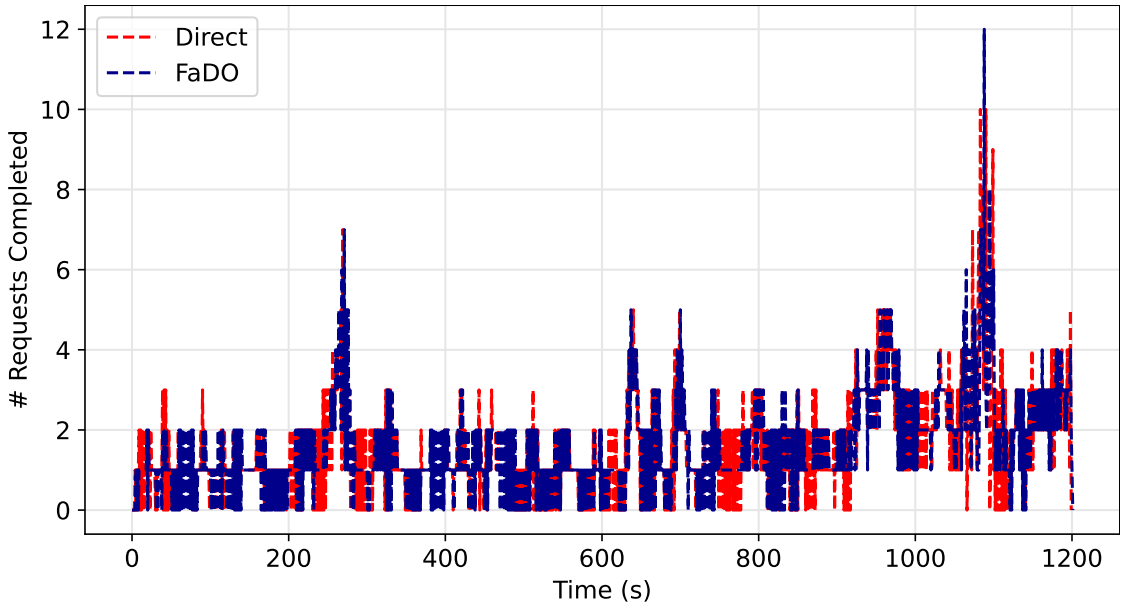


Figure C.7.: The number of requests completed over time when connecting to cloud cluster 2 directly and through FaDO. (Load Pattern 2)

