Informatics

STORE: Self-Provisioning Storage for the Next Generation of Serverless Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ing. Florian Trimmel, BSc
Matrikelnummer 12123438

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.techn. Stefan Nastic, BSc
Mitwirkung: Dipl.-Ing. Dr. Thomas Pusztai, BSc
Dipl.-Ing.in Cynthia Marcelino, BSc

Wien, 30. September 2025

[illegible] [illegible]

Florian Trimmel Stefan Nastic

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

STORE: Self-Provisioning Storage for the Next Generation of Serverless Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Ing. Florian Trimmel, BSc
Registration Number 12123438

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.techn. Stefan Nastic, BSc
Assistance: Dipl.-Ing. Dr. Thomas Pusztai, BSc
Dipl.-Ing.in Cynthia Marcelino, BSc

Vienna, September 30, 2025

[illegible] [illegible]

Florian Trimmel Stefan Nastic

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU Wien Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Ing. Florian Trimmel, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. September 2025

[illegible]

Florian Trimmel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Zunächst möchte ich meinem Betreuer, Assistant Prof. Dr. Stefan Nastic, danken, dass er mir die Möglichkeit gegeben hat, diese Arbeit zu schreiben, und mir wertvolles Feedback gegeben hat, um die Vision hinter dieser Arbeit zu formen und zu verfeinern. Als Nächstes möchte ich Cynthia Marcelino und Thomas Pusztai dafür danken, dass sie immer für Fragen zur Verfügung standen und mir Input gegeben haben.

Außerdem möchte ich meiner Familie für ihre vielfältige Unterstützung während meines Studiums danken. Abschließend möchte ich Christopher Pany dafür danken, dass er mich beim Aufbau der Testumgebung unterstützt hat.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First, I would like to thank my advisor, Assistant Prof. Dr. Stefan Nastic, for giving me the opportunity to conduct this thesis and for providing valuable feedback to shape and refine the vision backing the thesis. Next, I would like to thank Cynthia Marcelino and Thomas Pusztai for their continuous availability to answer questions and provide input.

Furthermore, I would like to extend my gratitude to my family for their support of my studies in many ways. Finally, I want to thank Christopher Pany for supporting me in setting up the testbed hardware.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Serverless Computing bietet bedarfsgerechte Elastizität, nutzungsabhängige Abrechnung und vereinfachte Bereitstellung. Serverless Functions sind in der Regel zustandslos und für den Datenaustausch auf externe Speicherdienste wie Objektspeicher oder Datenbanken angewiesen. Die Bereitstellung und Konfiguration dieser Speichersysteme erfordert nach wie vor manuelle Konfiguration oder deklarative Skripte, was die Komplexität erhöht und das Risiko von Konfigurationsfehlern erhöht. Während die Grundprinzipien des Serverless Computing für Serverless Functions bereits gut funktionieren, passen Speicherlösungen noch nicht wirklich in das Serverless Paradigma.

Die Probleme mit bestehenden Lösungen sind vielfältig. Einige Lösungen können die manuelle Konfiguration, Leistungsoptimierungen und Sicherheits-/Zugriffskontrollen nicht komplett eliminieren. Andere funktionieren nur mit bestimmten Cloud-Anbietern. Im Allgemeinen gibt es eine Vielzahl von Angeboten für die Datenspeicherung. Diese Heterogenität der Speicherlandschaft stellt ein Problem der Auswahlüberlastung dar. Insgesamt sehen wir eine paradigmatische Inkompatibilität zwischen aktuellen Speicherlösungen und dem Serverless-Paradigma. Um die Versprechen von Serverless Computing zu erfüllen, ist daher eine paradigmatische Vereinheitlichung von Speicher und Funktionen erforderlich.

Als Lösungsansatz stellt diese Arbeit STORE vor, einen Self-Provisioning Speicher für Serverless Functions. Wir beschreiben die Lebenszyklusphasen von Self-Provisioning sowie eine herstellerunabhängige und erweiterbare Architektur, die diese implementiert. STORE wählt automatisch das optimale Speicher-Backend aus und eliminiert den Aufwand für Entwickler durch Zero-Touch- und Zero-Configuration-Provisioning. Das wird durch die Verlagerung der Selbstbereitstellungslogik auf die Plattformebene erreicht. Es umfasst ein Modell für die Berechtigungsverwaltung und die Parallelitätskontrolle sowie verschiedene Minimal-Kontakt-Schnittstellen für die Interaktion mit dem Speicher. Zusätzlich wird ein Konzept für die Abwärtskompatibilität bestehender Funktionen entwickelt.

Unsere Evaluierung auf der Grundlage realer Anwendungsfälle für serverlose Funktionen und Workflows zeit, dass STORE den Implementierungsaufwand im Vergleich zu etablierten Infrastructure-as-Code-Frameworks wie Terraform und Pulumi um bis zu 84% reduziert. Die Leistungs- und Skalierbarkeitsexperimente zeigen, dass STORE eine geringe Latenz und lineare Skalierbarkeit mit relativen Overheads ab 0.1% erreicht. Die Abwärtskompatibilität wird durch eine erfolgreiche Zero-Touch-Migration demonstriert.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

Abstract

Serverless computing provides on-demand elasticity, pay-per-use billing, and simplified deployment. However, serverless functions are typically stateless and depend on external storage services such as object stores or databases to exchange data. Provisioning and configuring these storage systems still requires manual setup or declarative scripts, introducing complexity, slowing development, and increasing the risk of configuration errors. While embracing the pillars of serverless computing already works well for serverless functions, storage solutions, among other supporting systems, do not yet fit into the serverless paradigm very well.

Problems with existing solutions come in different flavors. Some solutions cannot bridge the last-mile effort gap, requiring users to do manual configuration, performance tuning, and security/access control. Others are vendor-locked to only work on specific platforms or with specific cloud providers. In general, there is a myriad of offerings for data storage, each having different interfaces, advantages, and disadvantages. This heterogeneity of the storage landscape presents as a problem of choice overload. Altogether, we see a paradigmatic incompatibility between current storage solutions and the serverless paradigm. To fulfill the promises of serverless computing, there is a need for paradigmatic unification of storage and compute.

To address the aforementioned challenges, this thesis introduces STORE, a self-provisioning storage for serverless functions. We describe the lifecycle phases of self-provisioning as well as a vendor-agnostic and extensible architecture that implements them. STORE automatically selects the optimal storage backend and eliminates developer effort through zero-touch and zero-configuration provisioning, achieved by moving the self-provisioning logic to the platform level. It includes a model for permission management and concurrency control, as well as different minimal contact interfaces for interaction with storage. Additionally, a concept for backward compatibility for existing function code is devised.

Our evaluation results based on real-world serverless function and workflow use cases show that STORE reduces implementation effort by up to 84% compared to well-established Infrastructure-as-Code frameworks such as Terraform and Pulumi. The performance and scalability evaluations demonstrate that STORE achieves low latency and linear scalability, with relative overheads as low as 0.1%. Backwards compatibility is evaluated by showing a successful zero-touch migration from an existing function.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

1.1 Motivation

In recent times, the paradigm of cloud-native software and serverless computing has a strong foothold in academics and industry [JSS+19, TJI+25]. Alleviating developers of the burden of manually managing infrastructure, as well as providing small deployment units and pay-as-you-go pricing, are some of the main benefits that cloud infrastructure providers such as Amazon AWS [amaa] or Google Cloud Platform [gooa] introduce.

While the promises of serverless computing are held for Function as a Service (FaaS), developers still have to do **manual steps for many supporting systems** such as object storage, databases, or Application Programming Interface (API) Gateways. From writing Terraform descriptions to using AWS CDK, many existing Infrastructure as Code solutions require considerable effort to specify how supporting infrastructure should be provisioned [FPO+24]. Additionally, manual management of security concerns, such as access control and role management, requires significant development time. Furthermore, conservative policies that ensure security by default may lead to tedious debugging sessions.

Another current challenge is **handling vendor lock-in** [MCCL23, JAB25]. Vendor lock-in is the need to use provider-specific libraries and products (e.g., Amazon Lambda, Amazon S3, Amazon DynamoDB) for the most convenience and performance when developing a serverless application. The result of sticking with provider-specific solutions is the overly tight coupling to a specific provider and the inability to switch providers in case of unexpected cost explosions or changed legal circumstances.

The recently introduced concept of **Self-Provisioning Infrastructure** suggests a solution to the challenges described above [Nas24]. To strengthen the core concepts of serverless computing, a provider-agnostic, zero-configuration, zero-touch solution to developing, deploying, and migrating serverless applications, including their supporting

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

BaaS services, would be optimal. This topic is especially pressing in serverless storage, where the amount of offerings is large, and the presence of storage is ubiquitous due to the complete disaggregation of storage in FaaS [MA24].

1.2 Problem

While there are no implementations of Self-Provisioning Infrastructure (SPI) at the point of writing, there have been attempts to reduce the complexity of infrastructure management in serverless environments. Those, however, do not completely eliminate infrastructure management or suffer from other drawbacks like vendor lock-in or heterogeneity. Below, we describe the current challenges in detail.

Last Mile Effort

Current solutions, even ones offering to completely abstract storage provisioning, still require manual configuration interaction [SKkP$^+$25]. This can range from connection configuration and schema definitions to manual performance tuning, security, and access control. Since these types of configurations can take a substantial amount of time and, if done incorrectly, can cause serious damage to a system's performance, security, and integrity, they often present as points where developers stumble and promising projects fail. Reducing or eliminating those last-mile configuration efforts is therefore important for serverless environments to flourish since developers can focus on business logic.

Vendor Lock-In

Languages like Darklang [dar] that offer completely automatic provisioning of data storage often act as a cloud provider themselves. Applications configured in this environment suffer from vendor-lock in, meaning a migration to different commercial or non-commercial cloud providers is hard or even impossible. The same happens when a proprietary storage solution (or interface to a storage solution) is chosen for data storage. Migration of serverless applications [JAB25] that use vendor-locked storage requires technical expertise and considerable development effort. Reducing or eliminating all dimensions of vendor lock-in, retroactively as well as proactively is necessary for resilient and portable serverless applications.

Product Heterogeneity

There is a broad variety of offerings for data storage [KGT$^+$23, SKkP$^+$25, MA24, Min25, The25], each offering its own interface, technologies, advantages, and disadvantages. Navigating the ever-growing landscape of storage options becomes increasingly complex for application developers. Reducing this complexity and bringing back focus to the core objective of storing and retrieving data via a unified interface lightens the cognitive load on developers and, at the same time, gives specialists a basis for generally effective performance and cost optimizations when they become necessary.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Paradigmatic Incompatibility

The serverless computing paradigm promises that no infrastructure management is necessary for developers. Existing Infrastructure as Code (IaC) solutions [ter, pul], while increasing the level of abstraction and portability, still require specification of a desired infrastructure state and, by extension, still require manual infrastructure management. Other promises of serverless computing, such as automatic elasticity and pay-per-use pricing, are already accurately depicted by some solutions [SKkP^{+}25]. Those, however, might still suffer from not being globally available due to being vendor-locked. The promise of rapid development and thereby reducing time-to-market is also not fulfilled if storage Application Programming Interfaces (APIs) have to be mastered prior to storing data. This paradigmatic incompatibility between data storage and serverless computing still exists, and the gap needs to be closed to fully harness the benefits that serverless computing provides a unified view of the paradigm for developers.

1.3 Illustrative Scenario

To better understand the challenges associated with serverless BaaS provisioning, we present a shopping cart management use case in Fig. 1.1. Our use case involves verifying input data, managing basket content, collecting statistics, and enriching baskets. It quickly becomes clear that external storage is needed for basket contents, current item stock, and product catalog, as well as for storing analytic data and generated documents. Initially, the developer may opt to model all stored data in a relational database. This includes choosing an offering, defining a schema, setting up user management, utilizing a client in the functions, and managing connections to the database. After all this is set up, the developer notices that storing documents, such as generated order previews, in a relational database might not be the optimal design decision, so they choose to switch to using an object store. Now the cycle of provisioning, data migration, and configuration management begins anew. Similar cases can be made for using a document database for basket data, warehouses for analytics data, and a key-value store (KVS) for stock data. For optimal utilization, the developer must master five different storage technologies and also consider the dimensions of availability, scalability, and cost, turning the need for storing data into a complex engineering task. STORE abstracts this complexity by mapping basket state, stock, documents, and analytics to the most suitable backends and switching transparently as requirements evolve, allowing developers to focus on business logic rather than storage provisioning and migration.

We envision the ideal case for storage handling as a complete abstraction of storage behind an API/function call. One call should be **the entirety** of the effort needed to **persistently** store data or retrieve it (potentially later in time or in a different function). Listing 1 shows an example of how shopping basket content should be retrievable, modified, and then stored persistently again (Lines 1 to 4). Further distilling this vision, persistence can be fully abstracted behind data structures (e.g, maps or arrays), making storage and retrieval as simple as treating the data as if it were already present in the structure

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

(Lines 6 and 7).

Listing 1 Basket Management Functions

```
1 function addItem(item):
2     basket ← retrieve("basket")
3     basket.items ← basket.items + item
4     store("basket", basket)
5
6 function changeQuantity(item, quantity):
7     data["basket"][item].quantity ← quantity
```

1.4 Research Questions

RQ1 - To what extent can we extend the serverless paradigm to include storage?

One of the primary goals of serverless computing is to alleviate developers from the burden of interacting with infrastructure and platforms [SKM22]. This already works well for Function-as-a-Service (FaaS), where agnostic computing models reduce the complexity for the developer to write handler code. For Backend-as-a-Service (BaaS) components, this is not yet fully realized due to the vast heterogeneity of offerings and vendor-lock-in [MCCL23, SBR+22]. This problem is especially prevalent in serverless storage, where the selection of storage can yield significant performance benefits, and migration between frameworks can be challenging [SKkP+25, KGT+23, ZWM+23]. While there exist tools that aim to abstract the provisioning of storage, they do not align well with the serverless paradigm, and the complexity of selecting and learning how to use those tools still remains [ter, pul]. This research question is about closing the gap between storage and compute in serverless environments and deriving a vision for paradigmatic unification.

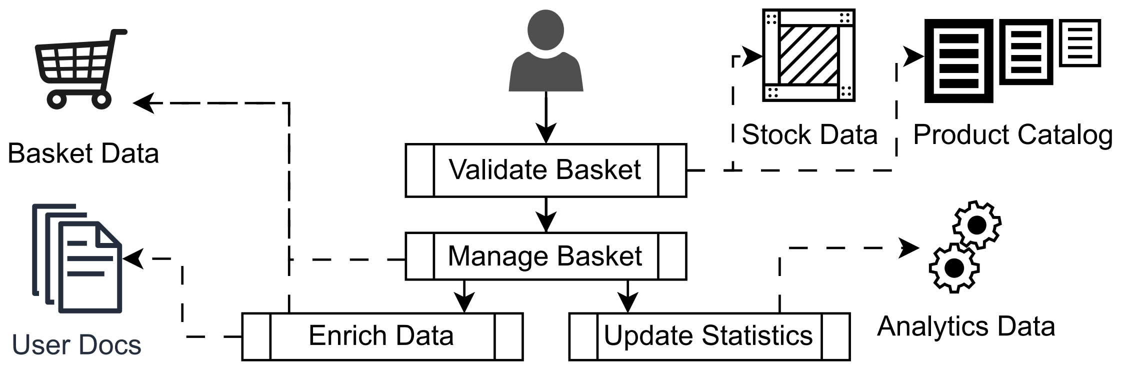


Figure 1.1: Serverless Workflow for Shopping Cart Management

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

RQ2 - How to allow a serverless platform to seamlessly select the optimal storage type based on data shape, and access pattern?

Learning the ins and outs of different storage types can be quite intricate [SGR15]. Approaches beyond SQL [MK19] often ship their own query languages or operate on different data storage and retrieval models, requiring further investment to efficiently utilize them. At an early stage in development, when detailed requirements such as Service Level Objectives (SLOs) and the concrete data shape are not yet finalized, selecting the wrong storage type can have long-term drawbacks, as switching at a later stage may incur additional cost and effort. Finding the right components to efficiently determine an appropriate or even optimal type of storage is hard [MSKV19, BGCA24, PVS19], and introducing the additional dimension of seamlessly switching further increases complexity. At the core of this research question lies the idea of concealing product heterogeneity and leveraging the broad spectrum of available storage solutions.

RQ3 - How to provide zero-touch and zero-configuration self-provisioning storage for serverless functions?

The cloud computing landscape encompasses a myriad of projects, most of which are designed to be highly configurable [cnca]. While this makes the landscape as a whole more versatile, it also increases the initial investment required to get started and bring projects online. Especially with storage, a significant amount of time can be spent on setup, ensuring elastic scalability, and performance tuning. Additionally, self-provisioning is a problem that cannot be solved purely statically [RL19], introducing another dimension of complexity. Solutions promising to alleviate developer efforts tend to shift the effort, pulling it up to a higher abstraction level, while still leaving last-mile efforts to the developer, and potentially creating adverse effects such as reduced portability and interoperability. This research question investigates the complete removal of last-mile efforts and vendor lock-in behind a developer-transparent mechanism.

1.5 Contributions

We summarize our contributions as follows:

- **STORE:** A novel lifecycle model and architecture for self-provisioning serverless storage that defines lifecycle phases (preparation, resolution, policy enforcement, dynamic binding, translation) and achieves a reduction in developer effort of up to 84% while having an overhead between 0.1% and 38% and providing linear scalability. This enables automated provisioning and management of storage resources, eliminates manual configuration, and facilitates the serverless paradigm for storage.
- **Dynamic Storage Selection:** A novel runtime mechanism for adaptive storage binding that automatically selects the optimal storage backend based on data shape,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

access pattern, and SLOs, ensuring efficient resource usage by matching workloads with the most suitable storage type without developer intervention.

- **Zero-Touch Configuration:** A developer-transparent provisioning mechanism that removes the need for manual setup. STORE achieves this by embedding self-provisioning logic at the serverless platform level, enabling functions to seamlessly access storage while enforcing fine-grained permissions, thereby ensuring zero-touch and zero-configuration operation.

1.6 Outline

This thesis has eight chapters. Chapter 1 presents the illustrative scenario and research questions. It also outlines the contributions of this thesis. Chapter 2 introduces the relevant background. Chapter 3 summarizes related work and how this thesis is distinct from related work. Chapter 4 describes the function abstraction model, the Self-Provisioning lifecycle, and the architecture overview. Chapter 5 describes the dynamic storage selection and the zero-touch configuration introduced by STORE and their usage. Chapter 6 presents the prototype implementation details. Chapter 7 discusses the experiments and evaluation. Chapter 8 concludes with a final discussion and future work. Additional resources can be found in the repositories linked in Appendix A.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

2.1 Cloud Computing

Cloud Computing is a computing paradigm centered around moving computing infrastructure from close proximity (i.e., a person's desk or a company's server room) to a central location elsewhere [AMI20]. The promised benefits included the alleviation of infrastructure management efforts by having an external entity manage server hardware. This way, additional benefits such as easy scalability and cost reduction through large-scale clusters of computing power result in a lower cost per compute unit. Besides the promised economies of scale effects, one of the core statements has been that users should not have to worry about managing server hardware manually.

Different commercial [gooa] and non-commercial [ope] offerings exist that allow following the cloud computing paradigm. Multiple levels of infrastructure abstraction can be distinguished in cloud computing:

- **Infrastructure as a Service (IaaS):** Operates at a comparatively low level of abstraction, where the user's view is on virtualized hardware resources such as compute, storage, and networking.

- **Platform as a Service (PaaS):** Sits at a higher level of abstraction compared to IaaS by concealing low-level infrastructure details. Users view services as a complete platform for developing, deploying, and managing applications rather than managing servers or networking.

- **Software as a Service (SaaS):** Provides an even higher level of abstraction compared to PaaS, where entire applications are delivered as a service and fully managed by the provider.

2.2 Serverless Computing

Further increasing the abstraction level beyond SaaS, we arrive at the FaaS concept, breaking down applications into single functions that are managed by the provider. This concept isolates functions into stateless units of computation that can be easily scaled and offer very fine-grained pricing.

The benefits of FaaS are in part achieved through the disaggregation of storage and computing. Storage is among the Backend as a Service (BaaS) that make up the second part of serverless computing.

There are again commercial [amac] and non-commercial [Pro24] serverless providers that each provide ecosystems of BaaS and have slightly different execution models, which leads to some of the key challenges in serverless computing [JSS+19]:

- **Vendor Lock-In.** Providers tend to offer proprietary BaaS and APIs (e.g., storage, messaging, authentication) that make migration or portability difficult. The function logic often becomes tightly integrated with the provider's storage or data services, making users vulnerable in the event of the provider going out of business or experiencing price hikes.
- **Data Storage.** Because functions are stateless (due to the aforementioned disaggregation), all persistent state must be stored externally. This introduces challenges in consistency, latency, data movement, and the architectural design of how/where state is stored (object stores, databases, caches). Since many common use-cases require persistent storage, the problem is even more pressing.

2.3 Kubernetes, Operators & Knative

Kubernetes is an open-source container orchestration system originally developed at Google [kuba]. It is currently widely used across modern cloud environments and is also provided as a service by many platforms.

At its core, Kubernetes enables resilient operation of containerized applications by automating deployment, scaling, load balancing, and failover. It provides features such as service discovery, persistent storage orchestration, automated rollouts and rollbacks, bin packing for resource efficiency, self-healing, secret management, batch job execution, and horizontal scaling. Unlike traditional PaaS systems, Kubernetes does not enforce specific logging, monitoring, or middleware solutions.

2.3.1 Kubernetes Operators

Rather than orchestrating workflows step by step, Kubernetes continuously reconciles the actual system state with the user-defined desired state, resulting in a robust, extensible, and developer-friendly control plane for containerized applications.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

The **Operator pattern** [cncb] extends the capabilities of Kubernetes by automating the management of complex, stateful applications through controllers. While Kubernetes natively supports resources such as Pods, Services, and Deployments, operators are required to handle more complex administrative tasks, including database backups and replication.

At the core of the pattern lies the reconciliation loop: a controller continuously monitors the current system state and compares it against the desired state, which is specified through Custom Resource Definitions (CRDs). Operators contain domain knowledge and automate the work of infrastructure engineers. This includes dynamic configuration of applications, automated recovery from failures, and safe execution of upgrades for distributed data stores or other stateful services.

Comparison with Alternatives

While simpler mechanisms, such as Helm charts [hel], tools like Kustomize [Rep], or custom scripts, can be used to manage resources on Kubernetes, they lack integrated reconciliation, state awareness, and embedded domain logic. That makes Kubernetes Operators more robust for ongoing lifecycle management. Operators extend the Kubernetes API and represent the native way of depicting and managing complex resources declaratively within the cluster.

2.3.2 Knative

Knative [Pro24] realizes a serverless platform on top of Kubernetes by providing higher-level abstractions for building, deploying, and managing cloud-native applications. Knative is composed of independent components, most notably Knative Serving and Knative Eventing, which together enable both request-driven and event-driven workloads.

Knative Serving

Knative Serving provides a serverless runtime for stateless Hypertext Transfer Protocol (HTTP) applications. It introduces custom resources centered around Services (`ksvc`). Key features include scale-to-zero, request-based autoscaling, and advanced traffic routing. By abstracting away low-level Kubernetes constructs, Knative Serving simplifies deployment and management while retaining compatibility with standard Kubernetes features.

Knative Eventing

Knative Eventing enables event-driven architectures on Kubernetes, utilizing the CloudEvents specification [git] and providing components such as Event Sources, Brokers, Triggers, and Sinks. These abstractions enable loose coupling, with events being routed declaratively based on metadata. Knative Eventing can be used for workflow orchestration with native support via resources like Sequence and Parallel.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.4 MinIO and Cassandra

MinIO [Min25] and Cassandra [The25] are two distributed storage system technologies. Cassandra is a distributed NoSQL database optimized for high availability and scalability. MinIO is an object store that is marketed as lightweight, cloud-native, and highly compatible with existing cloud APIs.

