

Thesis Dissertation

**AUTOMATICALLY CONSTRUCTING SERVERLESS
MICROSERVICES**

Ioannis Antoniou

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Automatically Constructing Serverless Microservices

Ioannis Antoniou

Supervisor
Dr. Haris Volos

Thesis submitted in partial fulfilment of the requirements for the award of
degree of Bachelor in Computer Science at University of Cyprus

May 2025

Acknowledgements

I would like to express my honest gratitude to my supervisor Dr. Haris Volos for his continuous guidance and mentorship throughout the entirety of this thesis research. Dr. Volos was always available for the researches' needs and allocated more than enough time for discussion, problem solution and insight. Without his support and advice, this work would not have been possible.

I am equally grateful to my university providing the opportunity to work with CloudLab by granting me access to its infrastructure. This privilege was significant for providing a stable and adaptable environment for development and testing contributing to the outcomes of my research. It also allowed for an error free deployment and interaction with the Apache OpenWhisk serverless platform, playing an integral role in completing this thesis.

Lastly, I would like to give out my thanks to my family and close friends for their unwavering support and confidence they had in me throughout this process. Without the unconditional love and emotional comfort, they provided, especially in the most challenging phases of this work, I would not be able to have the outcome I achieved.

Abstract

As cloud-native systems increasingly shift toward event-driven architectures, the transition from traditional microservices to serverless computing presents a promising path for improving scalability, resource efficiency, and operational simplicity. This thesis investigates the process of converting microservices into serverless functions using Apache OpenWhisk, with a particular focus on enabling this transformation through automation. Using the microservices from the DeathStarBench benchmark suite, most notably the Hotel Reservation application, as a foundation, we first develop and apply a robust methodology for manually converting complex Go-based services into OpenWhisk-compatible actions. This process includes restructuring logic for stateless execution, handling data dependencies through external systems and adapting service interfaces to OpenWhisk’s input-output conventions.

The methodology is validated through the successful transformation and reintegration of key services such as User and Reservation, followed by empirical benchmarking using high-intensity workloads. These benchmarks reveal that while cold starts pose significant latency challenges, warm serverless actions often outperform microservice implementations in high-percentile response times, indicating their viability in performance-critical systems.

Building on these findings, the thesis presents a CLI-based tool powered by a large language model, capable of automating the conversion process. The tool leverages the semantic capabilities of generative AI to produce OpenWhisk-compatible actions and deployment artifacts. Comparative analysis between AI-generated and manually written versions of the `makeReservation` service demonstrates high structural and functional fidelity, validating the feasibility of LLM-assisted automation. The results of this work suggest that with appropriate prompt engineering and input structuring, generative AI can meaningfully reduce the manual burden of service transformation while preserving correctness and performance, offering a scalable path toward hybrid and fully serverless cloud-native architectures.

Contents

INTRODUCTION	7
1.1 Background	7
1.1.1 Current State of Cloud Computing	7
1.1.2 Microservice Architecture	7
1.1.3 Serverless Computing.....	8
1.2 Research Gaps in Microservice to Serverless Conversion	9
1.3 DeathStarBench Microservice Suite.....	9
1.4 OpenWhisk Serverless Platform.....	10
1.5 Objective	10
1.6 Research Questions	11
1.6.1 Successful Conversion and Performance Overview	11
1.6.2 Methodology Extraction	11
1.6.3 Automation of Conversion	11
RELATED WORK.....	13
2.1 Existing Work for Microservice Serverless Migration.....	13
2.1.1 Boxer Framework	13
2.1.2 Boxer Utilization and Extension.....	14
2.1.3 SigmaOS.....	14
2.1.4 Performance Enhancements in Serverless	15
2.1.5 μ 2sls	15
2.2 Choice of Serverless Platform: OpenWhisk	15
2.2.1 OpenFaaS	16
2.2.2 Fission	16
2.2.3 Knative	17
2.2.4 Apache OpenWhisk.....	17
OPENWHISK DEPLOYMENT.....	20
3.1 OpenWhisk Deployment	20
3.1.1 OpenWhisk Devtools - MacOS	21
3.1.2 OpenWhisk Devtools – Windows Subsystem for Linux: Ubuntu	22
3.1.3 OpenWhisk Documentation Instructions for Local Deployment - MacOS	22