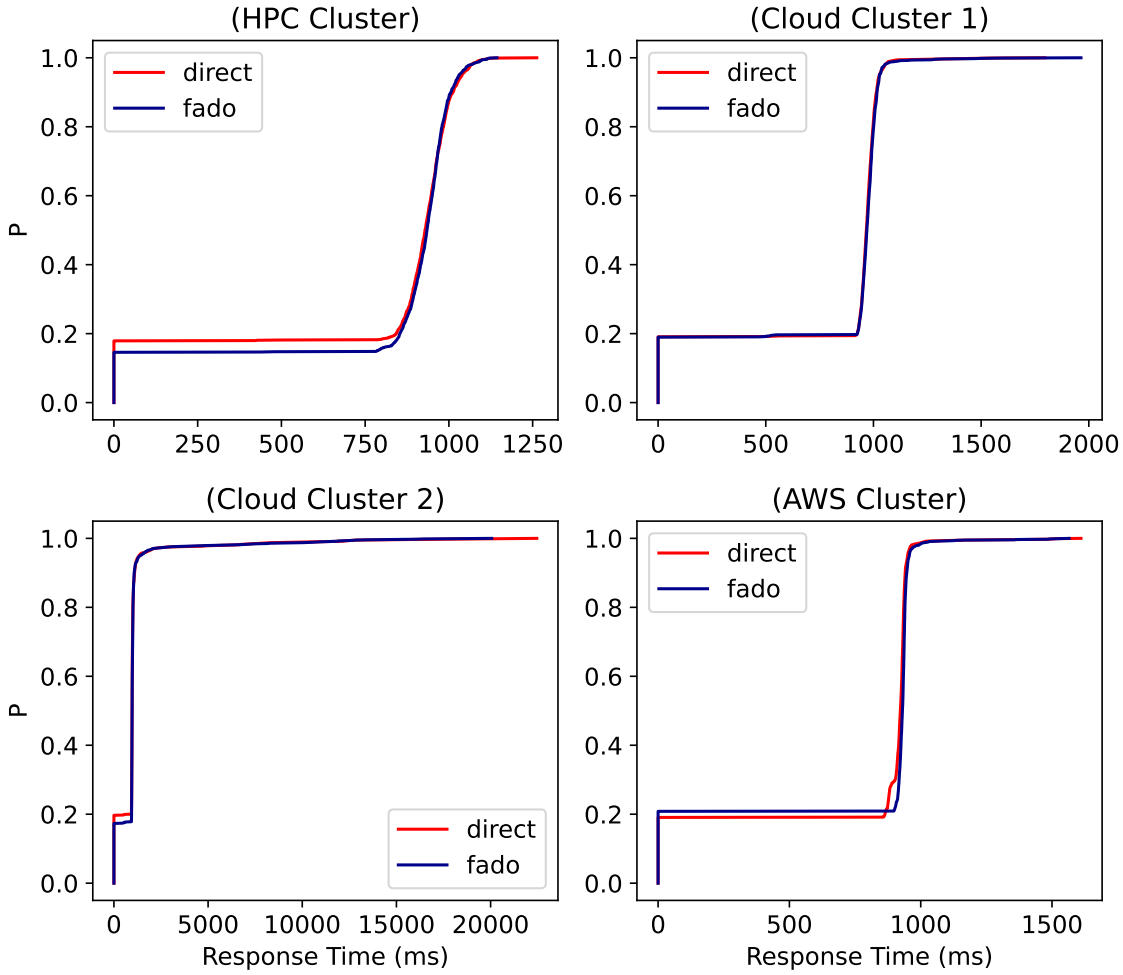


Figure C.8.: The cumulative distribution functions for the response times recorded when connecting to the different clusters either directly or through FaDO. (Load Pattern 2)

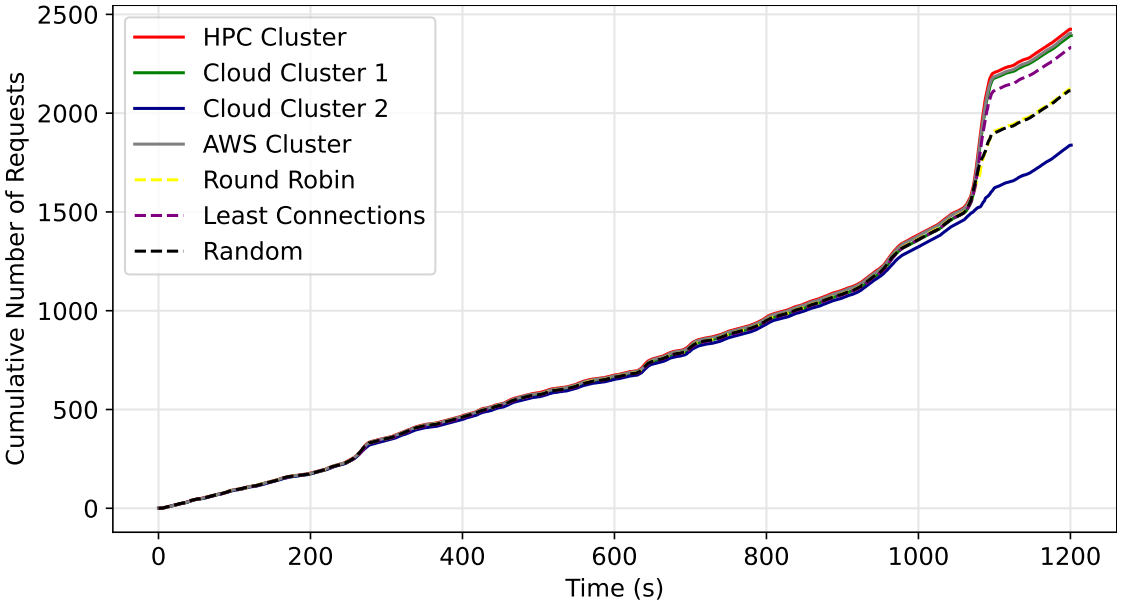


Figure C.9.: The cumulative sum of completed requests recorded when connecting through FaDO directly to the different clusters or load balancing the requests between the HPC cluster, cloud cluster 1, and cloud cluster 2. (Load Pattern 2)