MinIO

MinIO is fully compatible with the Amazon S3 API. Its core abstraction is the *object*, stored within *buckets*, which can be accessed using any S3-compatible client or library. This compatibility allows applications originally built for AWS S3 to run on MinIO without code modifications, making it attractive for hybrid and on-premises deployments.

From an architectural perspective, MinIO is typically deployed in clusters consisting of multiple nodes. The nodes form a storage pool. Objects are distributed across nodes using erasure coding, which provides resilience against node and disk failures while maintaining efficient storage utilization.

Apache Cassandra

Cassandra [The25] is a wide column store and distributed NoSQL database. Cassandra's architecture is fully distributed and based on a peer-to-peer model, where all nodes in a cluster have equal responsibilities. Nodes communicate using a gossip protocol to share information about cluster membership and state. Data is distributed across nodes using consistent hashing and partition keys, ensuring balanced load and fault tolerance. Replication provides resilience, with a replication factor (RF) determining how many copies of data are maintained across nodes and datacenters. Any node in the cluster can serve as a coordinator for client requests, forwarding reads and writes to the appropriate replicas.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Related Work

In this chapter, we refine the connection between our work and SPI, and then review state-of-the-art approaches that simplify serverless storage management, focusing on declarative infrastructure, portability models, and serverless storage systems.

3.1 Self-Provisioning Infrastructure

SPI, as a high-level concept, has already been introduced. This includes a partial schematic of high-level components of Self-Provisioning Infrastructure as well as more concrete ideas of how storage can be combined with the paradigm in the form of Self-Provisioning Storage Attachments [Nas24]. To differentiate the thesis from existing work, the degree of detail and the focus on a particular subfield of the paradigm have been chosen. To this date, no concrete implementation of Self-Provisioning Storage Attachments exists. Therefore, there are also no answers to the questions of how the design principles of SPI can be achieved in practice. SPI describes many design principles, and in this work, we lay particular focus on two of them:

- **Self-Provisioning** is described as the ability to provision a complete serverless function environment based on the business logic of the function alone. This includes provisioning of configuration models, BaaS, security, and permissions, among many others, that need to be automatically deduced from function code and, in extension, user behaviour and intentions. Reducing or eliminating infrastructure management effort becomes even more important in the context of overarching or even more complex computing paradigms like sky-computing [Sto24], federated FaaS [CBL+20], or in the Edge-Cloud continuum [NRF+22].
- **Multi-Level Portability and Interoperability** defines the goal for SPI to provide portability and interoperability at multiple levels in the architecture.

> Function handler code should be portable to multiple providers of FaaS. SPI should be independent of the used serverless platform, so it requires interoperability at the platform level. It should also be portable at the workflow level. We have refined the notion of portability and interoperability to encompass the notion of "it does not matter where we run" and call this design principle **Multi-Level Agnosticity**.

Beyond the aforementioned two core design principles, we also address the design principles of Self-Optimization (focusing on the dynamic, changing, and reevaluation of provisioning decisions) and SLO/Cost-awareness (the infrastructure should be aware of non-functional goals and strive for monetary cost minimization).

3.2 Declarative Infrastructure

First- [Ama25, ter] and second-generation [amab, pul] infrastructure-as-code tools are already well-established and widely used. Those, however, have the shortcomings of still requiring manual specification of the desired system state, as well as a limited ability to react to dynamic system changes and complex requirements. The concept of intent-based infrastructure [All25, BHB+19, AWA24] aims to address these shortcomings by enabling the specification of higher-level user intents in the system, which are evaluated and acted upon based on telemetry to achieve a state that matches the user's intent. While intent-based infrastructure reduces developer effort, it still requires explicit specification of intents, which are then extended and translated into lower-level configuration, often via the use of AI [BHB+19, AWA24]. EDMM [WBF+20] can be used to generate a target deployment model (a solution tailored to a concrete cloud provider) from an infrastructure model. EDMM puts the infrastructure model at the center of the deployment process, and in this sense, falls into the category of first-generation IaC tools. Approaches rooted in model engineering [VRVV24, OLB16, SIA17, SIA19, BBGK18] allow very abstract declaration of infrastructure. While approaches like CloudCAMP [BBGK18] or ARGON [SIA19] promise end-to-end provisioning, they rely on defining a model of either a system architecture (to be converted into infrastructure definitions) or the required infrastructure itself. Additionally, most model-based approaches operate on a higher level of abstraction and produce IaC as output, still requiring interaction with IaC tools. Approaches such as Darklang [dar] or Wing [win] promise to completely eliminate infrastructure declaration. Darklang acts as a cloud provider. Thus, while no provisioning is necessary, a type of vendor lock-in to Darklang cloud infrastructure still exists. Wing is a cloud-native programming language that intermingles infrastructure and business logic code. While tightly integrated, declaration of infrastructure is still necessary, and the translation step from Wing code to lower-level IaC presents an interface point to IaC tools.

Although declarative and intent-based approaches reduce effort, they still rely on explicit infrastructure specifications or new programming models. STORE eliminates infrastructure declarations entirely, abstracting storage at the level of workflows and data shapes, and letting the platform handle provisioning and adaptation transparently.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.3 Portability Models

Function-centric portability models [QZ20] allow reducing function code to a minimum, providing portability and composability across different cloud providers. Those frameworks tend to restrict interaction with BaaS to provide simplified views on functions and, therefore, do not usually interact with storage. Cloud Service Abstraction, promising portability by introducing an abstraction layer above BaaS, has been explored for serverless use cases [MCCL23]. While abstraction libraries such as Libcloud [Fou] or jclouds [apa] address vendor lock-in, the actual provisioning of infrastructure still has to be done manually. Model-based approaches provision federated serverless environments by abstracting primitives across providers but require extensive models and often depend on IaC for execution. For example, UMLPMSC [SAA+19] defines stereotypes for storage and databases that map to provider resources. BaaSLess [LGNR24] enables the dynamic provisioning of storage buckets from two different storage providers, focusing on realizing federated systems where the supporting BaaS are potentially constrained by provider-specific requirements, whereas STORE focuses on more general aspects of provisioning. Similarly, StoreLess [RHG+24] focuses on selecting the optimal type of storage in a federated serverless environment by utilizing colocation and a list-based heuristic. While the goal of selecting the best storage is similar, StoreLess focuses on selecting the best storage region, while our solution focuses on selecting the best type of storage. SEAPORT [YBKL20] is a model for assessing the portability of serverless applications. It includes an agnostic description of serverless applications as a high-level pipes-and-filters architecture in the form of a CASE model. The authors suggest using static code analysis to detect patterns (e.g., used storage) in function code. After transforming an existing application into a CASE model, it is suggested to use EDMM [WBF+20].

Although portability models aim to avoid vendor lock-in, they typically require extensive upfront modeling and static analysis, requiring predefined abstractions that must be maintained over time. In contrast, STORE performs provisioning and storage selection dynamically at runtime, adapting to workload characteristics without additional modeling effort from the developer.

3.4 Serverless Storage

Cloud provider offerings, such as Firestore [KGT+23] or DynamoDB [EGG+22], carry the issue of being vendor-locked to their respective platforms. FaunaDB, based on Calvin [TDW+12], is a deterministic database offering serverless capabilities through an API-based interface. CockroachDB Serverless [SKkP+25], based on the open-source database CockroachDB, provides relational database capabilities via an SQL interface. All solutions abstract many database management tasks, particularly those related to elasticity and scalability. However, the last-mile effort, including connection configuration, setting up permissions, defining schemas, learning the APIs, or formulating SQL queries, still remains.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

While serverless databases and STORE share some similarities in terms of pay-as-you-go pricing and compute scaling, STORE is not an implementation of a serverless database itself. STORE acts as an enabler to further simplify serverless storage and combat vendor lock-in while additionally providing zero-touch configuration. Serverless databases can be easily integrated into STORE as an underlying storage via a pluggable translation component.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

STORE Lifecycle Phases and Architecture Overview

STORE aims to achieve the overarching goal of paradigmatic unification of compute and storage for serverless functions. Just as when writing a function handler, developers do not need to think about where this function code will run, provisioning of compute resources, or scaling. In essence, they only need to think about what the function should do. Analogously, STORE abstracts the notion of storage for developers, condensing it to the sole decision of wanting to store a data instance. No need to think about the storage location, storage provisioning, or storage scalability.

To realize our vision of self-provisioning storage, we present lifecycle phases that enable self-provisioning, as well as an overview of the architecture that implements them.

4.1 Self-Provisioning Lifecycle Phases

As shown in Fig. 4.1, STORE self-provisioning happens in five phases: Preparation, Resolution, Permission Management, Selection, and Translation. These phases are designed to automate provisioning, enforce security, and dynamically adapt storage. Each phase is responsible for specific tasks, as outlined below.

4.1.1 PREPARATION

In this phase, the groundwork for Self-Provisioning is laid out. The aim is to handle all tasks that can be done statically at deployment time. For example, it is essential to ensure that storage systems (e.g, MinIO) are in place and that communication with them is possible. Tasks in the Preparation phase do not introduce runtime latency directly; therefore, it is advisable to pull as many tasks as possible into this lifecycle phase. Tasks in the preparation phase also act as a source of information for tasks in the resolution phase. This, for example, includes outputs of provisioning tasks concerning usage quotas, reachability, and access tokens for the provisioned systems.

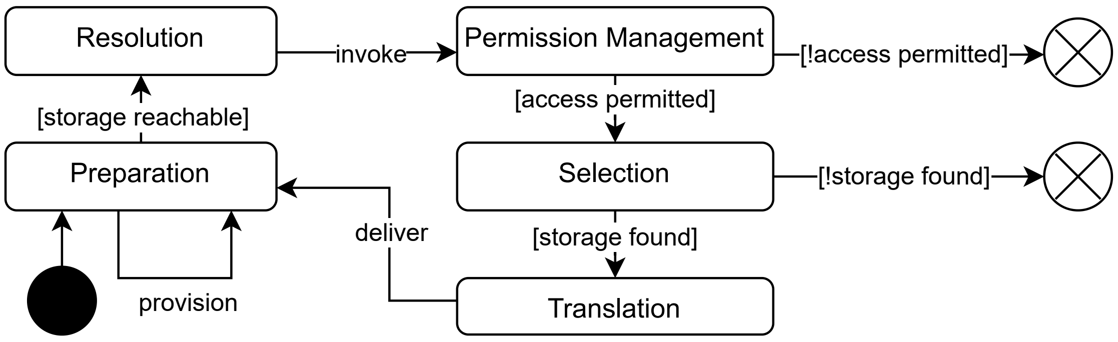


Figure 4.1: Self Provisioning Lifecycle Phases

4.1.2 RESOLUTION

This phase is a part of enabling Zero-Touch/Zero-Configuration by ensuring that functions can communicate with STORE without the developer needing to provide connection details or authentication credentials. This can, in part, be done statically by instrumenting resources. Data that changes for every function call can only be resolved dynamically. Resolution encompasses the dimensions of resolving connection and communication information, as well as resolving identity and security information.

- **Connection and Communication Information** includes service reachability, communication modalities, and resilience in communication. This information is classically provided by a developer in the form of configuration data. To eliminate this effort, the information needs to be fully derivable from earlier phases and the current system state.
- **Identity and Security Information** is typically provided in the form of credentials, tokens that are manually wired into functions. Identifying the identity of a function typically includes a name, a position in the system (i.e., the location in a workflow), as well as identities linked to the current execution, such as a user ID or a workflow execution ID.

4.1.3 PERMISSION MANAGEMENT

Two modes of permission management can be distinguished. One type, handled directly STORE, enables uniform access checking across all storage. The second type, relayed onto the underlying storage, ensures that other systems accessing the underlying storage directly experience a similar level of security as when using it through our solution. Permissions are managed according to different security levels, ensuring that only specific functions or workflows have access to the data. One model for security levels is described below:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **FUNCTION**. Restrict access to data instances to the same logical function instance. This can be the default security level for functions. Repeated execution of a function can all retrieve data stored by the function previously.

  Considering further tightening access, for example, to only encompass access within the same function execution, it quickly becomes clear that this level of security is too restrictive. Functions maintain data in memory during execution. After the execution, no other process can ever access the data instance again. There are clearly only a limited number of use cases that justify having this maximally restrictive default.

- **WORKFLOW.** For functions placed in a workflow, this security level can enforce that only functions within the same workflow can retrieve data instances stored under this security level. Considering use cases such as data passing between functions, there is merit in further tightening this security level.

- **WORKFLOW_EXECUTION.** This security level includes only functions assigned to the same execution of a workflow. This is the default for any function in a workflow context. Use cases, such as data passing between workflow steps, are an example of what this level can be used for.

- **PUBLIC.** Makes the data instance available to everyone. This security level can be explicitly set or inferred by noticing that a data instance needs to be globally available.

- **CUSTOM.**For maximum configurability, it should also be possible to define or infer, for example, a list of functions that are allowed to access a data instance. This security level can be modeled by a lock-and-key system, where a data instance can have a list of alternate required locks, and retrieval operations provide a list of identities (keys).

4.1.4 SELECTION

Based on selection criteria such as the size of data to store, usage statistics, or based on the structure of the data (for example, distinguishing between tabular and binary data), a suitable type of underlying storage is selected. Switching storage when data changes is also possible. At the core of the selection phase, the *Dynamic Storage Selection* mechanism is utilized. The inner workings of this mechanism are detailed in Section 5.1.

The importance of an abstraction that allows for selection and switching between multiple storage types becomes clear when considering the design principles of Multi-Level Agnosticity, Cost/SLO-Awareness, and Self-Optimization.

Multi-Level Agnosticity requires that it does not matter which storage solution is chosen, as well as which platform is chosen. When choosing a vendor-locked storage solution, it becomes impossible to freely choose the implementation platform, violating

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

the design principle of Multi-Level Agnosticity. Had the vendor-locked storage solution been hidden behind a storage type abstraction, switching providers (and if necessary also storage solutions) could be done without additional development effort.

Cost/SLO-Awareness requires the retrieval and management of high-level SLOs and different types of costs. For heterogeneous storage systems, each different type of storage has to be fitted into the model of Cost/SLO-Awareness. Abstracting storage allows for measuring and reacting on a unified model of SLOs and costs at the abstracted storage level, making it easier for a system to fulfill the design principles of Cost- and SLO-Awareness.

Self-Optimization involves refining the provisioning of the underlying storage system and switching to a different storage option in case suboptimal usage is detected. The requirement of dynamically switching storage solutions without involving developers already necessitates having an abstraction of storage that allows seamless switching.

4.1.5 TRANSLATION

In this phase, storage has already been selected. A received request has to be translated into $1-n$ new instructions for storing or retrieving data from the underlying storage. Translation can happen directly, such as when storing tabular data in a column store, where minimal extra translation effort is needed. Storing data with a structure that is not the primary objective of a storage system is often possible, but it requires additional effort. For example, storing chunks of binary data in a column store requires careful serialization and creation of suitably typed columns. Deepening the pool of selectable underlying storage solutions by also including the storage of data types not natively supported is beneficial, as it reduces the number of parallel solutions that need to be provisioned to support all data types. Ideally, every storage system supports all data types, reducing the minimum number of overall underlying storage systems required to one.

4.2 STORE Architecture Overview

Fig. 4.2 shows the overall architecture of STORE. The system is designed as a set of modules on top of an orchestrator and integrated with a serverless platform. Pluggable components are introduced to facilitate integration with different serverless platforms and storage technologies.

The control plane ensures the system is operated in the correct state. This includes ensuring communication pathways exist from the client to the storage, that the underlying storage is provisioned correctly, and that requests are routed to the called functions. It is comprised of a Watcher monitoring serverless resources, a STORE-Service Manager provisioning the STORE-Service, and a Storage Manager for provisioning underlying storage. The data plane, consisting of Client-SDK, Auto-Migration Framework, and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

STORE-Service, manages dynamic storage selection and is crucial for enabling zero-touch configuration.

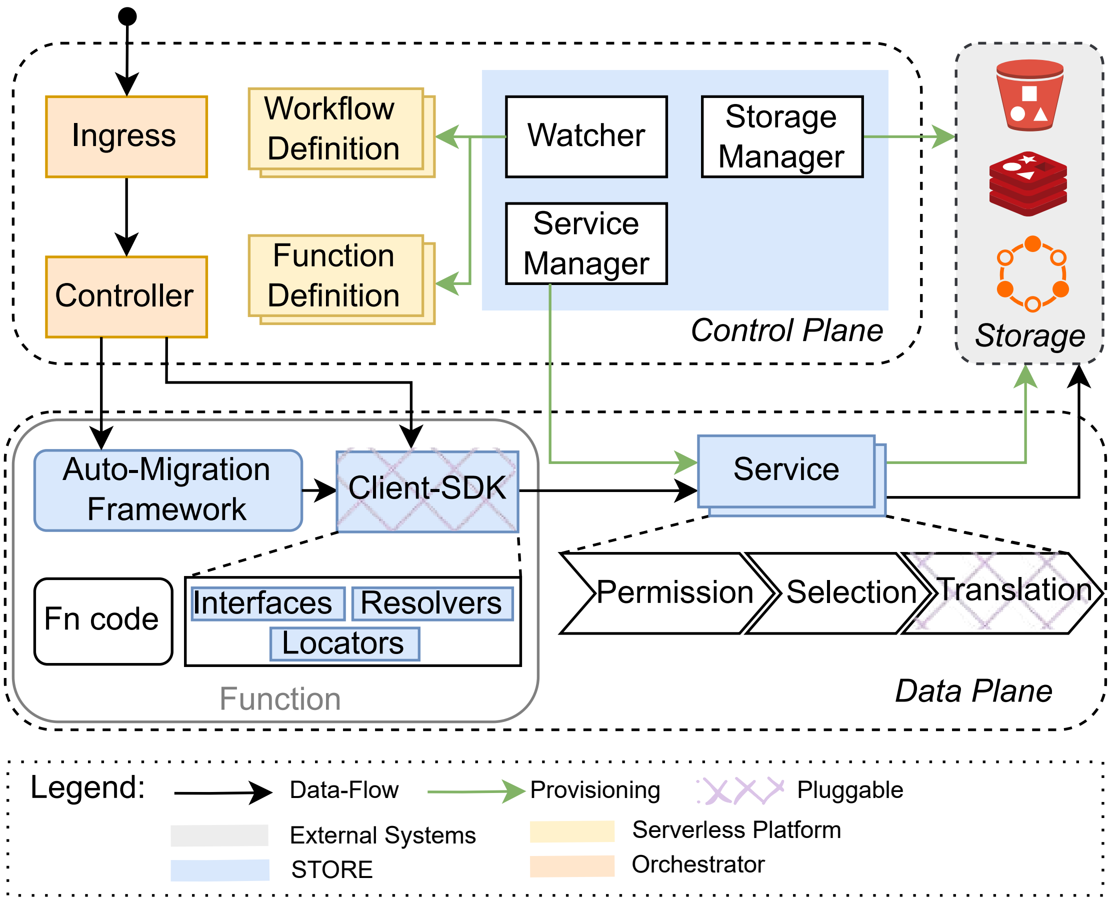


Figure 4.2: STORE Architecture Overview

In the following, we describe the six components that comprise STORE: Watcher, Storage Manager, STORE-Service Manager, STORE-Service, Client SDK, and Auto-Migration Framework.

4.2.1 Watcher

The Watcher component connects to the underlying orchestrator and interacts with resources provided by the serverless platform. It aims to instrument serverless resources, such as functions and workflows, with the information required for Zero Touch Configuration. Configuration information is injected into serverless resources in a way that allows it to be picked up by the Client-SDK. Utilizing events from the orchestrator, changes in the STORE-Service configuration, as well as addition or manipulation of serverless resources, trigger the reinjection of configuration information.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Information Gathering is performed by the Watcher through monitoring of the system state and listening for events from the orchestrator. Integration with the serverless platform can be achieved through common resource definitions or by calling the serverless platform's APIs. The Watcher mostly handles statically available configuration and provides information to other decision-making components that can then act at runtime.

4.2.2 Storage Manager

The Storage Manager performs provisioning of underlying storage systems. It is integrated with the orchestrator through the definition of resources that depict storage. Those resources serve both as a definition of the desired state of the underlying storage system and collectively as the pool of available storage for the STORE-Service.

The Storage Manager handles all statically doable provisioning tasks concerning the storage by enforcing a defined state of the storage systems and reacting to events received from the operator. Information relevant to connecting to the storage, current cost, and SLO metrics can be propagated through the storage manager.

4.2.3 STORE-Service Manager

It manages configuration and performs provisioning of the STORE-Service instances. Receives events about configured storage and provides it to the STORE-Service. It is integrated with the orchestrator through resource definitions. An instance of the STORE-Service is created for each tenant of the system and linked to the available storage systems for this tenant.

The STORE-Service Manager is also responsible for handling and provisioning the persistence used by the STORE-Service internally. Different strategies can be implemented, as already described earlier. Via resource configuration, one of the persistence strategies can be selected.

4.2.4 STORE-Service

The STORE-Service lies at the heart of the system. It is responsible for the lifecycle phases of permission management, dynamic storage selection, and translating requests into the APIs of concrete storage systems. It is also responsible for validating the state of the underlying storage systems and for provisioning resources on them. The Translation part of the STORE-Service is designed as a pluggable set of components, allowing for quick addition of additional types of underlying systems.

Permission Management

In addition to the security levels, we define two subcomponents that handle permission management: the AuthChecker and the PermissionStore.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

AuthChecker The `AuthChecker` defines the access control logic based on caller identities and security levels. It provides a uniform contract for authorization checks with three core operations:

- **Storing** (`canStore`) Determines whether a caller, identified by a set of identities, is allowed to create or store new data under a specified security level.
- **Updating** (`canUpdate`) Evaluates whether a caller is permitted to modify existing data. The decision is made by comparing the caller's identities with the identities and security level already associated with the data.
- **Retrieving** (`canRetrieve`) Governs read access, verifying if the caller can access data given its existing security level and associated identities.

PermissionStore Permissions are persisted through the `PermissionStore` interface, which associates a resource identifier with its stored permissions. The interface provides three methods: writing, retrieving, and existence checking. The `put` method saves the permissions for a given key, the `get` method retrieves them, and the `contains` method checks whether a resource has any permissions stored.

Permissions themselves are represented by two pieces of information: a *security level*, and a list of *identities* associated with the resource.

Translation

The service uses storage providers as an abstraction for interacting with storage backends. The configuration is passed by the STORE-Service Manager according to the configuration of the StorageProvider resources. Multiple provider libraries can be integrated as pluggable components.

The providers handle storage and retrieval of data as well as the necessary provisioning of the implemented storage backends. A common interface that all providers need to fulfill must be extracted.

4.2.5 Client-SDK

It is responsible for enabling Zero-Touch integration when developing serverless functions. Retrieves data about the current system state, the location of the STORE-Service, and data about the context in which the function is run. The Client-SDK is designed as a high-level interface and a set of pluggable bindings that integrate with different development environments. The core interface stays the same, while binding libraries are woven in at build time to create concrete programs that can run in diverse serverless environments and with diverse base technologies. The interfaces subcomponent encompasses various ways of interacting with the SDK, such as via a client or a map-like interface. Resolvers provide information about the function in the context of the system, for example, the function or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

workflow names, while locators allow the client to locate other system components, such as the STORE-Service.

Client Interface

The simplest client interfaces can provide two styles of methods: flat methods for common use cases, and overloaded request-based methods that support provider-specific options. Retrieval follows the same pattern. To increase ease of use, fluent builder-style methods can be introduced.

When developing a function, the client interface is provided from a context. Under the hood, the STORE-Service is located, identities are resolved (for functions statically, for workflows dynamically), and the appropriate networking is used to transmit the requests and responses.

The key thing is that everything happens automatically following the principles of SPI.

Data-Structure Interface

Data-structure interfaces, for example, a map-like associative interface, where storage and retrieval are abstracted behind the `put` and `get` methods familiar from map data structures, provide an even higher level of abstraction. In addition to the automated mechanics described above, a data structure interface manages concurrency by transiently retrieving and checking the data version on every `get` operation. `Put` operations for the same key, then attach the previously saved data version to the storage request. Challenges like serialization also need to be handled transparently for the user.