3.1.4	OpenWhisk Documentation Instructions for Local Deployment – WSL Ubuntu	24
3.1.5	OpenWhisk CloudLab Deployment	25
DEATHSTARBENCH MICROSERVICE DEPLOYMENT		29
4.1	Choice of DeathStarBench Service	29
4.1.1	Social Network Application	30
4.1.2	Hotel Reservation Application	30
4.2	Deployment Experimentation.....	31
4.2.1	Hotel Reservation Helm-Chart Deployment.....	31
4.2.2	Social Network Helm-Chart Deployment.....	33
4.3	Selected Service: Hotel Reservation	34
HOTEL RESERVATION MICROSERVICE APPLICATION OVERVIEW.....		35
5.1	Overview	35
5.1.1	Breakdown of Core Services	36
5.1.2	Communication Protocols and Runtime Behavior	36
5.1.3	Service Code Structure and Components	37
5.1.4	Relevance and Observability	39
5.2	User Service Breakdown	39
5.2.1	User Service Structure	40
OPENWHISK GO RUNTIME.....		45
6.1	OpenWhisk Go Runtime	45
6.2	Example of Action provided by OpenWhisk Go Runtime Repository.....	47
METHODOLOGY OF CONVERSION.....		50
7.1	Identifying and Isolating the Target Service Logic	50
7.2	Ensuring Statelessness.....	51
7.3	Refactoring into the OpenWhisk Runtime Model	51
7.4	Managing External Dependencies	52
7.5	Deployment.....	52
CONVERSION OF USER SERVICE		54

8.1	User Service Conversion Attempt	54
8.1.1	Core Function Signature	57
8.1.2	Database Logic	57
8.1.3	Authentication Logic	57
8.1.4	Deployment Attempt	58
INTEGRATION OF ACTION WITH MICROSERVICE APPLICATION		60
9.1	GPRC Invocation of Original User Service	61
9.2	The checkUserHTTP function.....	62
9.3	Building new Image of the Application	65
CONVERSION OF RESERVATION SERVICE.....		68
10.1	Reservation Service Conversion.....	68
10.2	Reservation Service Integration	73
10.3	Reservation Service Invocation	73
AUTOMATION OF CONVERSION PROCESS		75
11.1	Introduction	75
11.2	Initial Attempts.....	75
11.3	Adopting Generative AI for Code Conversion.....	76
11.4	Tool Architecture and Workflow	76
11.5	Tool Architecture and Workflow	77
11.6	Observations.....	79
11.7	Closing Remarks	79
OPERATIONAL INTEGRITY AND PERFORMANCE EVALUATIONS.....		80
12.1	Benchmarking	80
12.2	Evaluation of Operational Integrity	81
12.3	Performance Evaluation	83
12.3.1	Reservation Service	84
12.3.2	User Service.....	87

12.3.3	Mixed Service Workload	90
12.3.4	Automatically Converted Reservation Service	93
12.4	Conclusion.....	95
CONCLUSIONS.....		97
13.1	Conclusion.....	97
13.2	Future Work	98

Chapter 1

Introduction

1.1	Background	7
1.1.1	Current State of Cloud Computing	7
1.1.2	Microservice Architecture	7
1.1.3	Serverless Computing.....	8
1.2	Research Gaps in Microservice to Serverless Conversion	9
1.3	DeathStarBench Microservice Suite.....	9
1.4	OpenWhisk Serverless Platform.....	10
1.5	Objective	10
1.6	Research Questions	11
1.6.1	Successful Conversion and Performance Overview	11
1.6.2	Methodology Extraction	11
1.6.3	Automation of Conversion	11

1.1 Background

1.1.1 Current State of Cloud Computing

The rapid evolution of cloud computing paradigms has transformed the architectural landscape of modern applications. From monolithic systems to distributed microservices and, more recently, to serverless models, these paradigms reflect a continuous effort to maximize flexibility, scalability, and operational efficiency. Each architectural shift has addressed particular challenges, but also introduced new complexities and trade-offs.

1.1.2 Microservice Architecture

Microservices architecture decomposes applications into small, autonomous services, each responsible for a distinct piece of business logic. These services communicate over lightweight

protocols such as HTTP, gRPC [20], or Thrift and are independently deployable, allowing for isolated development, scaling, and failure containment. The microservices model solves problems of agility and modularization inherent in monolithic applications, offering teams the ability to deploy new features rapidly and scale individual parts of an application selectively.