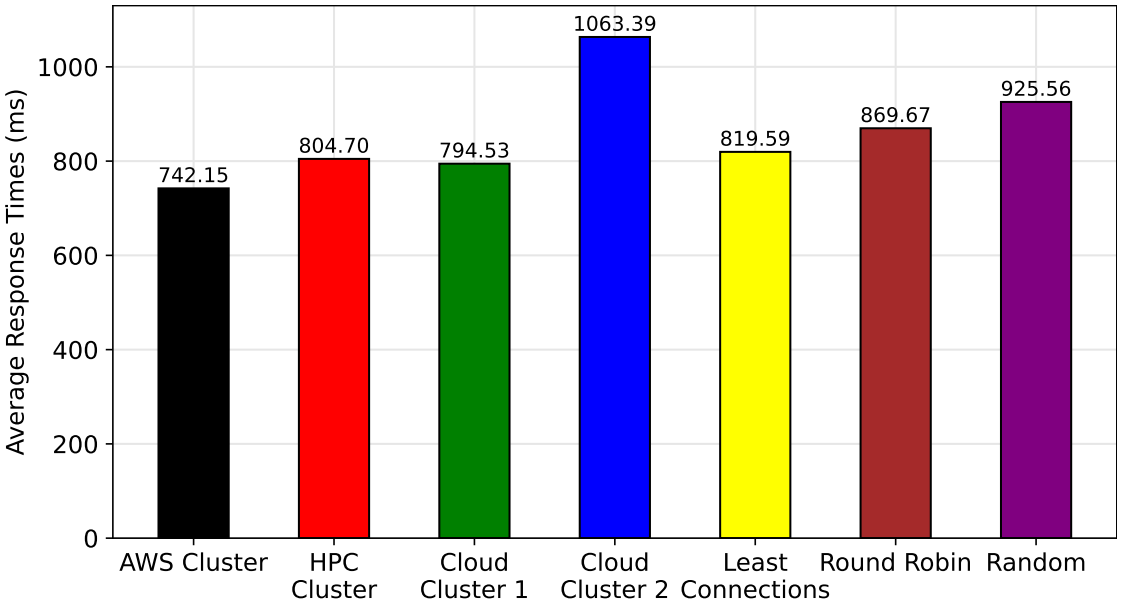


Figure C.10.: The average response times obtained when connecting through FaDO directly to the different clusters or load balancing the requests between the HPC cluster, cloud cluster 1, and cloud cluster 2. (Load Pattern 2)

List of Figures

List of Tables

List of Listings

Bibliography

[Amaa] Amazon Web Services, Inc. *Press release - Amazon Web Services Announces AWS Lambda*. URL: https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-announces-aws-lambda/ (visited on 09/20/2021).

[Amab] Amazon Web Services, Inc. *Serverless Computing*. URL: https://aws.amazon.com/lambda/ (visited on 09/20/2021).

[Amac] Amazone Web Services, Inc. *Serverless Computing – AWS Lambda Pricing – Amazon Web Services*. URL: https://aws.amazon.com/lambda/pricing/ (visited on 09/20/2021).

[Amad] Amazone Web Services, Inc. *What is Amazon S3?* URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#BasicsBucket (visited on 09/20/2021).

[Baa+21] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad. "On Merits and Viability of Multi-Cloud Serverless." In: *Proceedings of the ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 600–608. ISBN: 9781450386388.

[Bod] T. Bodz. *Announcing IBM Bluemix OpenWhisk Beta*. URL: https://www.ibm.com/blogs/cloud-archive/2016/11/announcing-ibm-bluemix-openwhisk-beta/ (visited on 10/01/2021).

[Bro+14] A. Brogi, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, E. Pimentel, and F. D'Andria. "SeaClouds." In: *ACM SIGSOFT Software Engineering Notes* 39.1 (Feb. 2014), pp. 1–4. DOI: 10.1145/2557833.2557844.

[Car] B. Carlson. *node-postgres - Welcome*. URL: https://node-postgres.com/ (visited on 04/17/2021).