This interface eliminates the notion of storing something completely and replaces it with employing a map data structure that many users are familiar with.

4.2.6 Auto-Migration Framework

It allows for seamless migration from a wide variety of existing storage code to STORE by providing interface mimics for storage-specific libraries used in existing functions. Integrates with the Client-SDK by mapping used function calls to corresponding Client-SDK calls. A comprehensive set of Auto-Migration Libraries comprises the Auto-Migration Framework that enables backward compatibility.

Diverging Featuresets. Fully fledged storage solutions offer a broad interface with numerous specialized functionalities. STORE does not yet support all functionalities natively (and by design, it never will, due to its aim for a generic least common denominator (LCD) interface). Here, different strategies can be employed:

- **Replace by No-OPS.** If the functionality is obsolete since it is already handled by STORE, for example, provisioning operations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- **Direct mapping.** If STORE natively supports the functionality, calls can be redirected to the corresponding STORE functionality, such as storing files or tabular data.
- **Indirect mapping.** If the functionality is not directly supported by STORE but can be emulated using helper data. Here we can again make a distinction between using multiple keys to enable functionality, changing or wrapping the stored data, or enabling functionality client-side. For example, calculated metadata like tags or object categories can be stored by either enriching the object client-side or making multiple calls, storing an additional metadata object.
- **No mapping.** If there is no (easy) way of supporting the functionality using STORE, the error can either implicitly be ignored or explicitly handled by throwing an exception with a description of why this functionality is not supported. Checking if a mapping is possible at an earlier time can be achieved by following a slightly different approach that only mimics the parts of the interface that are actually mapped. Missing parts can lead to programs not compiling or programming environments detecting an error.

Maintenance and evolution. Maintaining a large number of zero-touch migration libraries is challenging because changes need to be incorporated as the external libraries update their interfaces. Strategies for creating and maintaining an ecosystem of such migration libraries are a topic for future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

STORE Mechanisms

In this chapter, we detail STORE's mechanisms: Dynamic Storage Selection and Zero-touch configuration. Dynamic storage selection picks the best storage provider based on function characteristics, and Zero-touch configuration enables self-storage provisioning without developer configuration inputs. We additionally describe a service breakdown of what happens to a request being processed by STORE, including Translation, Provisioning, Permissions, Metadata Management, and Connection Pooling. Finally, we go into detail on client-side processing, including Request Interception, Dependency Injection, and Environment Variable Injection.

5.1 Dynamic Storage Selection

Dynamic Provider Selection aims to choose the best storage provider implementation based on different criteria. The structure of the data (e.g, tabular, binary), the data size, and usage statistics are taken into account. Algorithm 1 shows how the decision which provider to use is made.

5.1.1 Selection Algorithm

The dynamic provider selection algorithm (Algorithm 1) receives as input the data size, data shape, and data structure. In case the data shape is not known, STORE picks the default storage selection blob storage type (Line 2). Additionally, STORE provides flexibility to enable the developer to preselect file-type storage providers (Line 4).

The algorithm runs on the *STORE-Service* and filters the list of available providers (Lines 5 and 8). Due to the negative effects on latency, when storing large chunks of data synchronously, providers that do not support request streaming are excluded above a certain streaming threshold (Line 7). Next, a translation penalty is applied for providers that do not natively support the data structure to be stored (Line 12). Finally, the best-scoring provider among the remaining candidates is selected (Line 17). A provider score can be adjusted by sampling latency, cost, or the current fulfillment status of a Service Level Objective (SLO).

Algorithm 1 Dynamic Storage Selection

```
Require: data.size $\neq \emptyset$, data.stream $\neq \emptyset$, data.structure
Require: providers (type, canStream, structure, stats)
Require: optional typePreselect
 1 if data.structure $= \emptyset$ then
 2     data.structure $\leftarrow$ blob
 3 end if
 4 if typePreselect $\neq \emptyset$ then
 5     providers $\leftarrow$ filter(providers, $p \rightarrow p.type$ = typePreselect)
 6 end if
 7 if data.size > STREAMING_THRESHOLD then
 8     providers $\leftarrow$ filter(providers, $p \rightarrow p.canStream$ = true)
 9 end if
10 for all $p \in$ providers do
11     if $p.structure \neq data.structure$ then
12         $p.score \leftarrow p.stats$ − TRANSLATION_PENALTY
13     else
14         $p.score \leftarrow p.stats$
15     end if
16 end for
17 bestProvider $\leftarrow$ argmax(providers, $p \rightarrow p.score$)
18 return bestProvider
```

5.1.2 Provider Switching

Switching providers can be handled in two different ways. Either a switch penalty is introduced to account for additional provisioning efforts, or a lazy approach is taken, where provisioning of the underlying storage begins once the decision to switch is made. When the provisioning is finished, the next request triggers the switch. This way, no latency increase is incurred for provisioning the new storage destination. Furthermore, a switching rate limit should be introduced to avoid resource churn that can occur with frequent switches.

5.1.3 Selection Storage

After provider selection, the selected provider has to be remembered. This is necessary to properly serve subsequent requests to the same data instance. Different considerations have to be taken into account when handling the storage of provider data or metadata in general.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Selection Storage Location

Where to store metadata depends on essentially the same criteria applicable to general storage selection. This enables the idea of recursively applying STORE to also store metadata with a data instance ID and retrieve it prior to making the actual request. While this would be possible, it would induce additional overhead (albeit small). A different approach is to pick the fastest available option and compromise on some other aspect. For example, in-memory storage of data is fastest but not persistent. Network locality can be utilized in combination with a fast storage type, such as Key-Value Store (KVS), to enable fast and persistent storage, albeit at the expense of limited scalability.

Selection Storage Format

The next question is about what information to store about the selection. While more information can be useful (for example, storing network addresses in case of configuration changes), each retrieval becomes slower. The simplest approach would be to store only the selected storage type identifier. This would place the responsibility to keep the link of storage provider ID to concrete storage solution consistent (i.e, if storage provider ID `1` is linked to storage `A`, and storage `A` changes network address, adapting that is fine. Linking storage provider ID `1` to storage `B` would be a violation)

5.2 Zero-Touch Configuration

The Zero-Touch Configuration mechanism is designed to eliminate configuration interaction between the developer and the system. In our system, zero-touch configuration comprises deployment-time instrumentation, runtime resolution, and runtime data passing.

5.2.1 Zero-Touch Configuration Overview

Fig. 5.1 illustrates the key steps comprising Zero-Touch Configuration, split between deployment time and runtime. ⓐ At deployment time, the STORE-Service Manager informs the watcher about the currently active STORE-Service instances and how they can be reached. ⓑ The watcher then instruments the function and workflow definitions with information about the location of the STORE-Service and the identities of the functions. This includes the function name and workflow name, if present. ⓒ The information is made available to the function instances in a way that the identity resolver and service locator components can retrieve it via runtime resolution. ① At runtime, a request arrives at the ingress (either as a function execution request or as an event for workflow execution) and is sent to the controller component, which provisions a function instance (in case of a cold start). ⓒ The identity resolver and service locator parse their respective information after startup. ② The controller then relays the request to a function instance. ③ The request reader picks up the request and starts runtime data processing, such as extracting the workflow execution ID. ④ The information is then

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

stored in the identity resolver for later pickup. Here, the scoping of the information is relevant, and isolation per request has to be provided. ⑤ After extracting the necessary information, the request reader calls the function handler. In the function handler, the Client-SDK can be used without any prior configuration. ⑥ The Client-SDK internally retrieves configuration information from the resolvers and the service locator. ⑦ Service communication happens through the Client-SDK, storing and retrieving data as instructed by the function. ⑧ After the function returns a response body, the response writer inspects the returned value and enriches it with additional runtime-specific information, finishing runtime data passing.

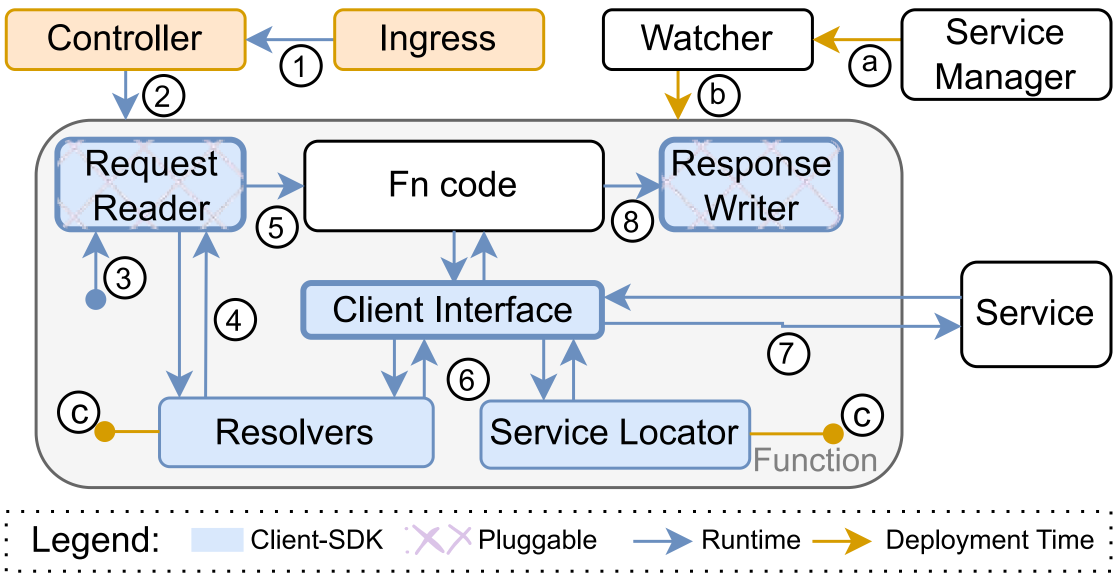


Figure 5.1: Zero-Touch Configuration Overview

5.2.2 Deployment Time Instrumentation

Deployment Time instrumentation is the static part of Zero-Touch configuration. It falls into the *Preparation* lifecycle phase. Instrumentation is necessary to dynamically react to the function's behavior and provide information about the service's location and identities to functions.

Different types of instrumentation may be necessary depending on the specific programming language and environment used. Ideally, instrumentation can be limited to plugging into already existing mechanisms and supplying information to functions. More involved cases may require the analysis and injection of code into functions to achieve the same goal.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.2.3 Runtime Resolution

Runtime Resolution falls into the *Resolution* lifecycle phase. The primary objective is for the function to automatically identify the service it should communicate with and also determine its own identity, capabilities, and position within a workflow. Resolvers and Locators are used in Runtime Resolution, relying on information or possibly hooks injected by Deployment Time Instrumentation.

Table 5.1 shows the API that should be made available to functions. The described methods are used internally by the STORE-Client SDK, but can also be used by developers explicitly.

Table 5.1: Resolver and Locator API

Function	**Description**	**Location**
`getIdentity(): List<Identity>`	Collects all identities that apply to an execution of a function.	Resolvers
`getName(): String`	Resolves the current function name	Resolvers
`getWorkflowName(): String?`	Resolves the current workflow name	Resolvers
`locate(): String`	Resolves the responsible instance of the Service	Locators

5.2.4 Runtime Data Passing

Runtime Data Passing is used to transfer data between functions transparently. This should be done without the developer noticing and without requiring developer actions. Our approach involves integrating into existing communication protocols through the efficient interception of requests.

Algorithm 2 shows how runtime data passing is achieved via the example of propagating a workflow execution ID. The algorithm acts as a wrapper around request processing. As an optimization, runtime data passing starts after reading the request headers (Line 1). First, it determines whether a workflow execution ID needs to be added by checking for its prior existence and positioning within a workflow (Lines 2 and 3). If no execution ID is present, a new one is generated (Line 6). Execution IDs must be unique at least during the current execution of the workflow, ideally until workflow execution-specific data is cleaned up, or even globally. A function handler is resolved, for example, according to the requested path (Line 10), and then called with a context containing all the information that has been read (Line 11). Efficient response handling is again enabled by inspecting the response after the header part is sent (Line 12). If the function handler has already specified a workflow execution ID, it is used; otherwise, the previously obtained ID is passed along automatically (Lines 13 to 17).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 2 Zero-Touch Runtime Data Passing

```
Require: request $req$, response $res$, context $ctx$
Require: ctx.functionName $\neq \emptyset$
hdrs $\leftarrow$ readUntilHeaderEnd(req)
if ctx.workflowName $\neq \emptyset$ then
    if hasHeader(hdrs, "ce-workflowId") then
        weId $\leftarrow$ getHeader(hdrs, "ce-workflowId")
    else
        weId $\leftarrow$ generateWeId()                          $\triangleright$ e.g., UUIDv4
    end if
    ctx.workflowId $\leftarrow$ weId
end if
h $\leftarrow$ resolveHandler(req)
callHandler(h, req, res, ctx)
rhdrs $\leftarrow$ waitUntilResponseHeaders(h, res)
if ctx.workflowName $\neq \emptyset$ then
    if $\neg$hasHeader(rhdrs, "ce-workflowId") then
        appendHeader(res, "ce-workflowId", ctx.workflowId)
    end if
end if
commitResponseHeaders(res)
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

STORE Prototype Implementation

The implementation of STORE is structured into three main subprojects, each addressing a different layer of the system. The first is the STORE-Operator, a Kubernetes operator built with *kubebuilder*[1] that is responsible for managing the STORE-Service and manipulating Knative services. The second is the STORE-Service, which handles provisioning, provides the external API, and performs the translation to underlying storage backends. The third is the STORE-SDK, designed to integrate with *Quarkus Funqy* and serve as the primary interface for function developers. As part of the SDK project, a zero-touch migration library has been implemented for MinIO, enabling seamless migration of existing applications without requiring code modifications.

All subprojects have been transferred into a single monorepo. This monorepo serves as the basis for the references included in Appendix A.

This chapter first describes each of the subprojects in detail and subsequently provides an example that demonstrates how STORE can be applied in practice.

6.1 Operator

The STORE-Operator is the component that lays the base of the system and enables seamless and zero-touch integration by manipulating the definitions of Knative Services. Using CRDs, the configuration and storage backends of the STORE-Service can be configured. Additionally, the storage of metadata for the STORE-Service via Redis can be configured through the Operator. The source code can be found in Appendix A.1.1.

[1] `https://github.com/kubernetes-sigs/kubebuilder`

TU WIEN Bibliothek, Your knowledge hub

6.1.1 Resource Definitions

At the core of the operator is the STOREService Custom Resource Definition (CRD). Listing 2 shows the Go type definition, including references to StorageProvider CRD and how the Redis sidecar configuration is defined. The type definition uses the `json` struct tag to specify how serialization is performed.

Listing 2 STORE-Operator STOREService type

```go
type STOREServiceSpec struct {
 Image                string                            `json:"image"`
 Port                 int32                             `json:"port"`
 ServiceType          string                            `json:"serviceType"` //
 ↪  ClusterIP, NodePort, LoadBalancer
 StorageProviderRefs []corev1.LocalObjectReference
 ↪  `json:"storageProviderRefs,omitempty"`
 RedisSidecar         *RedisSidecarSpec
 ↪  `json:"redisSidecar,omitempty"`
}

type RedisSidecarSpec struct {
 Image          string   `json:"image,omitempty"`
 Persistent     bool     `json:"persistent,omitempty"`
 MemoryLimit    string   `json:"memoryLimit,omitempty"`
 CPURequest     string   `json:"cpuRequest,omitempty"`
 AdditionalArgs []string `json:"additionalArgs,omitempty"`
}
```

The `StorageProviderRefs` reference the second CRD used to define the storage backends in a generic way. The type definition shown in Listing 3 shows the rather Spartan definition. The dynamic handling of providers is implemented via a `Registry` where handlers can register themselves. At runtime, the concrete handler is determined via the `Type` of the Storage Provider and queried from the registry.

Listing 3 STORE-Operator StorageProvider type

```go
type StorageProviderSpec struct {
 Type string `json:"type"`
 // Provider-specific configuration as raw JSON
 Provider runtime.RawExtension `json:"provider,omitempty"`
}
```

Listing 4 shows how a handler for MinIO can be registered at startup, including the functions that the handler offers. `Handle` is used in the control loop of the controller. In this function, provisioning of resources on the cluster can be done. For example, a new standalone MinIO instance could be provisioned here. In the case of attaching an existing MinIO, the `Handle` function does not need to do anything. The `Config` function produces information that can be passed on to the STORE-Service. A similar handler was defined for Cassandra.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Listing 4 STORE-Operator Handler implementation

```go
type MinioHandler struct{}

func (m *MinioHandler) Config(ctx context.Context, sp
↪   *storev1alpha1.StorageProvider) (map[string]string, error) {
        var cfg configs.MinioConfig
        if err := json.Unmarshal(sp.Spec.Provider.Raw, &cfg); err != nil {
                return nil, fmt.Errorf("invalid Minio config: %w", err)
        }
        configMap := map[string]string{
                "type":       "minio",
                "endpoint":   cfg.Endpoint,
                "accessKey": cfg.AccessKey,
                "secretKey": cfg.SecretKey,
                "useSSL":     strconv.FormatBool(cfg.UseSSL),
        }
        return configMap, nil
}

func (m *MinioHandler) Handle(ctx context.Context, sp
↪   *storev1alpha1.StorageProvider) error {
        var cfg configs.MinioConfig
        if err := json.Unmarshal(sp.Spec.Provider.Raw, &cfg); err != nil {
                return fmt.Errorf("invalid Minio config: %w", err)
        }
        return nil
}