However, microservices introduce non-trivial operational complexity. Each service must be independently managed, requiring substantial effort in infrastructure provisioning, orchestration, monitoring, scaling policies, and fault recovery. Systems such as Kubernetes emerged precisely to address this complexity, but at the cost of requiring additional layers of management expertise.

1.1.3 Serverless Computing

In contrast, serverless computing (or Function-as-a-Service, FaaS) represents an even further abstraction. In serverless architectures, developers deploy discrete functions that are triggered by events and automatically scaled by the platform provider. Serverless models eliminate the need for server provisioning and management, promoting a pure consumption-based billing model. Functions scale to zero when idle and burst massively when necessary, thus offering exceptional elasticity.

Despite their promise, serverless platforms also impose constraints: statelessness, execution time limits, resource capping, and event-driven invocation are design assumptions that developers must embrace. Applications not originally designed with these assumptions must be adapted or re-architected.

Serverless architectures typically offer cost advantages over traditional microservice deployments due to their pay-per-use billing model and dynamic resource allocation. In serverless systems, functions are executed only in response to specific events and are billed solely for the compute time consumed during execution. In contrast, microservices often run continuously on containers or virtual machines, accumulating costs regardless of utilization. Additionally, serverless platforms eliminate the need for maintaining idle resources or over-provisioning to handle peak loads. These factors collectively reduce infrastructure expenses, making serverless a more appealing choice for workloads with variable traffic patterns.

1.2 Research Gaps in Microservice to Serverless Conversion

Despite the rapid proliferation of both microservices and serverless computing paradigms in recent years, there remains a noticeable lack of systematic research addressing the conversion of existing microservices into serverless functions, particularly regarding the automation of this transformation. While the independent advantages and challenges of microservices and serverless architectures have been extensively studied, little attention has been given to the methodologies required to migrate complex, interconnected, and often stateful microservices into stateless, event-driven serverless models. Comprehensive frameworks or automated toolchains capable of parsing service logic, externalizing state, refactoring communication patterns, and deploying serverless-ready artifacts without extensive developer reengineering are still largely absent. This gap signifies a critical research opportunity: enabling the practical and scalable adoption of serverless benefits for a vast body of existing cloud-native applications without necessitating costly and error-prone manual rewrite.

1.3 DeathStarBench Microservice Suite

The DeathStarBench [17] suite represents one of the most comprehensive and widely used microservice benchmark collections developed for evaluating the performance, scalability, and complexity of modern cloud-native systems. Developed by the Systems and AI Lab (SAIL) at Cornell University, DeathStarBench models real-world microservices applications in a way that spans the entire system stack, from frontend APIs to backend databases. The suite provides end-to-end applications that simulate production-grade workloads and service dependencies, thus allowing researchers and practitioners to study microservice interactions, communication bottlenecks, scalability patterns, and failure modes under realistic conditions.

By offering standardized benchmarks, DeathStarBench bridges the gap between theoretical microservices research and practical, reproducible experimentation. The project is actively maintained through its core repository on GitHub and continuously receives updates and refinements to remain aligned with contemporary technologies and practices. The comprehensive structure of the suite which offered proper documentation and active maintenance deemed it as an attractive candidate for selecting a testing environment for the project. More specifically, the HotelReservation service was the core microservice environment that was studied, analyzed and deconstructed while conducting this research and

some of its services were the ones chosen to be converted to serverless functions, creating a hybrid environment of the two architectures.

1.4 OpenWhisk Serverless Platform

Apache OpenWhisk is an open-source, distributed serverless computing platform that provides a robust Function-as-a-Service (FaaS) environment for building and deploying event-driven applications. Designed with extensibility, openness, and modularity at its core, OpenWhisk allows developers to deploy functions that are executed dynamically in response to external events such as HTTP requests, database updates, or messaging system triggers.

Serverless functions are packaged as entities called “actions” to achieve successful deployment to the Apache OpenWhisk platform. An action is a stateless, event-driven function that is executed in response to an invocation, triggered by an HTTP request, event source, or internal rule. Each action encapsulates a specific computational task and can be implemented in various programming languages including Go, Python, JavaScript, and Java, or it can be packaged as a Docker container to support arbitrary runtimes. From a serverless perspective, OpenWhisk actions align precisely with the Function-as-a-Service (FaaS) model: they are ephemeral, executed in isolated containers, and automatically scaled based on demand.

OpenWhisk was selected as the target serverless platform for this work, providing a research-friendly environment to automate the deployment, execution, and benchmarking of microservice-converted functions. A more in-depth technical analysis of OpenWhisk’s architecture, operational model, and experimental deployment experiences will be provided in later chapters of the thesis.