[Chr] J. Christensen. *pgx - PostgreSQL Driver and Toolkit*. URL: https://github.com/jackc/pgx (visited on 04/17/2021).

[Doca] Docker Inc. *Caddy - Official Image | Docker Hub*. URL: https://hub.docker.com/_/caddy (visited on 04/17/2021).

[Docb] Docker Inc. *Empowering App Development for Developers*. URL: https://www.docker.com/ (visited on 04/17/2021).

[Docc] Docker Inc. *MinIO Server by MinIO, Inc. | Docker Hub*. URL: https://hub.docker.com/_/minio (visited on 04/17/2021).

[Docd] Docker Inc. *openfaas's Profile | Docker Hub*. URL: https://hub.docker.com/u/openfaas (visited on 04/17/2021).

[Doce] Docker Inc. *Overview of Docker Compose*. URL: https://docs.docker.com/compose/ (visited on 04/17/2021).

[Docf] Docker Inc. *Postgres - Official Image | Docker Hub*. URL: https://hub.docker.com/_/postgres (visited on 04/17/2021).

[F5] F5, Inc. *NGINX Documentation*. URL: https://docs.nginx.com/ (visited on 04/17/2021).

[Fac] Facebook, Inc. *React - A JavaScript library for building user interfaces*. URL: https://reactjs.org/ (visited on 04/17/2021).

[Goo] Google LLC. *Key terms*. URL: https://cloud.google.com/storage/docs/key-terms#buckets (visited on 09/20/2021).

[Gra] Grafana Labs. *Load testing for engineering teams*. URL: https://k6.io/ (visited on 09/20/2021).

[Hap] Haproxy Technologies. *HAProxy*. URL: http://www.haproxy.org/ (visited on 04/17/2021).

[IBM] IBM. *IBM Cloud Docs*. URL: https://cloud.ibm.com/docs/cloud-object-storage?topic=cloud-object-storage-getting-started-cloud-object-storage (visited on 09/20/2021).

[Jin+21] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen. "Function delivery network: Extending serverless computing for heterogeneous platforms." In: *Software: Practice and Experience* (2021). DOI: https://doi.org/10.1002/spe.2966. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2966.

[Kir] Y. Kiriaty. *Announcing general availability of Azure Functions*. URL: https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/ (visited on 10/01/2021).

[Kub] Kubernetes Authors. *Kubernetes*. URL: https://kubernetes.io/ (visited on 04/23/2021).

[Mat] Material-UI SAS. *The React component library you always wanted*. URL: https://mui.com/ (visited on 04/17/2021).

[Mina] MinIO, Inc. *Bucket Replication*. URL: https://docs.min.io/minio/baremetal/replication/replication-overview.html (visited on 05/10/2021).

[Minb] MinIO, Inc. *Bucket Replication*. URL: https://docs.min.io/minio/baremetal/replication/replication-overview.html (visited on 05/10/2021).

[Minc] MinIO, Inc. *Golang Admin Client API Reference*. URL: https://github.com/minio/madmin-go (visited on 05/10/2021).

[Mind] MinIO, Inc. *mc mirror*. URL: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc/mc-mirror.html#command-mc-mirror (visited on 05/10/2021).

[Mine] MinIO, Inc. *MinIO*. URL: https://min.io (visited on 05/10/2021).

[Minf] MinIO, Inc. *MinIO Admin (mc admin)*. URL: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc-admin.html (visited on 05/10/2021).

[Ming] MinIO, Inc. *MinIO Client (mc)*. URL: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc.html (visited on 05/10/2021).

[Minh] MinIO, Inc. *MinIO - Go Client API Reference*. URL: https://docs.min.io/docs/golang-client-api-reference.html (visited on 05/10/2021).

[Mini] MinIO, Inc. *MinIO - JavaScript Client API Reference*. URL: https://docs.min.io/docs/javascript-client-api-reference.html (visited on 05/10/2021).

[Minj] MinIO, Inc. *MinIO | Learn how to configure your MinIO server*. URL: https://docs.min.io/docs/minio-server-configuration-guide.html (visited on 04/17/2021).

[Mona] MongoDB, Inc. *GridFS*. URL: https://docs.mongodb.com/manual/core/gridfs/ (visited on 05/10/2021).