func init() {
        providers.Register("minio", &MinioHandler{})
}
```

6.1.2 Controllers

Each of the previously described resources has a corresponding controller where the resources are reconciled. In addition to controllers for the CRDs managed by the STORE-OPERATOR, there is an additional controller watching Knative CRDs to enable zero-configuration integration. Each of the controllers is described below.

STOREService controller This controller reconciles instances of the STORE-SERVICE, including the configurable Redis sidecar for persistent metadata storage. Furthermore, this controller reads all applicable resources of type StorageProvider and collects their respective connection configuration via the `Config` function. The aggregated configuration data is passed to the service via an environment variable. Alternative approaches, like attaching the configuration to the service via a ConfigMap, are easy adaptations. By saving the reachability details of the provisioned service into the resource status, the controller creates the contact point where the current location of the service can be read.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

StorageProvider Controller The provisioning of underlying storage backends is done in this controller by calling the `Handle` function of the defined StorageProvider resources. Since this kind of provisioning was not the focus of this thesis, the functionality of the StorageProvider Controller is kept simple.

KsvcWatcher Controller The concept of Zero-configuration (from a user's perspective) requires finding an alternative method of wiring a client to the STORE-Service. This is especially challenging when multiple instances of STORE are deployed in a cluster. The `KsvcWatcher` controller plays a crucial role in achieving Zero-configuration for STORE. It injects the required environment variables directly into the `ksvc` CRD of Knative, ensuring that functions are automatically provided with the information needed to connect to the correct STORE-Service. By also watching `Sequence` and `Parallel` CRDs, as well as the STORE-Service itself, the controller is able to inject metadata such as the function name, workflow name, and STORE location into the runtime environment of each function. Support for multiple STORE instances is realized by leveraging Kubernetes namespaces.

Despite enabling Zero-configuration, this static approach introduces some limitations. First, only a single STORE instance can be supported per namespace. More elaborate load balancing would be needed to mitigate this issue. Second, each function can currently be associated with only one workflow, limiting reuse across multiple compositions. This is an inherent limitation of the chosen approach. Dynamic passing of configuration data would be needed to enable function reuse. Finally, since environment variables are injected into the `ksvc` resources, changes in the StorageProviders can trigger reprovisioning of many functions.

6.1.3 Summary

Figure 6.1 gives an overview of the CRDs and controllers that make up the STORE-Operator. The figure shows the interactions between the controllers and several types of resources, including the CRDs defined by the operator, as well as those of Knative, and the base resources defined by Kubernetes.

6.2 Service Component

The STORE-Service is the component at the core of storing things. It provides an interface for clients over HTTP with different endpoints for storing and retrieving file, tabular, and dynamic data. The service also handles concurrency control, metadata storage, permission management, and selection of the storage backend for dynamic data.

The service project is set up as a Quarkus Kotlin project. The source code can be found in Appendix A.1.2. Quarkus provides a quickstart template that was used to kick off the project [2]. Gradle (using Kotlin as the configuration language) was used to configure the

[2]`https://code.quarkus.io/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

project. Additional helper tasks for building the project in different modes (native and JVM) and pushing the project to a Docker registry were added.

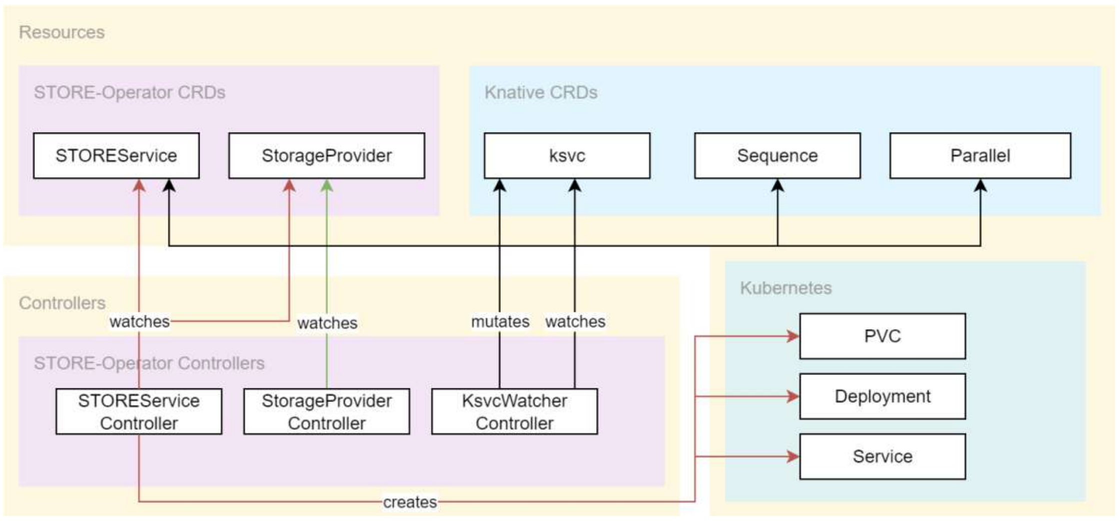


Figure 6.1: Visualization of the STORE-Operator interactions

6.2.1 Providers

In the scope of this thesis, two storage providers (MinIO and Cassandra) were implemented. A common interface that all providers fulfill was extracted. The interface is mainly used to identify storage providers by their names. Below, the implemented storage providers are described in more detail.

MinIO provider

The MinIO provider wraps the Quarkus MinIO Client Extension to communicate with the storage backend. It additionally handles creating buckets dynamically. File storage and retrieval are implemented via streams to integrate well with request streaming in the HTTP layer. The provider also offers wrapper methods for synchronous access by consuming the entire stream and wrapping the results as a byte array. In general, the used MinIO client already accurately depicts the envisioned interface, so little adaptation was necessary.

Cassandra provider

Attaching the Cassandra storage backend was implemented using the Apache Cassandra Driver (formerly maintained by DataStax). Since Cassandra is based on sessions with persistent connections and driven by Cassandra Query Language (CQL) queries, more effort was required for translation and connection management.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

The Cassandra client allows configuring different execution profiles for queries that contain, among others, the timeout configuration for queries. Due to Data Definition Language (DDL) statements taking longer than Data Query Language (DQL) statements, different execution profiles were configured for slow and fast statements.

The provider manages automatic provisioning of the following Cassandra components:

- **Cassandra Keyspace.** This component can be compared to the database abstraction in a relational Database Management System (DBMS). All data saved to Cassandra by STORE is saved to the same keyspace. Creation and existence check of this keyspace is performed when the provider is initialized. For this to work, a session in the default keyspace is started, and after ensuring the keyspace is configured correctly, it is destroyed again. Operations related to creating keyspaces are executed with the slow execution profile.
- **Cassandra Tables.** Tables are analogous to their counterpart in relational DBMS. STORE manages the creation of tables by dynamically checking for table existence when something should be stored. Tables are created by dynamically constructing CQL `CREATE TABLE` statements. Tables are always created with an `id` column where the identifier of the tabular data to store is saved.
- **Cassandra Columns.** Again, analogous to relational tables, columns also need to be created and updated when using Cassandra. The tricky part is that there is a distinction between altering tables and creating tables with columns. This is again done via existence checks and corresponding translation to different types of DDL. Translation of Kotlin Types to Cassandra types can happen in two ways. Either the user specifies the type to use explicitly when calling the STORE-SDK methods or the type is inferred automatically by matching the dynamic Kotlin Type to a Cassandra Type. Additional handling to support different types of binary data was implemented.

When retrieving tabular data via a tablename and row-key, a DQL statement is dynamically constructed containing either all columns or only a subset if specified explicitly by the user. When querying data, the column metadata in the form of a Time to Live (TTL) and modification timestamp can also be retrieved. Because querying happens via the `id` column, at most one dataset will be returned. This way, no pagination or alternative methods for handling large numbers of returned datasets are necessary. A big limitation is that, due to statements being transmitted via CQL, all of the data to be transmitted has to be available before the translation step. This means that there is no easy way to implement request streaming in Cassandra.

6.2.2 Permission Management

For permission management, the described security level concept was implemented. As a basis for identifiers Uniform Resource Names (URN) [rfca] syntax was used. In the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

STORE-SERVICE, the described AuthChecker and PermissionStore were implemented. These implementations are discussed below.

AuthChecker

The `SimpleInclusionAuthChecker` implementation provides a straightforward realization of the `AuthChecker` interface. Its design is based on two resource namespaces: function identities, prefixed with `urn:function:`, and workflow identities, prefixed with `urn:workflow:`. Authorization decisions for storing, updating, and retrieving resources are made by comparing these identities with the security level and the identities associated with the data.

For *storing* data, the policy is permissive in the case of public data, where no special identity is required. For the other security levels, data can be stored only if at least one identity of the relevant type is present in the caller identities (i.e, when storing with security level `FUNCTION`, at least one function identity needs to be provided).

For *updating* existing data, the rules are mostly based on simple inclusion. Public resources can be updated by anyone. For other security levels, inclusion checks are performed to ensure that at least one of the caller's identities matches an identity of the data. Preprocessing is necessary in some cases, for example, when matching for workflow identities, parts of the URN need to be extracted for string equality matches to work.

For *retrieval*, the rules are identical to those for updating. Thus, read and write access are aligned in this implementation.

PermissionStore

Two implementations are provided: an in-memory version (`InMemoryPermissionStore`) using a thread-safe `ConcurrentHashMap` for fast, ephemeral storage, and a Redis-backed version (`RedisPermissionStore`) that encodes permission data in Redis, enabling persistence and sharing across multiple instances. The concrete implementation to use is decided at runtime via a Contexts and Dependency Injection (CDI) producer.

Wrapping both the PermissionStore and AuthChecker, a simple AuthorizationService was implemented, allowing to check if data can be stored or retrieved with a single function call, throwing `SecurityExceptions` in case of mismatches.

6.2.3 Concurrency

To coordinate concurrent access, the system employs a version-based optimistic concurrency control mechanism, formalized by the `ConcurrencyController` interface. The controller interface manages data versions and allows for short-lived locks during writes. The interface exposes methods to query and update versions, check version matches, and acquire or release locks. A workflow for ensuring consistency when writing data is prescribed by the controller and described below:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

1. Check for a fast-fail exit by comparing the caller's expected version against the current version.
2. Acquire a short-lived lock for the key.
3. Revalidate the version to prevent race conditions.
4. Perform the actual data write.
5. Update the version to reflect the new state.
6. Release the lock.

A utility method, `handleVersionedWrite`, encapsulates the entire write protocol, raising `VersionConflictException` on mismatched versions or `ResourceLockedException` if the key is currently locked. Reads require no locking, but the version has to be fetched before the data is read to ensure consistency. This design ensures that no reader observes a version newer than the data actually written.

The `ConcurrencyController` interface has two concrete implementations, both of which realize the same optimistic concurrency protocol but with different backends. The `InMemoryConcurrencyController` uses `ConcurrentHashMaps` to track versions and `ReentrantLock` objects to depict write locks.

Secondly, the `RedisConcurrencyController` relies on Redis to support distributed coordination. Locks are emulated using the `SETNX` command to acquire a key if not already present, combined with `PEXPIRE` to enforce a TTL, ensuring that locks are eventually released even in the event of a client failure.

At runtime, a CDI producer is responsible for selecting which implementation to use.

6.2.4 Metadata

The STORE-Service provides a generic interface for metadata handling, abstracted by the `MetadataStore`. Each resource is identified by a globally unique `Key`, which provides an encoded string representation to prevent accidental collisions or deliberate injection attacks (e.g., by normalizing separator characters). Metadata itself is represented as a flat map of string key-value pairs associated with each resource key.

The interface supports basic Create, Read, Update, and Delete (CRUD)-style operations on metadata: setting or replacing individual fields, setting multiple fields at once, retrieving all metadata for a key, retrieving a single field and removing either all metadata or a specific field. An optional method, `listKeys`, allows enumerating all resources with metadata entries.

In STORE, the metadata store is primarily used to record the storage *provider* chosen for each resource key. Tight integration is achieved by introducing Kotlin extension

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

methods on the MetadataStore. In this way, provider selection becomes part of the generic metadata infrastructure rather than requiring a separate specialized mechanism.

As with permission and concurrency management, two implementations were created for in-memory and Redis-backed metadata management.

6.2.5 HTTP layer

The HTTP layer of the STORE-SERVICE is split between two approaches. The first part is implemented using `io.quarkus:quarkus-rest`. The second part utilizes the `io.vertx.ext.web.Router` directly. The use of different libraries to handle requests stems from problems encountered when implementing request streaming with `quarkus-rest`. Below, the different approaches and the endpoints implemented with each of them are described in detail.

Quarkus-Rest endpoints

All Quarkus-Rest endpoints are implemented following the `jakarta.ws.rs` specification[3]. Each request may carry an `Authorization` header, which contains the identities of the caller. During request processing, the previously described components – AuthorizationService, ConcurrencyController, MetadataStore, and the providers – are used to implement business logic. Processing errors are transmitted via HTTP status codes. The following endpoints were implemented this way:

- **Retrieving files.** Checks if the caller has the permissions to retrieve the requested file, the requested key is stored with a file provider, retrieves the current version from the concurrency controller, translates the request using the MinioProvider, and returns the resulting stream.

- **Retrieving tabular data.** Follows the same overall procedure as retrieving files. Returns `JSON` data as response.

- **Retrieving dynamic data.** Instead of validating that the provider where the data is stored, the provider is retrieved from the metadata store, and depending on the result, the data is retrieved differently. As the common interface binary data is returned, it is either streamed to the client or the entire response is read and wrapped as a stream for returning.

- **Storing tabular data.** Checks for caller permissions and then uses the `handleVersionedWrite` utility function to save the row and the provider used.

[3]`https://jakarta.ee/specifications/restful-ws/4.0/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Vert.x Router endpoints

Request streaming as a mechanism needed to be implemented where possible. Through tests, it became clear that some component of the Quarkus HTTP stack internally reads the whole request before proceeding with deserialization. This disqualified Quarkus-Rest for handling endpoints where request streaming is needed, and an alternative solution had to be devised. Since Quarkus-Rest uses the Vert.x router under the hood, it was decided that plugging into the router manually at a lower level of abstraction was the way to go.

Combining low-level Vert.x and Quarkus-Rest turned out to be rather difficult, since Quarkus-Rest attaches itself at a low level as well. The solution was to manually register routes with a very low order number (≤ -1000000000), ensuring that the registered routes are hit before the Quarkus-Rest internal routes. While this approach solved the problem of the entire request being read, it also introduced the necessity of handling some HTTP behavior manually. For example, handling for the status code `100 Continue` [moz] had to be implemented manually.

The biggest manual implementation part was the custom multipart handler. Luckily, there are many implementations available [ore, goob] that could be adapted or taken inspiration from. The specification for the content type `multipart/form-data` [Mas] does not enforce a predefined order. If there is a file part in the request, it would be assumed to be the largest part (at least in the usage scenario for STORE, where it is). This enables easy optimization for request streaming. If all expected form parameters have already been read, the file part that follows can be streamed, and the request handler can be called before the entire request is read.

This is exactly what the implemented custom multipart handler does. It reads the request in chunks, and as soon as the file part begins, it calls a handler method with the read form data and an input stream containing the file chunks. The request body can process the data and translate the service calls from there. The following endpoints were implemented utilizing the custom multipart reader:

- **Storing files.** Manually extract headers from the request and set up the multipart handler. In the handler function, the caller permissions are checked, and then the `handleVersionedWrite` utility function is called to save the file and the provider used.

- **Storing dynamic data.** Same overall procedure as for storing files. Instead of selecting the file storage provider, dynamic provider selection is run. For providers that do not support request streaming, the whole request is read and passed to the provider.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.2.6 Performance Tuning

During testing of the STORE-Service, several performance bottlenecks were identified. To address these, targeted optimizations were applied at different layers of the system. Below, we describe each performance tweak in detail.

HTTP body and form attribute size. By default, Quarkus enforces limits on the maximum request body size and the maximum size of individual form attributes. During testing, these limits were reached when transferring larger files. To mitigate this, the configuration parameters `quarkus.http.limits.max-body-size` and `quarkus.http.limits.max-form-attribute-size` were both increased to 200 MB. The sizes were adapted to easily accommodate the largest size used in tests. At some point, larger files should be split into chunks.

Thread pool configuration. Another bottleneck observed during testing was related to the default Quarkus thread pool settings. By default, Quarkus uses a worker thread pool of size $max(200, 8 \times \#CPUs)$[4]. This limits the number of concurrent requests and adds overflowing requests to a queue. To remove this restriction, the maximum number of worker threads was increased by setting `quarkus.thread-pool.max-threads` to 1000000, and the internal request queue was disabled with `quarkus.thread-pool.queue-size` set to 0. This was done to immediately notice if the pool is full, since the requests fail immediately when the queue is disabled.

Cassandra profiles. As previously described, the Cassandra driver allows specifying execution profiles with different timeouts. Initially, only DDL operations were run with a larger timeout. The initial timeout for querying data proved to be too small for large column sizes and was increased to 30 seconds.

Redis and MinIO client configuration. Both the Redis and MinIO integrations rely on underlying HTTP clients whose connection pools require tuning under heavy load. For Redis, the `RedisOptions` were adapted by increasing the maximum pool size to one million connections (`setMaxPoolSize(1000000)`), while disabling the waiting queue (`setMaxPoolWaiting(0)`). The latter was again done to simplify identifying bottlenecks during testing.

For MinIO, the client is backed by `OkHttp`, where the dispatcher limits were raised to allow up to one million concurrent requests globally and per host (`maxRequests` and `maxRequestsPerHost`), and the read timeout was extended to 120 seconds to support large file transfers.

[4] `https://quarkus.io/guides/all-config`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.3 Client SDK

The STORE-SDK is the client-side component of STORE, providing seamless integration and a lightweight API while remaining multi-level agnostic. Two different interfaces are provided, one being a traditional client enhanced with Kotlin Domain-Specific Language (DSL). The second interface is based around an associative data structure or map-like object. Storage and retrieval are hidden behind `get` and `put` operations on the map object. Below the project setup, both interface implementations and some of the implementation challenges are described.

6.3.1 Project setup

The client SDK is organized as a Gradle multiproject[5], providing a modular structure that separates concerns and enables reuse across different components. Build and language conventions are enforced through custom Gradle convention plugins[6]. In particular, the `KotlinConventionPlugin` standardizes Kotlin compilation, dependencies, and testing setup, while the `KnativeFuncConventionPlugin` configures dependencies for building Knative-compatible serverless functions. It also registers Gradle tasks for building the projects and pushing the resulting images to a Docker registry. The top-level project was configured to automatically include function and workflow projects placed in the `functions` directory.

The multiproject contains the following subprojects:

- **api.** Contains model classes and interfaces to be used by clients. Could also be abstracted further into completely language-agnostic formats [swa]. Source code can be found in Appendix A.1.2
- **core.** Houses internal business logic that is applicable to all Kotlin bindings. Source code can be found in Appendix A.1.4.
- **binding-quarkus.** Used as glue to integrate STORE tightly with Quarkus projects. Provides clients via CDI and creates a concrete HTTP client using `io.vertx.mutiny.ext.web.client`. Source code can be found in Appendix A.1.5.
- **functions.** Wrapper project for the individual serverless functions and workflows. Source code can be found in Appendix A.1.6
- **minio-adapter.** Houses the zero-touch migration library for functions using MinIO clients. Source code can be found in Appendix A.1.7

[5]https://docs.gradle.org/current/userguide/intro_multi_project_builds.html
[6]https://docs.gradle.org/current/samples/sample_convention_plugins.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.3.2 StoreClient

The `StoreClient` interface follows the described client interface approach: flat methods (e.g., `storeFile`, `storeRow`, `retrieveFile`) for common use cases, and overloaded request-based methods (e.g., `store(StoreFileRequest)`) that support provider-specific options. Kotlin DSL extensions (`client.store { ... }`) increase ease of use. When developing a Quarkus function, the client interface is injected via CDI.

6.3.3 StoreMap

The `StoreMap` provides a map-like associative interface, where storage and retrieval are abstracted behind the `put` and `get` methods known from map data structures. The `StoreMap` uses dynamic storage via the `StoreClient` internally. Serialization is achieved via the Concise Binary Object Representation (CBOR) [rfcb] format, integrated via the Kotlin serialization library[7]. CBOR was chosen for its being lightweight to encode/decode and reasonably compact due to being a binary format.

6.3.4 Implementation Challenges

During the implementation of the STORE-SDK, challenges were encountered in terms of performance and technical complexity. Below, some of the challenges and their solutions are described.

Performance tweaks. Several optimizations were required to address bottlenecks in the function and SDK parts. Of the previously identified bottlenecks, the Quarkus worker pool size was increased, the queue for worker threads was disabled, and the maximum body size was raised. On the client side, the Vert.x `WebClient` was configured with an increased connection pool size of one million connections. Finally, the JSON parser was adjusted to support documents and strings up to 200 MB, preventing errors for larger payloads.

Request interception. Communication between functions in a workflow follows the CloudEvents Specification [git]. To read the current workflow execution ID and propagate it to successor functions, the request used to trigger a function in the workflow has to be intercepted. This way, handling of workflow execution IDs can be made transparent for the user. By default, however, Quarkus Funqy tries to abstract away implementation details of the underlying serverless platform. This makes it hard to interact with the internals by design. Luckily, the library used to integrate Funqy with Knative uses Vert.x internally. Vert.x (also luckily) is integrated with CDI, allowing the injection of the current HTTP request, which in turn provides access to the Vert.x routing context.

The problem is tackled by registering a route for all requests with a low order number (-10). The handler reads the `ce-workflowId` header from the request and extracts the

[7] `https://kotlinlang.org/api/kotlinx.serialization/kotlinx-serialization-cbor/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

workflow execution id, or creates a new one. A curiosity is that, depending on the header name, the request processing fails silently in Funqy (i.e, no - or _ in the workflowId part of the header name). The handler registered on the response, after the headers are written, adds the `ce-workflowId` header if it is not already present. In the context of registering routers, a `RoutingContext` is available. The read workflow execution ID is stored in the `RoutingContext`. From a request-scoped context, the stored value can then be retrieved via the `CurrentVertxRequest` bean.

Two caveats remain: Firstly, the newly introduced beans are request-scoped. This leads to complications with CDI scoping, as a bean with a short-lived scope must be injected into a bean with a longer-lived scope. This can be handled by using a CDI `Provider` to wrap the short-lived bean. Secondly, the implemented flavor of request interception only works with the CloudEvents binary content mode (where metadata is transmitted in HTTP headers). The structured content mode (where all data is transmitted in the request body) would require more intricate and lower-level request parsing.

Map interface design. When designing the map interface, it became clear that dynamic deserialization was needed. Achieving this using `kotlinx.serialization` requires access to a class object at runtime. Passing this class as a parameter into the `get` method is a possible solution. This, however, does not align well with the Kotlin programming style. To further streamline the data retrieval, extension methods that allow passing the class for deserialization as a type parameter were added. Listing 5 shows an example of how this extension was implemented.

Listing 5 Map interface extension method

```kotlin
inline fun <reified T : Any> StoreMap.get(key: Long): T? {
    return this.get(T::class, key)
}
```

6.4 Zero-Touch Migration Library

A Zero-Touch Migration Library is the intended method for integrating STORE with existing function code. In the scope of this thesis, a Zero-Touch Migration Library was implemented for integration with the Quarkus MinIO Client. Below, some challenges and strategies in implementing the library are described.

6.4.1 Project setup

The library is set up as a Kotlin Gradle project that can be added as a dependency to other projects. It contains an extracted `MinioClient` interface as well as an implementation of the interface using the `StoreClient`. Model classes are reused from the original library. The integration and wiring are done by making the `MinioClient` implementation available via a CDI `Producer`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.4.2 Implementation Challenges

Here, we present some challenges and their solutions encountered during the development of the library.

Interfaces and Classes. Some libraries may already provide interface definitions that can be reused for implementing a migration library. For example, in the case of the Quarkus MinIO Client, this is not the case. Only a concrete `MinioClient` is provided. The solution, although a little tedious, is to manually extract an interface for the concrete class (in our case, with the same name as the original class). This way, by replacing the import statements, the usage of a class is replaced by the usage of the interface with the same name. This makes it easier to manage the concrete implementation instantiated by employing CDI.

Model Classes. Most libraries rely on a large set of model classes. Depending on the realization of the target library, those classes may even include business logic or call back into the target library. Depending on how model classes are implemented, they can be reused entirely (if provided as a separate library and do not contain business logic), copied (if no separate library is available but no business logic is present), or rewritten entirely (if no separate library is available and business logic is present).

Diverging Featuresets. The Quarkus MinIO Client, for example, offers 65 methods. Since STORE does not support all of them, the previously described strategies were employed:

- **Replace by No-OPS.** For provisioning operations like `createBucket`.
- **Direct mapping.** For storage and retrieval like `getObject`.
- **Indirect mapping.** For functionality not directly supported by STORE, like `setObjectTags`, by storing an additional tags data instance.
- **No mapping.** If there is no (easy) way of supporting the functionality using STORE like for `getPresignedObjectUrl`, where calls throw an exception.

6.5 Complete Example

Finally, we provide a comprehensive example of how `STORE` can be utilized in a serverless function body. The example demonstrates both the `StoreClient` and `StoreMap`, as well as how the STORE-SDK packages can be integrated into a project.

As a first step, the dependency for STORE needs to be added to the project. This includes adding the API, as well as the binding for the underlying technology used. In our case, we add the Quarkus binding and make sure that no implementation-specific classes are referenced by marking the dependency as `runtimeOnly`. Listing 6 shows

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

what that looks like in the context of the project configuration. Note that the convention plugin `store.func` is used to manage the needed Quarkus Funqy dependencies.

Listing 6 Example project configuration

```
plugins {
    id("store.func")
}

group = "store"
version = "1.0.0-SNAPSHOT"

dependencies {
    implementation(project(":api"))
    runtimeOnly(project(":binding-quarkus"))
}
```

To use the `StoreClient` or `StoreMap`, they can be injected using CDI. Listing 7 shows how the client can be injected and Listing 8 shows how the map is injected. Via an additional annotation `@DefaultSecurityLevel`, the default security level to use can be configured. While STORE-SDK methods also allow providing the security level for finer control, specifying it once at the injection points reduces the needed effort.

Listing 7 Injection of the STORE client interface

```
@Inject
lateinit var client: StoreClient
```

Listing 8 Injection of the STORE map interface

```
@Inject
@DefaultSecurityLevel(SecurityLevel.WORKFLOW)
lateinit var map: StoreMap
```

Using the client in a function body is as easy as saying, "I want to store something." All a user needs to do is get the data ready and call a function. There is no need to set up anything, no need to know where the data is stored, and no need to provision any buckets or tables. Listing 9 shows how the Kotlin DSL interface for storing columnar data and the flat interface for dynamic storage work.

Usage of the map interface is demonstrated in Listing 10 where storage and retrieval using type parameters are shown. In the demo, two counters are retrieved with different key types (`String` and `Long`); then, some operations are simulated on the retrieved data before storing them again. In case of a concurrency violation, the function fails and is retried externally (i.e., via a workflow engine).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 9 Usage of the STORE client interface

```
@Funq
fun demo(input: String): String {
    val colResponse = client.storeRow {
        tableName = "test"
        rowKey = input.ifBlank { "test" }
        column("col1", 1, ValueType.INT)
        columns("col2" to "test2", "col3" to "test3")
    }
    if (!colResponse.success) return colResponse.errorMessage
    val colRetResponse = client.retrieveRow {
        tableName = "test"
        rowKey = input.ifBlank { "test" }
    }
    if (!colRetResponse.success) return colRetResponse.errorMessage

    val response = client.store("testsimple", input.toByteArray())
    if (!response.success) return response.errorMessage
    val retResponse = client.retrieve("testsimple")
    if (!retResponse.success) return retResponse.errorMessage
    return retResponse.data.toString()
}
```

Listing 10 Usage of the STORE map interface

```
@Funq
fun demo(input: String): String {
    val counter = map.get<Long>("counter") ?: 0L
    map.put("counter", counter + 1)

    var longKeyCounter = map.get<Long>(1) ?: 0L
    longKeyCounter*=2
    map.put(1, longKeyCounter)
    return "Success: ${counter + 1}, $longKeyCounter"
}
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Evaluation

This chapter contains the evaluation of STORE according to multiple dimensions. The evaluation is structured into four parts: *Usability*, *Performance*, *Scalability*, and *Backward Compatibility*. The performance and scalability analysis focuses on request traces, evaluating absolute and relative overheads, as well as CPU and memory usage obtained through Prometheus data scraping. Usability is assessed through the implementation of representative use cases, which exercise both practical features of the system and the reduction in implementation effort measured in lines of code. Finally, backward compatibility is evaluated by migrating an existing function based on the MinIO client to the STORE-SDK with minimal changes.

7.1 Overview

This section outlines the overarching goals of the experiments, as well as the methodology for calculating the baselines and the rationale behind the chosen metrics.

Goal. The goal of the experiments is to demonstrate that the proposed system alleviates developers of the burden of manually provisioning infrastructure without compromising the system's performance and scalability. By evaluating the system under varying workloads, we aim to show that it achieves these benefits without sacrificing efficiency or resource usage.

Baselines. For the Lines of Code (LoC) experiments, the baselines are Terraform (first-generation IaC) and Pulumi (second-generation IaC). The STORE implementations are compared against these baselines to quantify reductions in lines of code.

Tracing Overhead. Execution traces are collected at a low sampling rate (1%), with events buffered and flushed asynchronously to minimize interference. Lower sampling rates proportionally reduce overhead, and at our chosen frequency, the impact on latency and resource usage is negligible, as confirmed by control runs showing no measurable difference beyond normal variance.

Metrics. We report absolute overhead (in ms) and relative overhead (in % of the service when compared to an overall storage operation) to capture both the concrete performance impact and its proportional significance with respect to the baseline. CPU (in millicpu) and memory consumption (in MB) are measured to evaluate resource efficiency, ensuring that performance is not achieved at the cost of excessive resource use. Throughput (in Requests per Second (RPS)) is evaluated via load tests in which the request rate is systematically increased, while the effect of increasing input sizes (in KB) is evaluated in a separate set of performance experiments. Additionally, LoC measurements are used to compare the system's usability with that of the state of the art. Together, these metrics provide a comprehensive view of the system's usability, scalability, efficiency, and overhead in realistic usage scenarios.

Overhead Calculation. The absolute overhead is calculated as the total time from receiving the first byte of a request at the service until the last byte of the response is sent, minus the time spent waiting on the underlying storage solution. The relative overhead is then computed as:

$$\text{Relative Overhead (\%)} = \frac{\texttt{store-service}}{\texttt{overall}} \times 100 \tag{7.1}$$

where `store-service` is the absolute service overhead and `overall` is the total request execution time measured by the root trace at the Knative Activator. Detailed calculation steps and implementation are provided in the analysis tools in Appendix A.1.9.

Resource Usage. The resource usage metrics are:

- **CPU (millicores):** measured for the STORE-SERVICE and the functions over the course of the experiment.
- **Memory (MB):** measured for the STORE-SERVICE and the functions over the course of the experiment.

Metrics are scraped every 5 s; CPU is exported as 15 s averages (Section 7.2.4). For each experiment, we compute statistics over the *entire run* (from the first received request to the last response sent). In case multiple instances of functions or multiple different functions are involved in an experiment, the sum of the individual function metrics is taken. We do not measure the overall resource consumption of the system, as it would require carefully dissecting which supporting components, such as Knative components, storage components, and Kubernetes system components, to include or exclude. This would introduce significant engineering overhead without yielding any novel scientific results.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

To measure some of the success criteria, we need to define what "constant" and "at most linear" are. For this, we take a regression analysis approach [Bur01]. To account for small variations, we also include a definition of constant via a small Coefficient of Variation (CV) [Ste11]. Here, we use the CV as a measure of consistency. Approaches using linear regression to predict linear behavior and slopes are well established [WSAP17, YYR21][1].

Definition of Constant We define a series as *constant or decreasing* if it exhibits low variability or no statistically significant upward trend. Low variability is defined as a CV of at most 10%. If the CV exceeds this threshold, a linear regression of the series against the tested input variable is performed. The series passes the criterion unless the regression slope is positive and statistically significant at the $\alpha = 0.05$ level (one-sided test for slope > 0).

Definition of at most Linear To evaluate whether a metric grows at most linearly with the tested input variable, the following approach is used: The measured values are first divided by the corresponding input variable ($\frac{Y}{X}$), producing a *per-unit* series. A linear regression is then applied to this normalized series. If the regression slope is positive and statistically significant at $\alpha = 0.05$ (one-sided test for slope > 0), the metric is classified as exhibiting superlinear growth and the criterion fails. Otherwise, if the slope is zero, negative, or not statistically significant, the growth is considered at most linear.

LoC calculation We exclude from the LoC metric comments and blank lines. This leads to the following formula for calculating LoC [SKKC11]:

$$\text{LoC} = \text{NCLoC} - \text{BLoC} \tag{7.2}$$

The parts used in the calculation are Non-Comment Lines of Code (NCLoC) and Blank Lines of Code (BLoC).

7.2 Experimental Setup

This section describes the necessary setup for reproducing the experiments, including how the cluster was set up, how the individual services are configured, how the monitoring and measurements are taken, and a visualization of the whole environment.

[1]Although caution is warranted when applying linear regression without careful consideration [CMB25], our aim here is restricted to assessing simple slopes, not to achieving a rigorous trend analysis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

7.2.1 Infrastructure

Hardware and Virtualization Layer

The experiments are conducted on a host with 32 CPUs, 377.79 GiB of RAM, and all storage provided by SSDs. The host runs Linux kernel 6.8.12-4-pve (release date: 2024-11-06) with Proxmox Virtual Environment (PVE)[2] 8.3.0 (manager version: pve-manager/8.3.0/c1689ccb1065a83b). Virtual machines for the Kubernetes cluster are provisioned and managed through Proxmox.

The cluster is comprised of 7 VMs, including one control plane VM and six worker VMs. The memory and CPUs are tuned to provide the services, functions, and underlying storage with similar amounts of compute resources. The VMs are all running `Debian GNU/Linux 12`[3]. Storage is on SSDs attached via `ZFS`. Table 7.1 shows the distribution of compute resources across the VMs.

Name	CPUs	Memory (GiB)
controlplane	4	8
general-0	9	50
general-1	5	50
general-2	5	50
general-3	3	25
general-4	3	25
general-5	3	25

Table 7.1: VM Resource Allocation

Kubernetes Cluster

The Kubernetes cluster runs version v1.32.2 with `containerd` 1.7.27 as the container runtime. It was provisioned using a modified version of the ClusterCreator tool[4], with adjustments to match the experimental requirements. The cluster consists of a single control-plane node, three dedicated storage nodes hosting MinIO and Cassandra, and three general-purpose worker nodes. One worker node is reserved exclusively for the service under evaluation, while the remaining two are used for the FaaS workloads. Node taints [kubb] are configured to enforce this separation, as summarized in Table 7.2.

7.2.2 Service Configuration

The following infrastructure was installed on the configured cluster:

[2] https://www.proxmox.com/en/products/proxmox-virtual-environment/overview
[3] https://www.debian.org/releases/bookworm/
[4] https://github.com/christensenjairus/ClusterCreator

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Node	Taints
controlplane	`node-role.kubernetes.io/control-plane:NoSchedule`
general-0	`nodeclass=store-service:NoSchedule`
general-1	—
general-2	—
general-3	`dedicated=storage:NoSchedule`
general-4	`dedicated=storage:NoSchedule`
general-5	`dedicated=storage:NoSchedule`

Table 7.2: Node taints in the Kubernetes cluster

Knative. Knative was installed using the official Knative Operator[5] with version 1.18.1. Both `knative-serving` and `knative-eventing` components were deployed, with distributed trace exporters configured via the Zipkin[6] protocol. For Knative Serving, cold starts were disabled to avoid the latency penalty of scaling to zero during the experiments. Autoscaling parameters were tuned to support higher concurrency by setting the concurrency target to 2000 and the burst capacity to 2000, allowing the system to handle large request bursts without prematurely scaling. This was done because the default (100 concurrent requests) would lead to an explosion of instances for long-running requests in load tests (this is due to requests building up). The sampling rate for traces was adapted for each experiment using the operator configuration to ensure the number of collected traces stays within a reasonable range (fewer than a few 1000 total traces per experiment).

Minio. MinIO[7] was deployed via the `minio-operator` Helm chart from the official MinIO repository[8] in version 7.1.1. The operator was installed into the `minio-operator` namespace using the default chart configuration. A tenant was configured to span all three available storage nodes, with each node providing two volumes. This layout enables erasure coding to operate correctly while utilizing the full storage capacity of the cluster.

Cassandra. Apache Cassandra[9] was deployed using the `cass-operator` v1.24.1 Helm chart from the official K8ssandra repository[10] The operator was installed in the `cass-operator` namespace. A single Cassandra datacenter was configured to span three racks, each placed on one of the three dedicated storage nodes, ensuring data distribution and replication across all available storage resources.

[5]`https://github.com/knative/operator/`
[6]`https://zipkin.io/`
[7]`https://www.min.io/`
[8]`https://operator.min.io/`
[9]`https://cassandra.apache.org/`
[10]`https://github.com/k8ssandra/cass-operator`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

7.2.3 Operator configuration

The custom STORE-Operator is configured to integrate both MinIO and Cassandra as storage providers. Metadata and permission data are stored persistently using the `persistent` Redis[11] sidecar option. Listing 11 shows the full configuration used in the experiments, with credentials replaced by placeholders. Detailed instructions for how to install the operator CRDs are included in Appendix A.1.1.

Listing 11 STORE-Operator configuration

```yaml
apiVersion: store/v1alpha1
kind: StorageProvider
metadata:
  name: minio-existing
  namespace: default
spec:
  type: minio
  provider:
    endpoint: minio-url:port
    accessKey: key
    secretKey: key
---
apiVersion: store/v1alpha1
kind: StorageProvider
metadata:
  name: cassandra-existing
  namespace: default
spec:
  type: cassandra
  provider:
    hosts: [cass-url]
    keyspace: STORE
    username: username
    password: password
---
apiVersion: store/v1alpha1
kind: STOREService
metadata:
  name: sample-store
  namespace: default
spec:
  image: store-service:latest
  port: 8080
  serviceType: ClusterIP
  storageProviderRefs:
    - name: minio-existing
    - name: cassandra-existing
  redisSidecar:
    image: redis:7.2.9
    persistent: true
```

[11] https://redis.io/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

7.2.4 Observability Stack

Jaeger[12] v2.9.0 provides distributed tracing, deployed via the OpenTelemetry Operator[13] v0.129.1 in the `observability` namespace. Jaeger is configured to accept Zipkin-format traces and to export span metrics to Prometheus using the `spanmetrics` processor, providing an additional source of RPS measurements. This additional RPS measurement (available via the Jaeger `Monitor Tab`) is used to validate the accuracy of the load testing tool described below.

Prometheus[14] v0.83.0 is deployed using the `kube-prometheus-stack`[15] 75.11.0 Helm chart from the Prometheus Community repository, also in the `observability` namespace. The scraping frequency for CPU and memory metrics is increased to 5 seconds to allow finer-grained resource usage analysis.

Grafana[16], included as part of the Prometheus stack, is customized to enhance the *Kubernetes / Compute Resources / Pod* dashboard with higher-resolution measurements and aggregated metrics across multiple concurrent function instances.

The queries in Listings 12 and 13 show the adapted Prometheus Query Language expressions used in the customized Grafana dashboard to aggregate metrics with lower rolling average values and across multiple function instances by removing dynamic suffixes from pod names. Those queries can easily be adapted to aggregate multiple different functions of the same workflow for easier reproduction of workflow experiments.

Listing 12 Prometheus query for CPU usage in the adapted dashboard.

```
sum by (container) (
  label_replace(
    rate(container_cpu_usage_seconds_total{
      namespace= "$namespace",
      cluster= "$cluster",
      container!="",
      pod=~"$pod.*"
    }[15s]),
    "pod_prefix", "$1", "pod", "^(.*)-[a-z0-9]{9,10}-[a-z0-9]{5}$"
  )
)
```

7.2.5 Workload Generation and Data Processing

Workload generation is performed using a custom-built load testing library developed as part of this work. The library is implemented by leveraging Vert.x's capabilities of stressing a system with as many concurrent connections as the host's available ports

[12]https://www.jaegertracing.io/
[13]https://opentelemetry.io/docs/platforms/kubernetes/operator/
[14]https://prometheus.io/
[15]https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack
[16]https://grafana.com/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 13 Prometheus query for memory usage in the adapted dashboard.

```
sum by (container) (
  label_replace(
    container_memory_working_set_bytes{
      job= "kubelet",
      metrics_path="/metrics/cadvisor",
      cluster= "$cluster",
      namespace= "$namespace",
      container!="",
      image!="",
      pod=~"$pod.*"
    },
    "pod_prefix", "$1", "pod", "^(.*)-[a-z0-9]{9,10}-[a-z0-9]{5}$"
  )
)
```

permit. It exposes a declarative interface for specifying the target URL, HTTP method, headers, total number of requests, the desired request rate, and an optional request body supplier. The request rate is controlled through the `RateSchedule` abstraction, which supports multiple modes:

- **unlimited** concurrency for saturating the system,
- **sequential** mode for issuing requests one after another, and
- **ramping** mode for gradually increasing the load over time.

In addition, `RateSchedule` supports *composition*, allowing complex workloads to be constructed by chaining multiple schedules together (e.g., a ramp-up phase followed by a steady state and a ramp-down). The accuracy of the generated request rate was validated using Jaeger's RPS measurements. On the other hand, the extracted trace data was compared to end-to-end measurements of duration taken via the load testing tool. All test case definitions used in the experiments are implemented within this framework and are included in Appendix A.1.8.

Results for CPU and memory consumption are collected manually from the Grafana dashboard. For trace data, a custom script is used to fetch Jaeger traces and transform them into a simplified CSV format, preserving references to parent traces. This CSV representation is then processed by an additional script to extract the required information, such as computing deltas and sums of traces for different functions. All Python scripts for trace processing and plot generation are included in Appendix A.1.9.

7.2.6 Cluster Visualization

Figure 7.1 provides an overview of the experimental setup, showing the relationships between hardware, cluster nodes, deployed services, observability components, and the external client used for workload generation and analysis. The nodes are color-coded accord-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

ing to their Kubernetes taints to make their roles visually distinguishable: red indicates the node reserved for the STORE-Service (`nodeclass=store-service:NoSchedule`), green denotes the general-purpose nodes used for FaaS workloads (no taints), blue marks the storage nodes hosting MinIO and Cassandra (`dedicated=storage:NoSchedule`), and purple is used for the control-plane node. Arrows represent the primary communication paths between components, including service calls, monitoring data flows, and data replication communication.

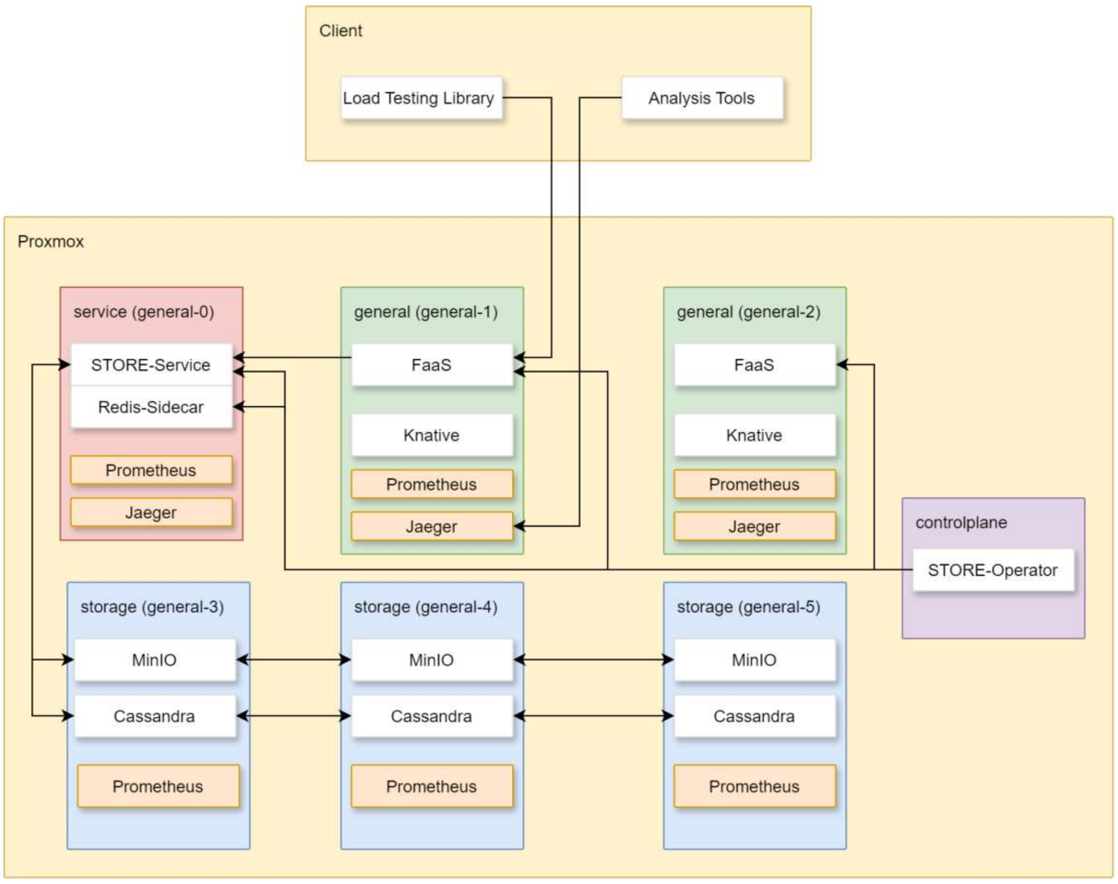


Figure 7.1: Visualization of the experiment setup

7.3 Experimental Design

This section defines the concrete experiments conducted to evaluate the proposed system. We begin with a usability assessment in which several representative FaaS use cases, including multi-function workflows, are implemented using STORE. These use cases are later used as the workload basis for subsequent experiments measuring scalability and performance, enabling a consistent comparison across evaluation dimensions. To assess backwards compatibility, we additionally perform a migration from an existing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

storage technology to STORE, demonstrating the feasibility of integrating the system into established environments without disrupting functionality.

7.3.1 Experimental Workflows

To evaluate the practicality of the proposed system, we implemented a set of representative FaaS use cases that exercise core features, including dynamic storage selection, fine-grained permissions, concurrency control, and the `map` interface. The selected use cases cover both simple data operations and more complex multi-function workflows, ensuring that different interaction patterns and system capabilities are tested. The functions in workflows are orchestrated sequentially. Table 7.3 summarizes the implemented use cases. Detailed descriptions of the use cases follow.

Use Case	#Functions	Dynamic	Map	Permissions	Concurrency
Storing Files	1			✓	
Retrieving Files	1				
Storing Tabular Data	1			✓	
Retrieving Tabular Data	1				
Sequential Workflow	3			✓	
Counter Workflow	3	✓	✓		✓
Shopping Cart Workflow	4	✓	✓	✓	✓

Table 7.3: Implemented FaaS use cases and the features they exercise.

Storing Files. This function is implemented using both the simple API call and the DSL-based interface of the `StoreClient`. It uses the `StoreClient` via CDI. The DSL variant allows explicit metadata configuration, such as setting the security level to `PUBLIC`, while both variants store the data and return either a success message with the location or an error message.

Retrieving Files. This function is implemented utilizing both a simple and a DSL-based API. It uses the `StoreClient` via CDI to retrieve a stored file by name, defaulting to `testfile` if no input is provided, and returns either a success message with the retrieved file size or an error message if the operation fails.

Storing Tabular Data. This function is implemented utilizing both a simple and a DSL-based API, using the `StoreClient` via CDI. The concrete implementation specifies custom permissions by adding additional identifiers to the allowed identity list and explicitly permits another function to access the stored data. It also demonstrates how a function can resolve and include its own identities in permission management.

Retrieving Tabular Data. This function is available in both a simple and a DSL-based variant, using the `StoreClient` via CDI. It retrieves a specified row from a table, with the ability to request individual columns, multiple named columns, or the entire set of columns.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Sequential Workflow. This workflow consists of three sequential steps, performing both file and tabular data storage and retrieval, and uses the `StoreClient` via CDI. It explores the permission levels `WORKFLOW` and `WORKFLOW_EXECUTION`, demonstrating how these affect data accessibility across workflow steps. The implementation also shows how the current workflow execution ID can be obtained. The obtained ID is then used as part of the identifiers to avoid name collisions.

Counter Workflow. This workflow uses the `map` interface, which inherently relies on dynamic storage, and is structured into three steps: first, checking a quota counter; second, simulating data processing; and third, incrementing the counter. It employs optimistic concurrency control by storing the version number of the counter and including it in subsequent updates to prevent conflicting writes.

Shopping Cart Workflow. This workflow consists of four steps: first, validating the input, which specifies an amount and an item to be added to a user's shopping cart; second, updating the shopping cart; third, computing analytics data such as sums and averages; and fourth, enriching the cart with information about the prospective delivery date. It is designed to utilize the full range of available features and to employ both types of underlying storage internally. The workflow uses the `map` interface to manage cart state and related data, with concurrency partitioned across users to avoid conflicts.

Summary

The implemented use cases collectively exercise the key features of the proposed system in realistic FaaS scenarios, ranging from simple single-function operations to multi-step workflows. They cover diverse storage patterns (file-based and tabular), access control configurations (including custom permissions, workflow-level restrictions, and execution-specific access), dynamic storage allocation, and the use of the `map` interface with concurrency control. This breadth of coverage illustrates the practicality of the system by showing how its capabilities can be applied to solve everyday serverless application tasks.

7.3.2 Usability Experiment Design

To complement the practicality evaluation, we define the following experiments evaluating the savings in LoC when comparing STORE to the state of the art using Terraform[17] or Pulumi[18] for provisioning MinIO and Cassandra. Quarkus Funqy is used for all function body implementations. The configuration, including client setup and connection details, is also included in the total LoC. Individual LoC calculations are performed according to Equation (7.2).

The total LoC sum for the state of the art is comprised of the following parts:

[17] `https://developer.hashicorp.com/terraform`
[18] `https://www.pulumi.com/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1. **Setup (Infrastructure as Code):** We measured the LoC required to provision the necessary resources:
    - Terraform: creation of a MinIO bucket or a Cassandra table with three columns.
    - Pulumi: creation of a MinIO bucket or a Cassandra table with three columns.
2. **Configuration (Client Wiring):** We measured the number of configuration lines required to connect the function code to the storage systems (MinIO and Cassandra). Only the connection setup is included (manual instantiation of the client, connection details), excluding Kubernetes deployment descriptors.
3. **Implementation (Function Bodies):** We implement function bodies for the relevant use cases using the MinIO and Cassandra clients.

Based on these measurements, the LoC per use case ($LoC_{use\text{-}case}$) are calculated according to the following formula:

$$\mathrm{LoC}_{\text{use-case}} = \mathrm{LoC}_{\text{Setup}} + \mathrm{LoC}_{\text{Configuration}} + \mathrm{LoC}_{\text{Function Bodies}} \tag{7.3}$$

where:

- $\mathrm{LoC}_{\text{Setup}}$ is counted at most once per technology (MinIO or Cassandra),
- $\mathrm{LoC}_{\text{Configuration}}$ is counted at most once per technology (MinIO or Cassandra),
- $\mathrm{LoC}_{\text{Function Bodies}}$ covers only those function bodies relevant to the given use case.

Three criteria define success:

U1 the implementation of each use case compiles without errors and runs successfully.

U2 all storage-related functionality in the implementation is realized exclusively through the STORE-SDK, without relying on external storage libraries or vendor-specific features.

U3 for each use case, the required LoC satisfy $LoC^{\text{STORE}}_{\text{use-case}} < LoC^{\text{Terraform}}_{\text{use-case}}$ and $LoC^{\text{STORE}}_{\text{use-case}} < LoC^{\text{Pulumi}}_{\text{use-case}}$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

7.3.3 Performance Experiment Design

A subset of the use cases defined in the usability evaluation is used for the performance experiments, where they are evaluated under varying data sizes. For the data size experiments, the Load Test Tool (Section 7.2.5) is used, where requests are submitted sequentially to isolate the effect of payload size on performance. Each experiment is repeated five times to reduce the influence of transient spikes and noise from the underlying system, and the reported results are calculated as the arithmetic mean across all repetitions. Each component involved in the experiment is restarted after each run.

Table 7.4 shows an overview of the experiments with their respective input ranges.

Use Case	**Data Size Range (KB)**
Storing Files	1 – 100000
Retrieving Files	1 – 100000
Storing Tabular Data	1 – 10000
Retrieving Tabular Data	1 – 10000

Table 7.4: Use cases and parameter ranges for performance experiments

For the data size experiments, the following success criteria are defined:

D1 the targeted maximum input size for each use case is processed without failures.

D2 relative overhead is constant or decreasing across the tested sizes

D3 absolute overhead grows at most linearly with input size

D4 CPU usage scales at most linearly with the data size

D5 memory usage scales at most linearly with the data size

7.3.4 Scalability Experiment Design

A subset of the use cases defined in the usability evaluation is used for the scalability experiments, where they are evaluated under varying request rates (RPS) and data sizes. For the RPS experiments, the Load Test Tool (Section 7.2.5) is used with a linear ramping profile, gradually increasing the request rate to the specified maximum. Each experiment is repeated five times to reduce the influence of transient spikes and noise from the underlying system, and the reported results are calculated as the arithmetic mean across all repetitions. Each component involved in the experiment is restarted after each run.

Table 7.5 shows an overview of the experiments with their respective input ranges.