[Monb] MongoDB, Inc. *MongoDB vs PostgreSQL*. URL: https://www.mongodb.com/compare/mongodb-postgresql (visited on 05/10/2021).

[Monc] MongoDB, Inc. *Replication*. URL: https://docs.mongodb.com/manual/core/replication/ (visited on 05/10/2021).

[Mond] MongoDB, Inc. *Sharding*. URL: https://docs.mongodb.com/manual/core/sharding/ (visited on 05/10/2021).

[Mone] MongoDB, Inc. *What is MongoDB?* URL: https://docs.mongodb.com/manual/ (visited on 05/10/2021).

[Ope] OpenFaaS. *Introduction - OpenFaaS*. URL: https://docs.openfaas.com/ (visited on 04/26/2021).

[Pol] J. Polites. *Google Cloud Functions: A serverless environment to build and connect cloud services*. URL: https://cloud.google.com/blog/products/gcp/google-cloud-functions-a-serverless-environment-to-build-and-connect-cloud-services_13 (visited on 10/01/2021).

[Pos20] L. R. Possani. "Self-Adaptive Data Management for Heterogeneous FaaS Platform." msthesis. Technische Universität München, 2020.

[Rao] L. Rao. *PiCloud Launches Serverless Computing Platform To The Public*. URL: https://techcrunch.com/2010/07/19/picloud-launches-serverless-computing-platform-to-the-public/ (visited on 09/20/2021).

[Reg] A. Regalado. *Who Coined 'Cloud Computing'?* URL: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/ (visited on 09/20/2021).

[Sar] J. J. Sarjeant. *Axios*. URL: https://axios-http.com/ (visited on 04/17/2021).

[Sof] Software Freedom Conservancy. *Git*. URL: https://git-scm.com/ (visited on 04/17/2021).

[SP19] I. Stoica and D. Petersohn. *Two missing links in Serverless Computing: Stateful Computation and Placement Control*. 2019. URL: https://medium.com/riselab/two-missing-links-in-serverless-computing-stateful-computation-and-placement-control-964c3236d18 (visited on 04/17/2021).

[Staa] Stack Holdings. *Welcome to Caddy*. URL: https://caddyserver.com/docs/ (visited on 04/17/2021).

[Stab] Stack Holdings GmbH. *Command Line*. URL: https://caddyserver.com/docs/command-line (visited on 04/17/2021).

[Str] StrongLoop, IBM, and other expressjs.com contributors. *Express*. URL: https://expressjs.com/ (visited on 04/17/2021).

[Syn] Synergy Research Group. *Cloud Market Ends 2020 on a High while Microsoft Continues to Gain Ground on Amazon*. URL: https://www.srgresearch.com/articles/cloud-market-ends-2020-high-while-microsoft-continues-gain-ground-amazon (visited on 09/15/2021).

[Thea] The Apache Software Foundation. *Documentation*. URL: https://openwhisk.apache.org/documentation.html (visited on 04/25/2021).

[Theb] The PostgreSQL Global Development Group. *PostgreSQL - Documentation*. URL: https://www.postgresql.org/docs/ (visited on 04/23/2021).

[Thec] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 13: 11.6. Unique Indexes*. URL: https://www.postgresql.org/docs/13/indexes-unique.html (visited on 04/23/2021).

[Thed] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 13: 33.14. Environment Variables*. URL: https://www.postgresql.org/docs/13/libpq-envars.html (visited on 04/17/2021).

[Thee] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 13: 3.4. Transactions*. URL: https://www.postgresql.org/docs/13/tutorial-transactions.html (visited on 04/23/2021).

[Thef] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 13: 5.4. Constraints*. URL: https://www.postgresql.org/docs/13/ddl-constraints.html (visited on 04/23/2021).

[Theg] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 13: CREATE VIEW*. URL: https://www.postgresql.org/docs/13/sql-createview.html (visited on 04/23/2021).

[TM] A. Tridgell and P. Mackerras. *rsync(1) man page*. URL: https://download.samba.org/pub/rsync/rsync.1 (visited on 05/10/2021).

[Too] G. W. Toolkit. *gorilla/mux*. URL: https://github.com/gorilla/mux (visited on 04/17/2021).