For the RPS experiments, small payload sizes of approximately 1 KB are used to focus on request processing scalability rather than transfer overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Use Case	**RPS Range**
Storing Files	1 – 100
Retrieving Files	1 – 500
Storing Tabular Data	1 – 500
Retrieving Tabular Data	1 – 600
Shopping Cart Workflow	1 – 100

Table 7.5: Use cases and parameter ranges for scalability experiments

Success Criteria. For the RPS experiments, the following success criteria are defined:

R1 the system sustains the targeted maximum RPS for each use case without failures

R2 relative overhead is constant or decreasing for the tested RPS range

R3 absolute overhead is constant or decreasing for the tested RPS range

R4 CPU usage scales at most linearly with the request rate

R5 memory usage scales at most linearly with the request rate

7.3.5 Backwards Compatibility

We define backwards compatibility as the existence of a migration path from a Quarkus Funqy Knative Function implementation using an existing storage solution to one using STORE without requiring changes to business logic. The scope is restricted to a Kotlin Quarkus Funqy Function, as this is the STORE binding implemented for the evaluation; other runtimes can be assessed analogously after implementing their respective STORE bindings. The migration path under test consists of replacing existing storage imports with those from an adapter library. Three criteria define success:

B1 the only source code change required is the update of import statements

B2 the function builds without errors after migration

B3 the functional behavior remains identical when executed with the same inputs as before migration

The metrics recorded are binary indicators for build success and behavioral equivalence, with the latter determined by comparing outputs and error behavior of the migrated and original versions under identical test cases.

For this experiment, we use the implemented *zero-touch migration library* for the Quarkus MinIO Client[19]. This means the interface we offer is extracted from the Quarkus MinIO

[19] `https://quarkus.io/extensions/io.quarkiverse.minio/quarkus-minio/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Client, but internally maps calls to STORE calls. Additionally, we restrict the scope of the experiment to only use a subset of the 65 functions that the client exposes at the time of writing.

7.4 Experimental Results

This section reports the outcomes of the usability, performance, scalability, and backwards compatibility experiments defined in Section 7.3 and aims to summarise the results. Results are presented in line with the corresponding success criteria, using tables for binary outcomes and plots for quantitative measurements.

7.4.1 Usability Results

All use cases listed in Table 7.3 were implemented using the STORE-SDK as described in the experiment design. The implementations compiled without errors, executed successfully, and passed their respective acceptance tests. No connection setup, manual resource provisioning, or other infrastructure-specific configuration was required. The complete source code of the implementations is provided in Appendix A.1.6.

LoC experiments

All use cases were additionally implemented using Terraform and Pulumi. For Terraform, the MinIO Provider[20] worked well. It is actively maintained and easy to use. In contrast, Cassandra support required relying on a niche provider[21], due to the lack of a fully fledged implementation that also supports creating tables and columns. Instead of Terraform, the open-source solution OpenTofu[22] was used. This, however, does not make a difference in terms of LoC.

For Pulumi, a Kotlin approach was followed. Since Pulumi does not maintain Kotlin Providers (only Java), a community-based Kotlin project[23] was used. This was to make the LoC comparable to business logic code also written in Kotlin. The adaptation of the Kotlin Pulumi project only had moderate success, since the used provider libraries did not have Kotlin implementations yet. So Kotlin DSL could only be used with limitations.

The used MinIO Provider for Pulumi was again easy to use and integrate. For Cassandra, however, there does not exist a provider as of writing. To still have a usable baseline, the provisioning for the Cassandra table and columns was done using the Pulumi Command Provider[24] to send commands directly via CQL.

[20] `https://github.com/aminueza/terraform-provider-minio`
[21] `https://github.com/konradotto/terraform-provider-cassandra`
[22] `https://opentofu.org/`
[23] `https://github.com/VirtuslabRnD/pulumi-kotlin`
[24] `https://www.pulumi.com/registry/packages/command/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

The business logic was implemented in Kotlin Quarkus Funqy functions. The Quarkus MinIO Client Extension[25] was used for communication with MinIO, and the DataStax Cassandra Client Extension[26] was used to interact with Cassandra. Configuration is done via the application properties file for both storage backends.

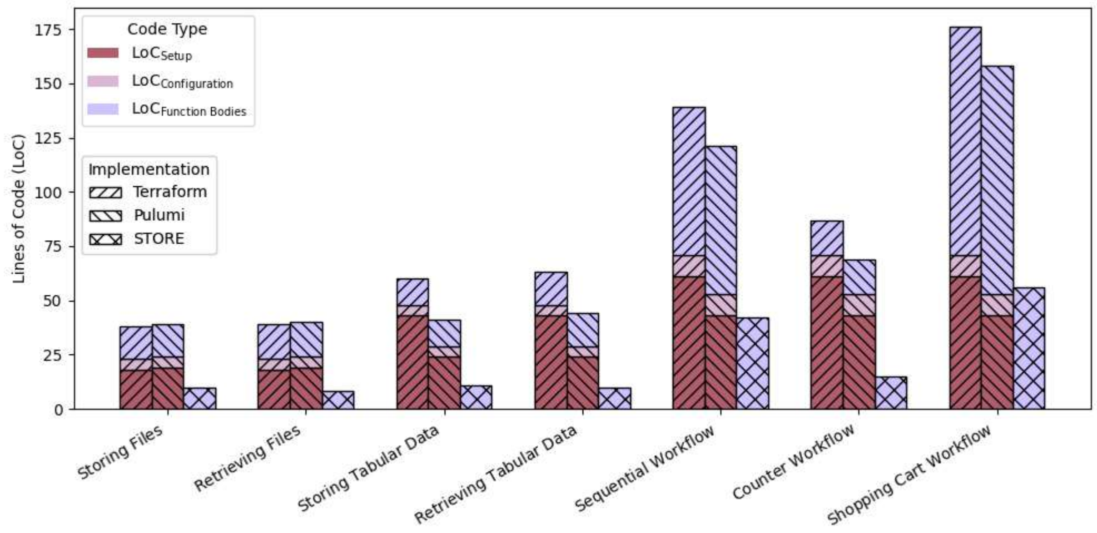


Figure 7.2: LoC per use case comparison of Terraform, Pulumi, and STORE

The concrete results of the LoC experiment (as shown in Figure 7.2) confirm the improvements of the STORE-SDK compared to Terraform and Pulumi. Across all use cases, STORE consistently requires fewer lines of code, with totals ranging from only 8 (*Retrieving Files*) to 56 (*Shopping Cart Workflow*). In contrast, Terraform ranges between 38 and 176, while Pulumi ranges between 39 and 158. The smallest difference is observed when *Storing Files*, where STORE requires 28 fewer lines than Terraform (10 vs. 38). The largest difference occurs in the *Shopping Cart Workflow*, where STORE reduces the implementation effort by 120 LoC compared to Terraform (56 vs.176). These results demonstrate that even for simple operations, STORE reduces the implementation effort considerably, and for more complex workflows, the reduction is substantial, with less than one third of the code required compared to IaC-based approaches.

Summary

The usability experiments were all successful and show significant improvements compared to the state of the art. Improvements range from 64.6% (Shopping Cart Workflow) to 84.1% (Retrieving Tabular Data) when measured as a percentual decrease in LoC.

[25]https://quarkus.io/extensions/io.quarkiverse.minio/quarkus-minio/
[26]https://quarkus.io/extensions/com.datastax.oss.quarkus/cassandra-quarkus-client/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Comparing only the length of the function bodies, the absolute reduction in lines of code ranges from 1 (*Storing Tabular Data* and *Counter Workflow*) up to 49 (*Shopping Cart Workflow*). The percentual improvements range from 6.2% (*Counter Workflow*) up to 50.0% (*Retrieving Files*).

When comparing Terraform, which is considered first generation IaC, to Pulumi, which is considered second generation IaC, the ease of use and initial setup of Terraform was perceived as better by the author. Pulumi (especially with the requirement to register with the online platform) had a non-trivial entry barrier. Compared to STORE, where this kind of manual provisioning is not needed anymore, both Terraform and Pulumi have higher complexity.

Table 7.6 summarizes the success criteria outcomes.

Use Case	**U1**	**U2**	**U3**
Storing Files	✓	✓	✓
Retrieving Files	✓	✓	✓
Storing Tabular Data	✓	✓	✓
Retrieving Tabular Data	✓	✓	✓
Sequential Workflow	✓	✓	✓
Counter Workflow	✓	✓	✓
Shopping Cart Workflow	✓	✓	✓

Table 7.6: Usability experiment results.

7.4.2 Performance Results

Below, the results of the performance experiments are presented. The data-size experiments are grouped into overhead evaluation and resource usage evaluation. Each subsection includes the corresponding figures and an evaluation of the defined success criteria.

Plot Organisation. Each experiment is presented with two rows of figures. The *first row* begins with a stacked chart of one representative test execution. From bottom to top, the stacks show: the underlying storage (blue), the service (orange), time spent in function calls (green), and the remainder (red). Lower layers are included in the overall time of all layers above them. Next to the stacked chart, the averaged *absolute* and *relative* service overheads are displayed.

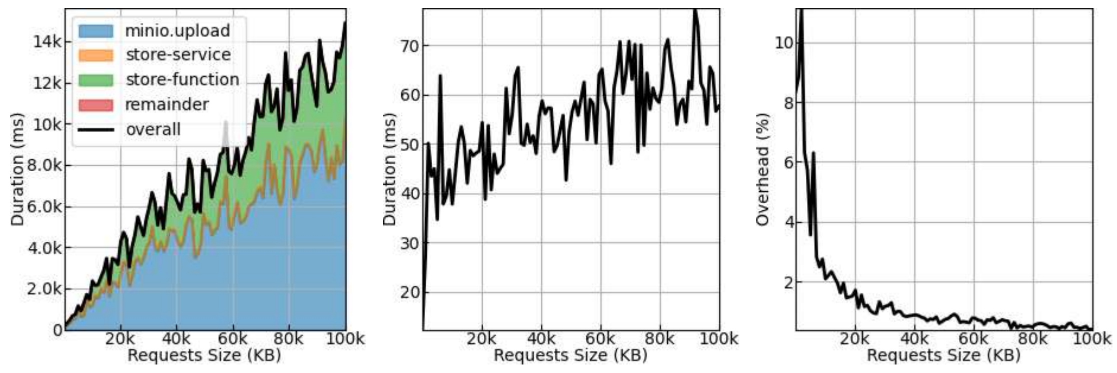
The *second row* shows CPU and memory usage over time for both the function and the service. Because each part may comprise multiple containers, the per-container series are shown alongside their sum (green). For the service component, we plot the service container (orange) and its Redis sidecar (blue). For functions, we plot the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

user container (orange) and the queue-proxy (blue). All evaluations and success-criteria checks for resource usage are performed on these summed series.

Overhead

For file-based workloads, the stacked execution views in Figures 7.3 and 7.4(a) show that the function and the MinIO dominate request time as the input size increases, while the service component contributes only a minor share. This reflects the function's need to fully read the request before processing, as well as the inherently storage-bound nature of the workload. The service utilizes request streaming and does not need to wait until the full request is received.



(a) Stacked View of one execution

(b) Absolute overhead

(c) Relative overhead

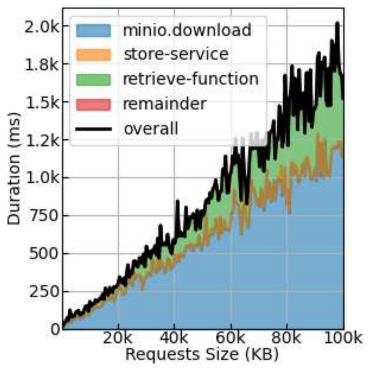
Figure 7.3: Overhead results for Storing Files

The absolute overheads in Figures 7.3 and 7.4(b) remain stable across the tested input sizes. For storing files, the coefficient of variation is 18.51% with a minimal slope of 2.37×10^{-4}, while for retrieving files the coefficient of variation is higher (60.73%) but the slope remains negligible (6.07×10^{-5}). In both cases, the normalized slopes are close to zero and not statistically significant ($p \approx 0.08$), indicating no evidence of superlinear growth and satisfying the success criterion of at most linear scaling.

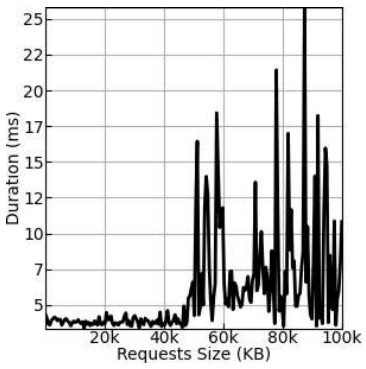
The relative overheads in Figures 7.3 and 7.4(c) decrease across the tested range. For storing files, the slope is -3.79×10^{-5} (normalized: -4.91×10^{-6}), and for retrieving files, the decrease is even more evident, with a slope of -4.11×10^{-5} (normalized: -6.79×10^{-6}). In both experiments, the overhead approaches zero for larger input sizes, clearly meeting the criterion of being constant or decreasing.

For tabular workloads, the stacked execution views in Figures 7.5 and 7.6(a) show that the service contributes a substantial portion of the total request time, comparable to the function and Cassandra. This behavior is explained by the need for the service to read the full request payload before translating it into CQL statements.

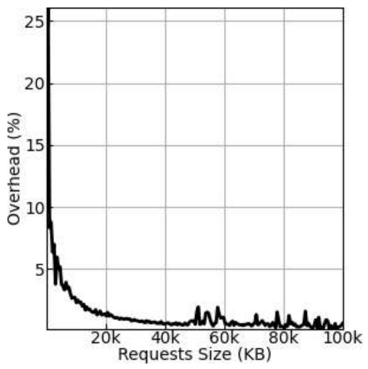
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Stacked View of one execution

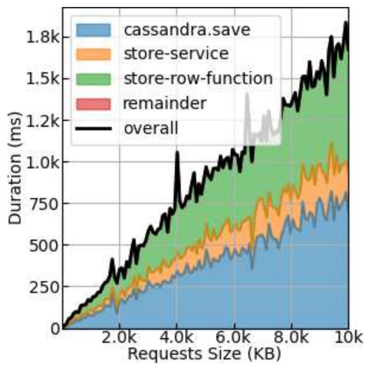


(b) Absolute overhead

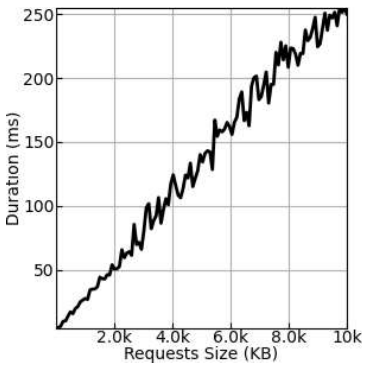


(c) Relative overhead

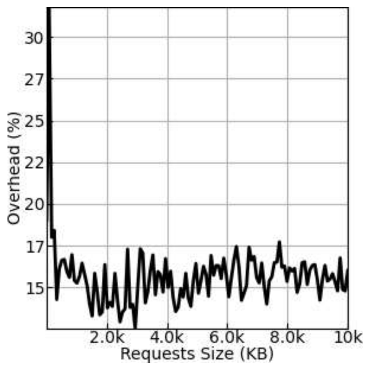
Figure 7.4: Overhead results for Retrieving Files



(a) Stacked View of one execution



(b) Absolute overhead



(c) Relative overhead

Figure 7.5: Overhead results for Storing Tabular Data

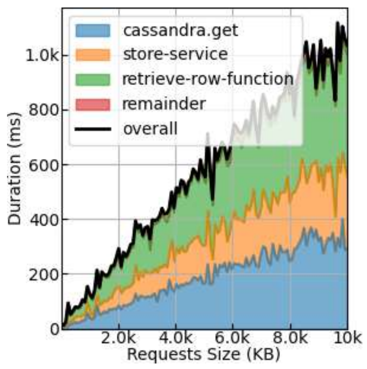
The absolute overheads in Figures 7.5 and 7.6(b) increase linearly with input size. For storing rows, variability is high (CV 57.72%) with a slope of 0.026 and very high explanatory power ($R^2 = 0.99$). For retrieving rows, a similarly strong linear trend is observed ($R^2 = 0.98$). In both cases, the normalized slopes are near zero and not statistically significant ($p > 0.05$), confirming the absence of superlinear growth.

The relative overheads in Figures 7.5 and 7.6(c) remain stable across the tested sizes. For storing rows, the CV is 12.16% with a slightly negative slope and low R^2, while for retrieving rows the CV is 8.40% with similarly low variability. Both results meet the success criterion of being constant or decreasing.

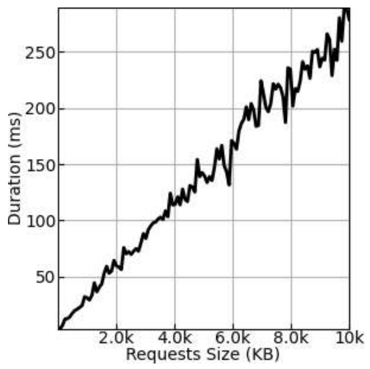
Resource Usage

Across all file- and table-based performance experiments, resource usage remains within the defined success criteria. CPU consumption is stable for both the service (Figures 7.7

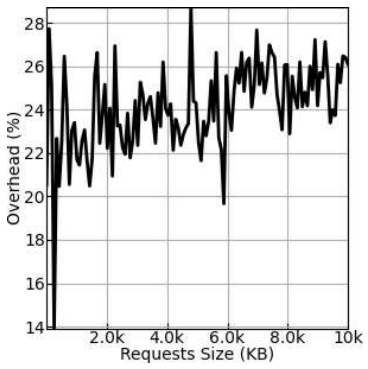
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Stacked View of one execution



(b) Absolute overhead

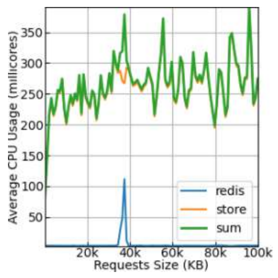


(c) Relative overhead

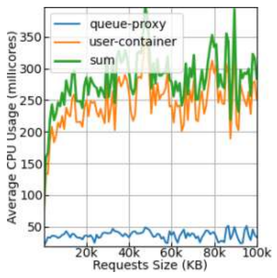
Figure 7.6: Overhead results for Retrieving Tabular Data

to 7.10a) and function components (Figures 7.7 to 7.10b), with low coefficients of variation (12–17%) and near-zero normalized slopes that are not statistically significant, confirming the absence of size-driven CPU growth.

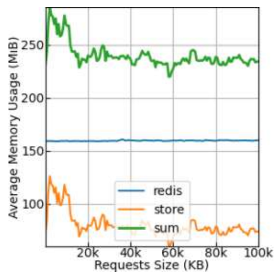
Service memory usage (Figures 7.7 to 7.10c) is consistently low and stable (CV 3–5%), satisfying the "at most linear" criterion. In contrast, function memory (Figures 7.7 to 7.10d) shows an upward trend with input size (R^2 = 0.63–0.86), consistent with buffering entire requests. However, normalized slopes are close to zero and statistically insignificant ($p > 0.05$), demonstrating that the per-MB memory cost does not grow with input size.



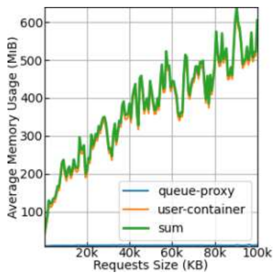
(a) Service CPU usage



(b) Function CPU Usage



(c) Service memory usage



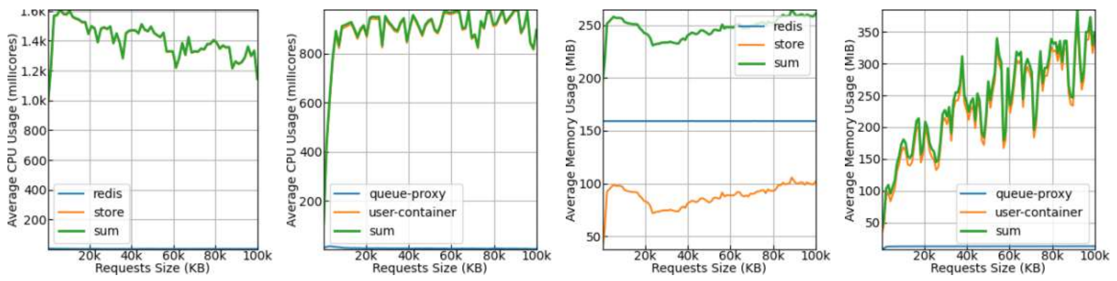
(d) Function memory Usage

Figure 7.7: Resource Usage for Storing Files

Summary

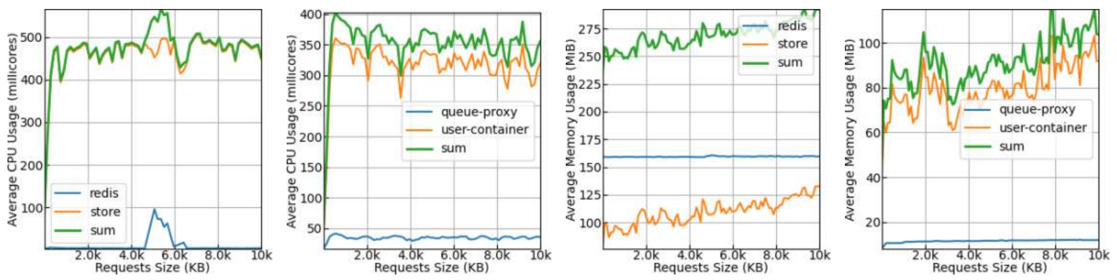
Below, the results of the performance experiments are specified as pass/fail for each of the success criteria. The success criteria are defined in Section 7.3.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



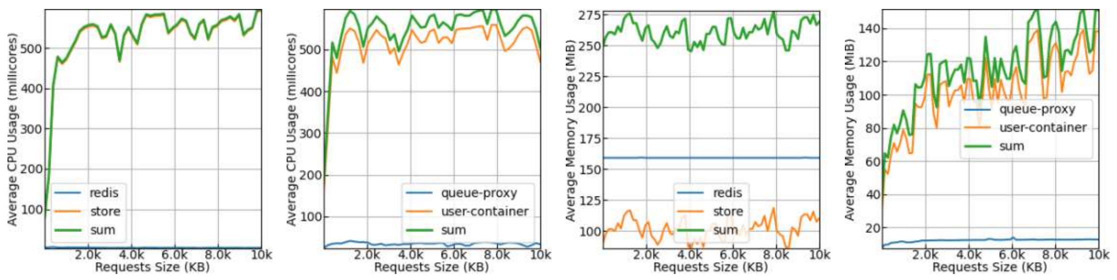
(a) Service CPU usage (b) Function CPU Usage (c) Service memory usage (d) Function memory Usage

Figure 7.8: Resource Usage for Retrieving Files



(a) Service CPU usage (b) Function CPU Usage (c) Service memory usage (d) Function memory Usage

Figure 7.9: Resource Usage for Storing Tabular Data



(a) Service CPU usage (b) Function CPU Usage (c) Service memory usage (d) Function memory Usage

Figure 7.10: Resource Usage for Retrieving Tabular Data

As we can see in Table 7.7, all data size experiments passed. The absolute overheads remain small across all use cases, ranging from as little as 3 ms (retrieving files) to a maximum of 289 ms when retrieving large tabular rows (with storing rows exhibiting similar results). Relative overheads vary more widely, with the lowest values around 0.2% for file-based use cases and the highest values of about 31% observed for storing and retrieving tabular data. These results confirm that the system scales well with input size; however, they also highlight that file-based storage benefits from request streaming, which

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

significantly reduces the relative overhead for large inputs. Accordingly, we recommend favoring file-based storage when handling large data volumes.

CPU consumption is low across all use cases, with retrieving files as the most demanding case (1609 millicores for the service and 980 millicores for the function) compared to stable ranges of 400–600 millicores in other scenarios. Memory usage, shows a minimum footprint of 36 MB (after startup) and scaling up to 640 MB in the most demanding case (store file). Observed variations are primarily explained by garbage-collection cycles.

Use Case	**D1**	**D2**	**D3**	**D4**	**D5**
Storing Files	✓	✓	✓	✓	✓
Retrieving Files	✓	✓	✓	✓	✓
Storing Tabular Data	✓	✓	✓	✓	✓
Retrieving Tabular Data	✓	✓	✓	✓	✓

Table 7.7: Data size experiments and success criteria (D1–D5)

7.4.3 Scalability Results

Below, the results of the scalability experiments are presented. The experiments are grouped into overhead evaluation and resource usage evaluation. Each subsection includes the corresponding figures and an evaluation of the defined success criteria. The plot organisation is the same as for the performance experiments (Section 7.4.2).

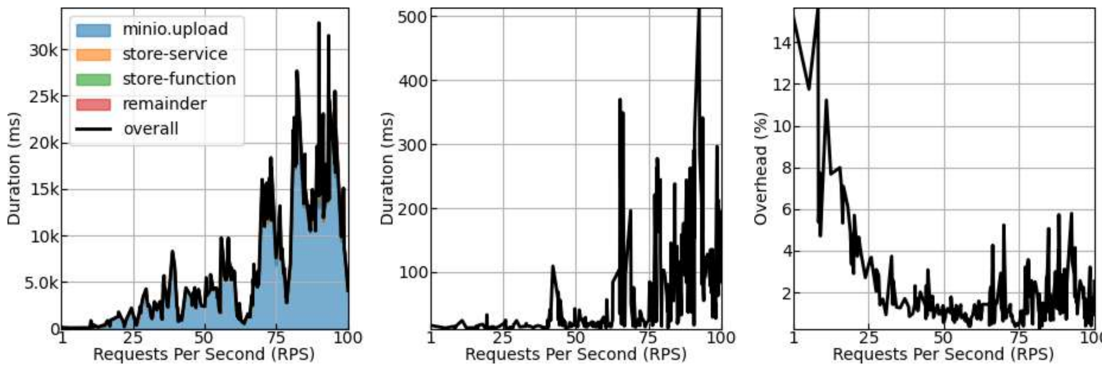
Overhead

Figure 7.11 presents the scalability results for storing files. The stacked execution view (a) shows that MinIO dominates request time, while the function and service contribute comparatively little. The absolute overhead (b) exhibits fluctuations, with a high CV of 118.57% and a relative change of 707.28%. A positive regression slope (0.33, $p < 0.05$) confirms a slight increasing trend. In contrast, the relative overhead (c) decreases across the tested range, with a negative slope (-0.0066) and a mean of 1.92% (max 15.6%). The one-sided p-value shows that the slope is not significantly greater than zero, satisfying the success criterion of being constant or decreasing.

The results for retrieving files are shown in Figure 7.12. The stacked execution view (a) indicates that request times remain very low overall, with only occasional spikes under higher load. Absolute overhead (b) is small, averaging 4.27 ms with moderate variability (CV 17.40%) and no significant growth trend (slope -1.56×10^{-4}, $p \approx 0.77$), thus fulfilling the stability requirement. Relative overhead (c) is higher in proportion to total request time, averaging 18.73% (CV 19.69%), but shows a clear decreasing trend (slope -0.0172, $p < 0.05$), meeting the success criterion of being constant or decreasing.

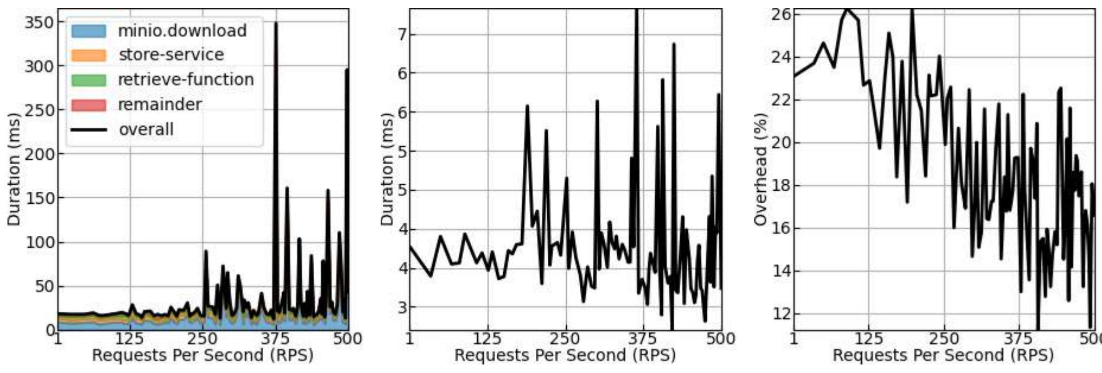
Figure 7.13 shows the scalability results for storing tabular data. The stacked execution view (a) indicates a balanced distribution of execution time across components, with overall durations remaining very low. The absolute overhead (b) is small (mean 4.63 ms)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Stacked View of one execution (b) Absolute overhead (c) Relative overhead

Figure 7.11: Overhead results for Storing Files



(a) Stacked View of one execution (b) Absolute overhead (c) Relative overhead

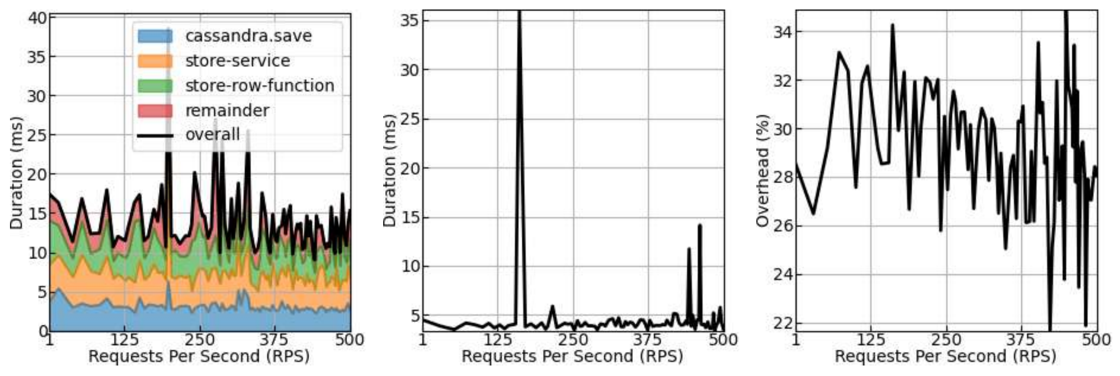
Figure 7.12: Overhead results for Retrieving Files

and exhibits no significant growth trend (slope -1.34×10^{-3}, $p = 0.62$). The relative overhead (c) is stable, averaging 29.06% (CV 8.89%), and meets the constancy criterion, with a slight decreasing slope (-5.33×10^{-3}).

The stacked execution view of Figure 7.14 (a) also shows a balanced distribution between the service, function, and Cassandra components. Execution times remain very low, which helps explain the relatively high relative-overhead values. Absolute overhead (b) is small (mean 4.15 ms, CV 43.19%) with a minimal positive slope (2.81×10^{-3}, $R^2 \approx 0.043$). Although small, this slope narrowly fails the stability criterion. Relative overhead (c) is high in percentage terms but decreases across the tested range (slope -1.38×10^{-2}, $R^2 = 0.29$), satisfying the success criterion of being constant or decreasing.

The workflow experiment (Figure 7.15) executes all operations, with execution times summed across all function components. As expected for a more complex orchestration involving Knative eventing and brokers, the stacked execution view (a) shows moderate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

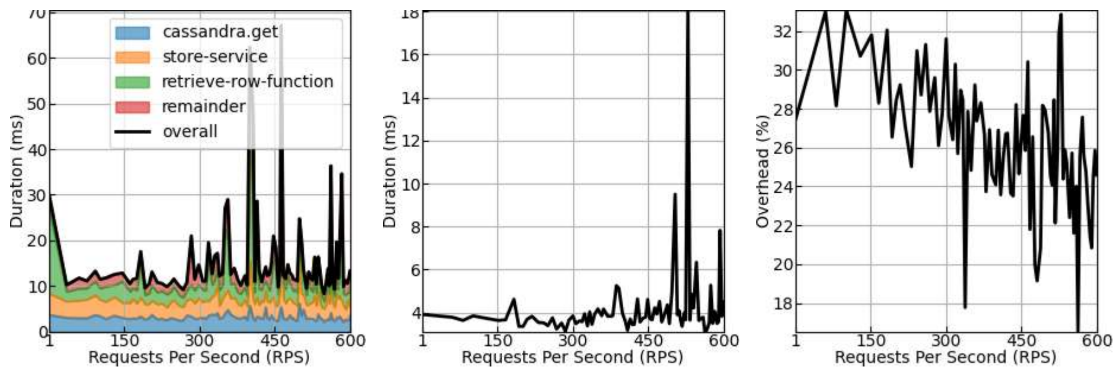


(a) Stacked View of one execution

(b) Absolute overhead

(c) Relative overhead

Figure 7.13: Overhead results for Storing Tabular Data



(a) Stacked View of one execution

(b) Absolute overhead

(c) Relative overhead

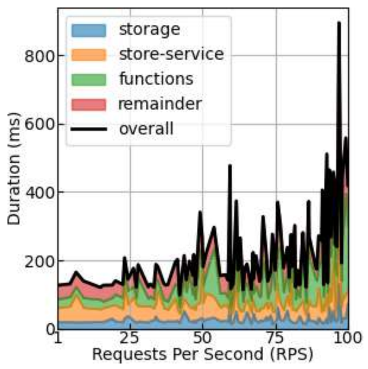
Figure 7.14: Overhead results for Retrieving Tabular Data

fluctuations, and the overhead is clearly visible. The absolute overhead (b) exhibits moderate variability (CV 30.32%) with a small but significant positive slope (0.052, $p < 0.05$). The relative overhead (c), however, decreases across the tested range (slope -0.035, $R^2 = 0.66$), satisfying the success criterion of being constant or decreasing. The positive slope in the absolute overhead is consistent with the fact that several individual workflow components already show similar trends.

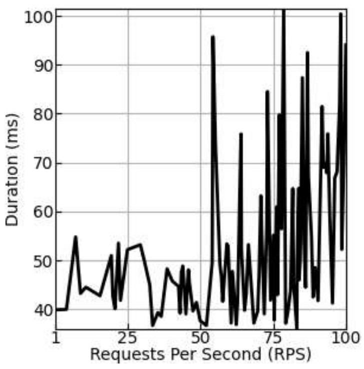
Resource Usage

Across all scalability experiments, CPU usage scales linearly with the request rate for both the service and function components. The resource plots (Figure 7.16a–b, Figure 7.17a–b, Figure 7.18a–b, Figure 7.19a–b, Figure 7.20a–b) consistently show very high R^2 values (≥ 0.96), indicating an excellent linear fit between CPU consumption and load. Furthermore, the normalized slopes are either slightly negative or near zero,

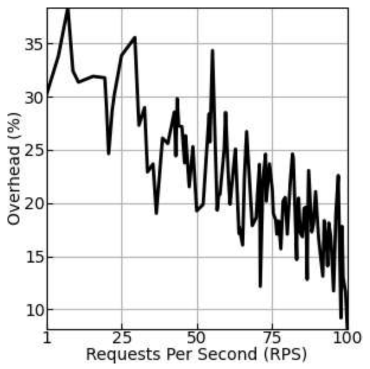
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Stacked View of one execution



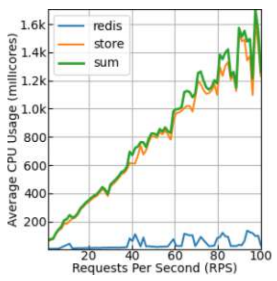
(b) Absolute overhead



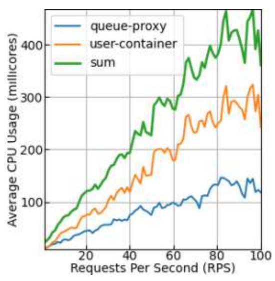
(c) Relative overhead

Figure 7.15: Overhead results for Workflow Execution

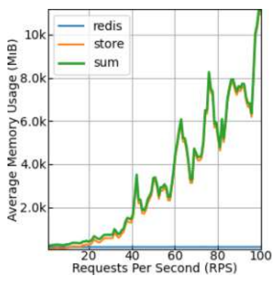
confirming that CPU usage does not grow faster than linearly with increasing request rates. These results meet the success criterion of at most linear scaling for all tested use cases.



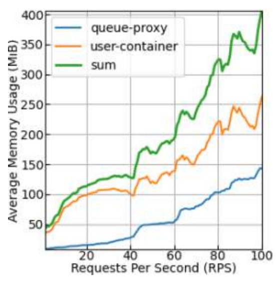
(a) Service CPU usage



(b) Function CPU Usage



(c) Service memory usage



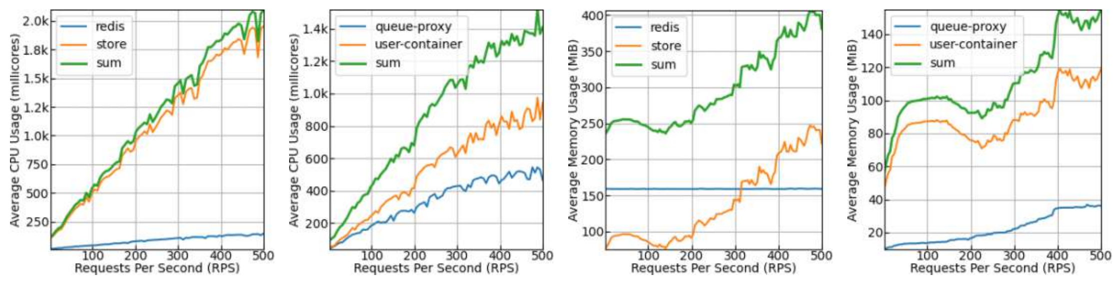
(d) Function memory Usage

Figure 7.16: Resource Usage for Storing Files

Memory usage for the store-file experiment (Figure 7.16c–d) shows that the service consumes a comparatively large amount of memory, averaging 3380 MB and peaking at 11.18 GB. This behavior can be attributed to the custom request-streaming implementation, where each pending request maintains its own buffer. The normalized slope (0.55) indicates a tendency toward slightly superlinear growth, suggesting that buffer sizing may be larger than necessary. In contrast, the function memory usage remains low, with a mean of 197 MB and a negative normalized slope (−0.080), confirming no upward trend relative to the request rate.

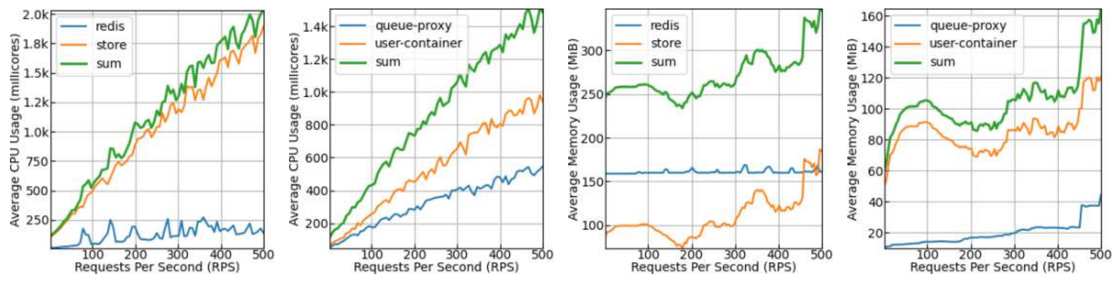
Across most scalability experiments, we observe a characteristic *initial bump* in memory usage for Quarkus-based components, visible in both the service and function containers (Figure 7.16d,Figure 7.17c–d,Figure 7.18c–d,Figure 7.19c–d,Figure 7.20d). This effect is consistent with the behavior of Quarkus Native, which allocates memory for buffers,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Service CPU usage (b) Function CPU Usage (c) Service memory usage (d) Function memory Usage

Figure 7.17: Resource Usage for Retrieving Files



(a) Service CPU usage (b) Function CPU Usage (c) Service memory usage (d) Function memory Usage

Figure 7.18: Resource Usage for Storing Tabular Data

thread structures, and runtime data after startup [qua, gra]. After this initialization phase, memory usage stabilizes and follows at most linear scaling, confirming that the observed bump is an artifact of the runtime's allocation strategy rather than workload-driven growth.

Storing files (Figure 7.16c) and workflow execution (Figure 7.20c) do not show this memory bump in the service. This is most likely due to the previously described per-request buffers overshadowing the bump.

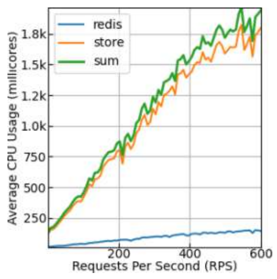
Summary

Below, the results of the scalability experiments are specified as pass/fail for each of the success criteria. The success criteria are defined in Section 7.3.4.

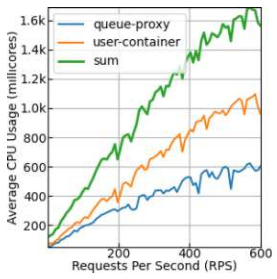
Table 7.8 shows that there are some failed success criteria for the RPS experiments. Most of those were already discussed in the detailed discussion of the experiment results. The failed criteria $R3$ can be due to the overall increased load on the cluster, causing the absolute overhead to increase marginally.

Absolute overhead remains low overall, with a minimum of 3 ms and a maximum of 514 ms (storing files). The maximum value might be an outlier, as the mean is 71 ms.

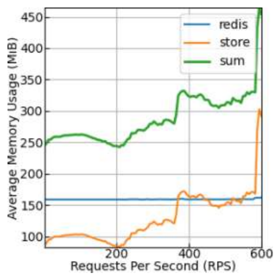
TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



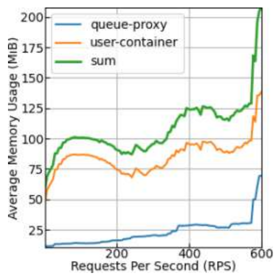
(a) Service CPU usage



(b) Function CPU Usage

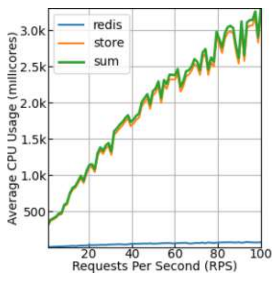


(c) Service memory usage

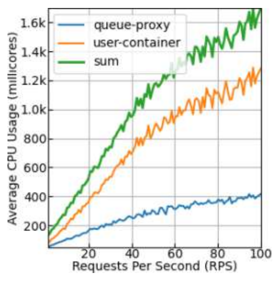


(d) Function memory Usage

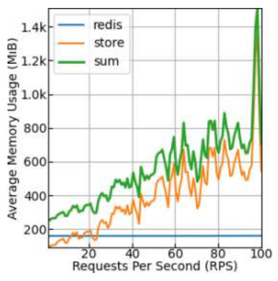
Figure 7.19: Resource Usage for Retrieving Tabular Data



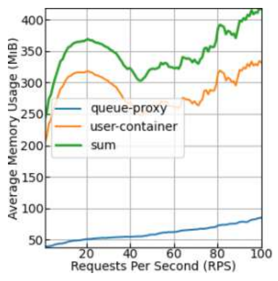
(a) Service CPU usage



(b) Function CPU Usage



(c) Service memory usage



(d) Function memory Usage

Figure 7.20: Resource Usage for Workflow Execution

The store file overhead is higher than the workflow maximum (101 ms), even though the workflow involves more components. A possible source of this discrepancy is that the workflow experiment stores ten different file keys, while the store file experiment only writes to a single key. This might increase the total duration considerably (800 ms for the workflow compared to 30 s for storing files). This difference leads to an increase in concurrent requests for the store files experiment, which in turn slows down the system.

Relative overhead ranges from as little as 0.2% (store file) to 38% (workflow), with most experiments averaging around 20%. These percentages must be interpreted in the context of very small absolute request sizes, which means that even moderate relative overhead values correspond to generally low overall latencies.

CPU consumption scales linearly across all experiments, with the workflow showing the highest demand (3306 millicores for the service and 1699 millicpu for the functions). In general, service and function CPU usage are comparable, except for the file-storage experiment (1708 millicpu vs. 467 millicpu), where the service dominates.

Memory usage again starts at (42 MB) due to using native builds. The maximum observed value is 11 GB in the file-storage case, which is attributed to the previously discussed per-connection buffer implementation issue. We consider this a trade-off to achieve request streaming and acknowledge that the implementation may require further

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

improvements. There are, however, approaches that could be tried on the problem in future work (like pooling buffers) [dzo].

Use Case	**R1**	**R2**	**R3**	**R4**	**R5**
Storing Files	✓	✓	×	✓	×
Retrieving Files	✓	✓	✓	✓	✓
Storing Tabular Data	✓	✓	✓	✓	✓
Retrieving Tabular Data	✓	✓	×	✓	✓
Shopping Cart Workflow	✓	✓	×	✓	✓

Table 7.8: RPS experiments and success criteria (R1–R5)

7.4.4 Backwards Compatibility Results

For this experiment, a Kotlin Quarkus Funqy function using the Minio Client extension is implemented. The function uses `putObject`, `getObject`, `bucketExists`, and `makeBucket` from the MinIO Client extension. In addition to that, connection details for accessing the targeted MinIO instance have to be specified in the properties of the function. As the state of the backing MinIO cannot be safely assumed, a `BucketInitializer` class runs on function startup, checks if the targeted bucket needs to be created.

Alternative solutions for ensuring bucket existence using tools like Terraform or Pulumi are considered out of scope for this scenario. Those scripts could simply be removed when migrating, since STORE handles this kind of provisioning.

For the migration procedure, the implemented drop-in replacement library is used, and the build configuration is changed from:

```
implementation("io.quarkiverse.minio:quarkus-minio:3.8.4")
```

To:

```
implementation(project(":minio-adapter"))
```

Afterwards (for example via global regex replace), the package imports are changed from:

```
import io.minio.*
```

To:

```
import com.store.*
```

The migration was completed without any further modifications to the function logic, and the application compiled and executed successfully, producing identical results to the original version. Therefore, the experiment is considered successful. The function implementations before and after the migration can be found in Appendix A.1.6.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 8

Conclusion

8.1 Summary

In this thesis, we present STORE, a novel architecture that enables Self-Provisioning storage, alleviating developers of the burden of manual storage selection, provisioning, and connection configuration. In Chapter 4 we define the self-provisioning lifecycle phases Preparation, Resolution, Permission Management, Selection, and Translation were described, along with their respective tasks and roles in self-provisioning. Next, we describe the architectural components, split between control plane components (Watcher, Storage Manager, STORE-Service Manager) and data plane components (STORE-Service, Client SDK, and Auto-Migration Framework). The architecture is designed in a way that multiple function handler frameworks and storage providers can be added easily by the conception of pluggable components.

Moreover, STORE provides dynamic storage selection based on the function data shape and access patterns via the dynamic storage selection mechanism presented in Chapter 5. The mechanism includes weighing of request streaming capabilities, translation vs. native data shape support, and provider scoring. Additionally, fallbacks are defined in case of unknown data shapes.

Zero-touch configuration, which allows self-provisioning storage without manual intervention, is presented in Chapter 5. The mechanism is spread across deployment time and runtime and also interacts with the orchestrator, serverless platform, and function components. Deployment time instrumentation is used to deliver connection and identity information into function packages and enable runtime resolution to pick the information up. Runtime resolution picks up the connection information and makes it available for the STORE-SKD and the developer. Finally, runtime data passing handles dynamically changing information by transferring it between functions in a transparent and efficient manner.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We evaluated STORE in a series of usability, performance, scalability, and backwards compatibility experiments. In the usability experiments, we compared STORE against state-of-the-art IaC frameworks and demonstrated that it reduces development effort by up to 84% for a wide range of serverless use cases. Even when comparing only the function handler code length, STORE yields improvements compared to other storage client libraries. In performance experiments, we evaluated absolute and relative service overhead via latency measurements with growing data size. STORE has shown linear scalability and relative overhead percentages that are quickly vanishing when providers supporting request streaming are employed. Resource usage of the STORE-Service and the STORE-SDK is relatively low, with startup memory usage below 100 MB RAM and linear or constant CPU usage. The scalability experiments were conducted by increasing the RPS for function and workflow executions. The results indicate linear scalability except for the use case that stores files, where memory growth exhibits slightly superlinear behavior. This is due to per-request data buffers and can be mitigated by using buffer pooling.

The main contributions of this thesis are:

- the definition of the Self-Provisioning Lifecycle and the STORE architecture
- the Dynamic Storage Selection mechanism for optimal storage provider selection
- the Zero-Touch Configuration mechanism, eliminating configuration and setup effort
- a prototype implementation of a Self-Provisioning storage

8.2 Research Questions

Looking again at the research questions defined in Chapter 1 we can detail the following answers.

RQ1 - To what extent can we extend the serverless paradigm to include storage?

This question is answered by showing the serverless design principles that are met with STORE. Chapter 4 describes the self-provisioning lifecycle and the overall architecture of STORE, laying the foundation as a scalable and performant system. Pluggable components are introduced to support multiple serverless platforms and underlying storage systems. Interfaces are introduced to define a clear and minimal contact point between the developer and the storage. The dynamic storage selection and zero-touch configuration mechanisms introduced in Chapter 5 and their implementations in Chapter 6 demonstrate the feasibility of a system where developer configuration and provisioning effort (and, by extension, all infrastructure management effort) are reduced to zero.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

The usability evaluations of STORE in Chapter 7 demonstrate that self-provisioning storage can effectively handle various real-world serverless use cases, including single function calls and workflows with both parallel and sequential steps. The LoC improvements of up to 84% indicate the STOREs interfaces are developer-friendly and more efficient than common state-of-the-art solutions. The design of a backwards compatibility strategy in Chapter 4 and the corresponding evaluation of its feasibility in Chapter 7 further show the ease of integration of STORE. The performance and scalability evaluations of STORE in Chapter 7 demonstrate that a self-provisioning storage solution can achieve low overhead and linear scalability. This conforms to the principle of elasticity. Altogether, we can conclude that a paradigmatic unification of storage and compute in serverless computing is possible to a very high extent.

RQ2 - How to allow a serverless platform to seamlessly select the optimal storage type based on data shape, and access pattern?

This question is answered by the introduction of the dynamic storage selection mechanism described in Chapter 5. The algorithm determines the storage solution based on the request streaming capabilities, the need for translation between storage systems, and reacts to a score that the storage solutions achieve. The performance experiments show that request streaming capabilities become increasingly important as data size grows. We compare a provider that supports request streaming with one that does not in Chapter 7. For the first provider, the relative overhead quickly decreases to below 1% with increasing data size, while the second exhibits constant relative overheads of around 20%.

Translation from data types that are not natively supported by a storage solution helps deepen the pool of available storage solutions, thereby making it easier to support a broad range of storage use cases. This, in combination with fallbacks for unknown or unassignable data shapes, is key to the usability of dynamic storage selection. The implementation described in Chapter 6 utilizes a Binary Large Object (BLOB) format as a fallback, and all providers support translation to this data shape. To favor providers that can natively depict a data shape and reduce the load on the service component, a translation penalty is introduced.

RQ3 - How to provide zero-touch and zero-configuration self-provisioning storage for serverless functions?

This question is answered by the introduction of the zero-touch configuration mechanism in Chapter 5 and the corresponding implementation in Chapter 6. The split between static and dynamic tasks is handled by distributing the mechanisms across the whole system and splitting it into deployment-time instrumentation, runtime resolution, and runtime data passing.

The prototype implementation performs deployment-time instrumentation by having a Kubernetes Operator interact with Knative resources and propagating information about the STORE-Service location, function, and workflow identity, and connection

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

parameters. At runtime, the STORE-Client SDK picks up the information via `Locators` and `Resolvers`. It then transparently handles authentication and connection to the STORE-Service. Passing runtime data is realized via request interception, and optimized for minimal overhead by forwarding request remainders as soon as possible.

The usage examples in Chapter 6 show that it is possible to reduce storage interaction to solely the instruction to store or retrieve a data instance or interaction with a data structure.

8.3 Future Work

The implementation of STORE, as provided in the scope of this thesis, is a step towards self-provisioning storage for the next generation of serverless computing. The implementation is open-source, integrated with a serverless platform, and usable with two alternative storage providers.

In the future, we plan to extend the dynamic storage selection to introduce a maximum number of storage locations that can be retained simultaneously, thereby reducing switching costs. Additionally, we will implement a holdout duration after which unused locations are cleared. This improvement would enable zero-cost storage switching, thereby increasing developer transparency of the system.

Regarding concurrency control, the implemented mechanism is currently purely optimistic. When utilizing function retries as the primary means for recovering from concurrency failures, an upper limit on retries cannot be guaranteed by the mechanism yet, but is often enforced by platforms or workflow definitions. Introducing transaction priority by using alternative concurrency control protocols, such as Polaris [YHCY23], can help ensure function success after a small number of retries.

Additionally, we aim to extend STORE to work in the edge-cloud continuum. In its current state, STORE is designed to work in cloud-centric environments. Extensions could include introducing the dimension of storage locality in dynamic storage selection and incorporating multiple levels of caches into the architecture.

Finally, we intend to broaden the feature set of STORE, exploring ways to integrate features such as batching, chunking, and commutative access, as well as various underlying storage paradigms, including relational databases and deterministic databases. Exploring those topics will help to refine the least common denominator interface that STORE can offer and further increase usability for real-world serverless applications.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

GitHub Repository

The GitHub repository containing the STORE prototype implementation can be found at: `https://github.com/polaris-slo-cloud/store`

A.1 Implementation Parts

A.1.1 STORE-Operator

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Operator`

A.1.2 STORE-Service

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Services`

A.1.3 STORE Api

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/api`

A.1.4 STORE Core

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/core`

A.1.5 STORE Binding

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/binding-quarkus`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

A.1.6 STORE Functions

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/functions`

A.1.7 STORE MinIO Adapter

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/minio-adapter`

A.1.8 Loadtest Tool

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/evaluation/loadtest`

A.1.9 Analysis Tools

`https://github.com/polaris-slo-cloud/store/tree/main/STORE-Functions/evaluation/analysis_tools`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Overview of Generative AI Tools Used

ChatGPT.

- Generation of LaTeX tables from raw text input.
- Generation of LaTeX figure layouts from textual layout description.
- Replacing facts in repeated text sections.

Grammarly.

- Spellchecking and rewriting sentences for clarity and consistency.

DeepL.

- Translation of acknowledgements and abstract.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

API	Application Programming Interface. 1, 3, 9, 13, 20, 29, 31, 42, 45, 87
APIs	Application Programming Interfaces. 3
BaaS	Backend as a Service. 8, 11
BLOB	Binary Large Object. 79
BLoC	Blank Lines of Code. 51
CBOR	Concise Binary Object Representation. 43
CDI	Contexts and Dependency Injection. 37, 38, 42–46, 58, 59
CQL	Cassandra Query Language. 35, 36, 63, 66
CRD	Custom Resource Definition. 32, 34
CRDs	Custom Resource Definitions. 9, 31, 33, 34, 54
CRUD	Create, Read, Update, and Delete. 38
CV	Coefficient of Variation. 51, 67, 68, 70–72
DBMS	Database Management System. 36
DDL	Data Definition Language. 36, 41
DQL	Data Query Language. 36
DSL	Domain-Specific Language. 42, 43, 46, 58, 63
FaaS	Function as a Service. 1, 2, 8, 11, 12, 52, 57–59, 87
HTTP	Hypertext Transfer Protocol. 9, 34, 35, 39–44
IaaS	Infrastructure as a Service. 7

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

IaC	Infrastructure as Code. 3, 49, 64, 65
KVS	Key-Value Store. 27
LCD	least common denominator. 22
LoC	Lines of Code. 49–51, 59, 60, 63, 64, 79, 85
NCLoC	Non-Comment Lines of Code. 51
PaaS	Platform as a Service. 7, 8
PVE	Proxmox Virtual Environment. 52
RPS	Requests per Second. 50, 55, 56, 61, 62, 74, 76, 78, 87
SaaS	Software as a Service. 7, 8
SLO	Service Level Objective. 26
SLOs	Service Level Objectives. 5
SPI	Self-Provisioning Infrastructure. 2, 11, 12, 22
TTL	Time to Live. 36, 38
URN	Uniform Resource Names. 36, 37

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[All25] Hitesh Allam. Intent-based infrastructure: Moving beyondiac to self-describing systems. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, 6(1):124–136, Jan. 2025.

[amaa] Cloud Computing Services - Amazon Web Services (AWS) — aws.amazon.com. `https://aws.amazon.com/`. [Accessed 27-03-2025].

[amab] Cloud Development Framework - AWS Cloud Development Kit - AWS — aws.amazon.com. `https://aws.amazon.com/cdk/`. [Accessed 14-09-2025].

[amac] Serverless Function, FaaS Serverless - AWS Lambda - AWS — aws.amazon.com. `https://aws.amazon.com/lambda/`. [Accessed 27-09-2025].

[Ama25] Amazon Web Services, Inc. Aws cloudformation, 2025. Accessed: 2025-09-05.

[AMI20] Juan A. Añel, Diego P. Montes, and Javier Rodeiro Iglesias. *Cloud and Serverless Computing for Scientists - A Primer*. Springer, 2020.

[apa] Apache jclouds :: Home — jclouds.apache.org. `https://jclouds.apache.org/`. [Accessed 27-03-2025].

[AWA24] Jimena Andrade-Hoz, Qi Wang, and José M. Alcaraz-Calero. Infrastructure-wide and intent-based networking dataset for 5g-and-beyond ai-driven autonomous networks. *Sensors*, 24(3):783, 2024.

[BBGK18] Anirban Bhattacharjee, Yogesh D. Barve, Aniruddha S. Gokhale, and Takayuki Kuroda. A model-driven approach to automate the deployment and management of cloud services. In Alan Sill and Josef Spillner, editors, *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018, Zurich, Switzerland, December 17-20, 2018*, pages 109–114. IEEE, 2018.

[BGCA24] Haoqiong Bian, Dongyang Geng, Yunpeng Chai, and Anastasia Ailamaki. Serverless query processing with flexible performance slas and prices. *CoRR*, abs/2409.01388, 2024.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[BHB+19] Mehdi Bezahaf, Marco Perez Hernandez, Lawrence Bardwell, Eleanor Davies, Matthew Broadbent, Daniel King, and David Hutchison. Self-generated intent-based system. In Antonio Cianfrani, Roberto Riggio, Rebecca Steinert, and Filip Idzikowski, editors, *10th International Conference on Networks of the Future, NoF 2019, Rome, Italy, October 1-3, 2019*, pages 138–140. IEEE, 2019.

[Bur01] Richard K. Burdick. Linear models in statistics. *Technometrics*, 43(2):234–236, 2001.

[CBL+20] Ryan Chard, Yadu N. Babuji, Zhuozhao Li, Tyler J. Skluzacek, Anna Woodard, Ben Blaiszik, Ian T. Foster, and Kyle Chard. funcx: A federated function serving fabric for science. In Manish Parashar, Vladimir Vlassov, David E. Irwin, and Kathryn M. Mohror, editors, *HPDC '20: The 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, June 23-26, 2020*, pages 65–76. ACM, 2020.

[CMB25] John B Carlin and Margarita Moreno-Betancur. On the uses and abuses of regression models: A call for reform of statistical practice and teaching. *Stat. Med.*, 44(13-14):e10244, June 2025.

[cnca] CNCF Landscape — landscape.cncf.io. `https://landscape.cncf.io/`. [Accessed 15-09-2025].

[cncb] CNCF Operator White Paper — tag-app-delivery.cncf.io. `https://tag-app-delivery.cncf.io/whitepapers/operator/`. [Accessed 27-09-2025].

[dar] Darklang — darklang.com. `https://darklang.com/`. [Accessed 27-03-2025].

[dzo] Principles to Handle Thousands of Connections in Java Using Netty — dzone.com. `https://dzone.com/articles/thousands-of-socket-connections-in-java-practical`. [Accessed 28-08-2025].

[EGG+22] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[Fou] The Apache Software Foundation. Apache Libcloud is a standard Python library that abstracts away differences among multiple cloud provider APIs

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

— libcloud.apache.org. https://libcloud.apache.org/. [Accessed 27-03-2025].

[FPO+24] João Frois, Lucas Padrão, Johnatan Oliveira, Laerte Xavier, and Cleiton Silva Tavares. Terraform and AWS CDK: A comparative analysis of infrastructure management tools. In *Proceedings of the 38th Brazilian Symposium on Software Engineering, SBES 2024, Curitiba, Brazil, September 30 - October 4, 2024*, pages 623–629, 2024.

[git] GitHub - cloudevents/spec: CloudEvents Specification — github.com. https://github.com/cloudevents/spec. [Accessed 27-08-2025].

[gooa] Cloud Computing, Hosting Services, and APIs — cloud.google.com. https://cloud.google.com/gcp. [Accessed 27-03-2025].

[goob] src/java/org/eclipse/jetty/util/MultiPartInputStream.java - platform/external/jetty - Git at Google — android.googlesource.com. https://android.googlesource.com/platform/external/jetty/+/refs/heads/marshmallow-release/src/java/org/eclipse/jetty/util/MultiPartInputStream.java. [Accessed 26-08-2025].

[gra] Memory Management — graalvm.org. https://www.graalvm.org/latest/reference-manual/native-image/optimizations-and-performance/MemoryManagement/. [Accessed 15-08-2025].

[hel] Helm — helm.sh. https://helm.sh/. [Accessed 27-09-2025].

[JAB25] Mohammad Amin Ghasvari Jahrmoi, Mehrdad Ashtiani, and Fatemeh Bakhshi. Faasflows: an approach for reducing vendor lock-in and response time in serverless workflows. *J. Supercomput.*, 81(1):206, 2025.

[JSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[KGT+23] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C. Mohan. Firestore: The nosql serverless database for the application developer. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 3376–3388. IEEE, 2023.

[kuba] Production-Grade Container Orchestration — kubernetes.io. https://kubernetes.io/. [Accessed 27-09-2025].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[kubb] Taints and Tolerations — kubernetes.io. `https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/`. [Accessed 11-08-2025].

[LGNR24] Thomas Larcher, Philipp Gritsch, Stefan Nastic, and Sashko Ristov. Baasless: Backend-as-a-service (baas)-enabled workflows in federated serverless infrastructures. *IEEE Trans. Cloud Comput.*, 12(4):1088–1102, 2024.

[MA24] Ravi Magham and Research Advisor. Cloud-native distributed databases: A comprehensive overview. *INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY AND MANAGEMENT INFORMATION SYSTEMS*, 15:60–74, 12 2024.

[Mas] Larry M Masinter. RFC 7578: Returning Values from Forms: multipart/form-data — datatracker.ietf.org. `https://datatracker.ietf.org/doc/html/rfc7578`. [Accessed 26-08-2025].

[MCCL23] Di Mo, Robert Cordingly, Donald Chinn, and Wes Lloyd. Addressing serverless computing vendor lock-in through cloud service abstraction. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2023, Naples, Italy, December 4-6, 2023*, pages 193–199. IEEE, 2023.

[Min25] MinIO, Inc. Minio: High-performance, s3-compatible, kubernetes-native object storage. `https://www.min.io/`, 2025. Accessed: 2025-09-05.

[MK19] Andreas Meier and Michael Kaufmann. *NoSQL Databases*, pages 201–218. Springer Fachmedien Wiesbaden, Wiesbaden, 2019.

[moz] 100 Continue - HTTP | MDN — developer.mozilla.org. `https://developer.mozilla.org/de/docs/Web/HTTP/Reference/Status/100`. [Accessed 26-08-2025].

[MSKV19] Somnath Mazumdar, Daniel Seybold, Kyriakos Kritikos, and Yiannis Verginadis. A survey on data storage and placement methodologies for cloud-big data ecosystem. *J. Big Data*, 6:15, 2019.

[Nas24] Stefan Nastic. Self-provisioning infrastructures for the next generation serverless computing. *SN Comput. Sci.*, 5(6):678, 2024.

[NRF$^+$22] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. A serverless computing fabric for edge & cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–12, 2022.

[OLB16] Flávio Oquendo, Jair C. Leite, and Thaís Batista. Executing software architecture descriptions with sysadl. In Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar, editors, *Software Architecture - 10th European*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings, volume 9839 of *Lecture Notes in Computer Science*, pages 129–137, 2016.

[ope] Open Source Cloud Computing Infrastructure - OpenStack — openstack.org. `https://www.openstack.org/`. [Accessed 27-09-2025].

[ore] src/com/oreilly/servlet/multipart/MultipartParser.java · master · examples / Java Servlet Programming · GitLab — resources.oreilly.com. `https://resources.oreilly.com/examples/9781565923911/-/blob/master/src/com/oreilly/servlet/multipart/MultipartParser.java?ref_type=heads`. [Accessed 26-08-2025].

[Pro24] Knative Project. Knative - kubernetes-based platform to build, deploy, and manage modern serverless workloads, 2024. Accessed: 2024-11-11.

[pul] Pulumi - Infrastructure as Code, Secrets Management, and AI — pulumi.com. `https://www.pulumi.com/`. [Accessed 01-04-2025].

[PVS19] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 193–206. USENIX Association, 2019.

[qua] Analysing Quarkus Native startup RSS consumption — quarkus.io. `https://quarkus.io/blog/native-startup-rss-troubleshooting/`. [Accessed 15-08-2025].

[QZ20] Chen Qian and Wenjing Zhu. F(X)-MAN: an algebraic and hierarchical composition model for function-as-a-service. In Raúl García-Castro, editor, *The 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, KSIR Virtual Conference Center, USA, July 9-19, 2020*, pages 210–215. KSI Research Inc., 2020.

[Rep] Replicated. Kustomize - Kubernetes native configuration management — kustomize.io. `https://kustomize.io/`. [Accessed 27-09-2025].

[rfca] RFC 2141: URN Syntax — rfc-editor.org. `https://www.rfc-editor.org/rfc/rfc2141.html`. [Accessed 25-08-2025].

[rfcb] RFC 7049: Concise Binary Object Representation (CBOR) — rfc-editor.org. `https://www.rfc-editor.org/rfc/rfc7049.html`. [Accessed 27-08-2025].

[RHG$^+$24] Sashko Ristov, Mika Hautz, Philipp Gritsch, Stefan Nastic, Radu Prodan, and Michael Felderer. Storeless: Serverless workflow scheduling with federated

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

storage in sky computing. In Walid Gaaloul, Michael Sheng, Qi Yu, and Sami Yangui, editors, *Service-Oriented Computing - 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3-6, 2024, Proceedings, Part II*, volume 15405 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2024.

[RL19] Moo-Ryong Ra and Hee Won Lee. Ioarbiter: Dynamic provisioning of backend block storage in the cloud. *CoRR*, abs/1904.09984, 2019.

[SAA+19] Fatima Samea, Farooque Azam, Muhammad Waseem Anwar, Mehreen Khan, and Muhammad Rashid. A UML profile for multi-cloud service configuration (UMLPMSC) in event-driven serverless applications. In *Proceedings of the 8th International Conference on Software and Computer Applications, ICSCA '19, Penang, Malaysia, February 19-21, 2019*, pages 431–435. ACM, 2019.

[SBR+22] Andrea Sabbioni, Armir Bujari, Stefano Romeo, Luca Foschini, and Antonio Corradi. An architectural approach for heterogeneous data access in serverless platforms. In *IEEE Global Communications Conference, GLOBECOM 2022, Rio de Janeiro, Brazil, December 4-8, 2022*, pages 129–134. IEEE, 2022.

[SGR15] Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. Towards automated polyglot persistence. In Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 73–82. GI, 2015.

[SIA17] Julio Sandobalin, Emilio Insfrán, and Silvia Abrahão. End-to-end automation in cloud infrastructure provisioning. In Nearchos Paspallis, Marios Raspopoulos, Chris Barry, Michael Lang, Henry Linger, and Christoph Schneider, editors, *Information Systems Development: Advances in Methods, Tools and Management - Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, Larnaca, Cyprus, University of Central Lancashire Cyprus, September 6-8, 2017.* Association for Information Systems, 2017.

[SIA19] Julio Sandobalin, Emilio Insfrán, and Silvia Abrahão. ARGON: A model-driven infrastructure provisioning tool. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 738–742. IEEE, 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[SKKC11] Won Shin, Tae Wan Kim, Doo-Hyun Kim, and Chun-Hyon Chang. Parametric software metric. In Jasni Mohamad Zain, Wan Maseri Binti Wan Mohd, and Eyas El-Qawasmeh, editors, *Software Engineering and Computer Systems - Second International Conference, ICSECS 2011, Kuantan, Pahang, Malaysia, June 27-29, 2011, Proceedings, Part III*, volume 181 of *Communications in Computer and Information Science*, pages 266–273. Springer, 2011.

[SKkP$^+$25] Jeff Swenson, Andy Kimball, Raphael 'kena' Poss, Rebecca Taft, Jay Lim, Adam Storm, Sumeer Bhola, Paul Bulkley-Logston, Pj Tatlow, Rachael Harding, Rafi Shamim, Aditya Maru, and Irfan Sharif. Cockroachdb serverless: Sub-second scaling from zero with multi-region cluster virtualization. In Volker Markl, Joseph M. Hellerstein, and Azza Abouzied, editors, *Companion of the 2025 International Conference on Management of Data, SIGMOD-/PODS 2025, Berlin, Germany, June 22-27, 2025*, pages 648–661. ACM, 2025.

[SKM22] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Comput. Surv.*, 54(11s):239:1–239:32, 2022.

[Ste11] Czeslaw Stepniak. Coefficient of variation. In Miodrag Lovric, editor, *International Encyclopedia of Statistical Science*, page 267. Springer, 2011.

[Sto24] Ion Stoica. *Sky Computing: Opportunities and Challenges*, pages 15–27. Springer Nature Switzerland, Cham, 2024.

[swa] OpenAPI Specification - Version 3.1.0 | Swagger — swagger.io. `https://swagger.io/specification/`. [Accessed 27-08-2025].

[TDW$^+$12] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.

[ter] Terraform by HashiCorp — terraform.io. `https://www.terraform.io/`. [Accessed 27-03-2025].

[The25] The Apache Software Foundation. Apache cassandra: Open-source nosql distributed database. `https://cassandra.apache.org/`, 2025. Accessed: 2025-09-05.

[TJI$^+$25] Adel Nadjaran Toosi, Bahman Javadi, Alexandru Iosup, Evgenia Smirni, and Schahram Dustdar. Serverless computing for next-generation application development. *Future Gener. Comput. Syst.*, 164:107573, 2025.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[VRVV24] Camilo Vargas-Romero and Jeisson Vergara-Vargas. A cloud-agnostic infrastructure provisioning approach for scalable microservices architectures. In Néstor Darío Duque-Méndez, Luz Ángela Aristizábal-Quintero, Mauricio Orozco-Alzate, and Jose Aguilar, editors, *Advances in Computing*, pages 422–430, Cham, 2024. Springer Nature Switzerland.

[WBF$^+$20] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Softw.-Intensive Cyber Phys. Syst.*, 35(1-2):63–75, 2020.

[win] Wing Programming Language for the cloud | Wing — winglang.io. `https://www.winglang.io/`. [Accessed 27-03-2025].

[WSAP17] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In Dilma Da Silva and Bryan Ford, editors, *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 487–498. USENIX Association, 2017.

[YBKL20] Vladimir Yussupov, Uwe Breitenbücher, Ayhan Kaplan, and Frank Leymann. SEAPORT: assessing the portability of serverless applications. In Donald Ferguson, Markus Helfert, and Claus Pahl, editors, *Proceedings of the 10th International Conference on Cloud Computing and Services Science, CLOSER 2020, Prague, Czech Republic, May 7-9, 2020*, pages 456–467. SCITEPRESS, 2020.

[YHCY23] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. Polaris: Enabling transaction priority in optimistic concurrency control. *Proc. ACM Manag. Data*, 1(1):44:1–44:24, 2023.

[YYR21] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Trans. Storage*, 17(3):17:1–17:35, 2021.

[ZWM$^+$23] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Vasily Tarasov, Dimitrios Skourtis, Feng Yan, and Yue Cheng. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, 2023.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.