1.5 Objective

This thesis project is centered around specifying a core methodology for the precise conversion of an application component from an architecturally microservice application, to a serverless function deployed in a serverless platform and conducting research on the possible automation of such process, resulting in the construction of a testbed environment of a hybrid implementation of a system with both microservice and serverless entities.

The vision is compelling: if we could automatically migrate microservices to serverless

platforms, developers would gain the operational simplicity and cost advantages of serverless computing without incurring the expensive and error-prone process of manual re-architecture. This motivation stems not only from theoretical interest but from extensive hands-on research and experimentation.

1.6 Research Questions

With the background and context established, this thesis will attempt to answer the following research questions:

1.6.1 Successful Conversion and Performance Overview

Is it possible to precisely convert specific service components from an already functional microservice application, intertwined with other similar services and supporting technologies, to fully functional serverless functions that run independently on a separate from the original microservice, serverless platform? In the case of a positive answer to the above question, a follow up question occurs: In what extend is this hybrid implementation of the previous system with the addition of the serverless components, equal in performance and effectiveness to its original version?

1.6.2 Methodology Extraction

Through the process of thorough research and experimentation, could a specific methodology of converting a program that is architecturally a microservice application component to a serverless function that seamlessly replaces the original microservice in full functionality, be formulated? This question will be answered while discovering the needs of this conversion process and the steps that were followed in midst of trial and error of manual conversion attempts of one architecture to the other.

1.6.3 Automation of Conversion

Could the process of converting a microservice application component to a serverless function of a specific serverless platform, be properly automated? What tools and techniques were utilized for the realization of the automation process? The answers to these questions will be

acquired after experimentation with automation techniques utilizing the products of the conversion process from question 1.6.2 in combination code analyzation and generation tools.

Chapter 2

Related Work

2.1	Existing Work for Microservice Serverless Migration.....	13
2.1.1	Boxer Framework.....	13
2.1.2	Boxer Utilization and Extension.....	14
2.1.3	SigmaOS.....	14
2.1.4	Performance Enhancements in Serverless	15
2.1.5	μ 2sls	15
2.2	Choice of Serverless Platform: OpenWhisk.....	15
2.2.1	OpenFaaS	16
2.2.2	Fission	16
2.2.3	Knative	17
2.2.4	Apache OpenWhisk.....	17

This chapter provides a brief review of the existing literature relevant bridging the gap between microservice and serverless architecture and their possible combination. The various serverless platforms that were considered for utilization during this research will also be explored highlighting the one chosen: OpenWhisk.

2.1 Existing Work for Microservice Serverless Migration

In this section of the chapter the current state of the field regarding microservice to serverless migration will be presented. After conducting deep research around the topic of the relation and combination of microservice designed systems with serverless functions, numerous attempts at utilizing the benefits of the two architectures together in singular systems were made by the academic community, products from which are showcased in the following sections.

2.1.1 Boxer Framework

Kansal et al. [1] proposed Boxer, an interposition framework that enables unmodified cloud applications to transparently benefit from serverless elasticity without needing to be rewritten into the traditional event-driven, stateless function model. By intercepting system-level calls

and emulating a conventional network-of-hosts abstraction over AWS Lambda, Boxer creates an environment familiar to existing distributed systems. This approach allows ephemeral elasticity to absorb load spikes and recover from failures with substantially faster reaction times compared to virtual machines, offering potential cost and availability improvements. For this thesis, Boxer serves as an important inspiration in demonstrating that transparent adaptation between microservice models and serverless platforms is achievable. Although Boxer focuses more on runtime system adaptation rather than static automated code transformation, which is the focus of this project, the latter was chosen to avoid the extra processing and performance cost that comes with modifying services while they are running, and to produce portable code that works directly with serverless platforms.

2.1.2 Boxer Utilization and Extension

Wawrzoniak et al. [3] build upon the ideas introduced by Boxer by introducing Imaginary Machines and extend them into a generalized serverless execution model suitable for legacy cloud applications. Without modifying application source code, Imaginary Machines can transparently map cloud applications onto elastic serverless infrastructures. This model highlights the broader vision of serverless evolution towards general-purpose cloud application execution. Although closely aligned in spirit with the goals of this thesis, Imaginary Machines prioritizes system-level adaptations rather than code-level transformation and deployment automation.

The “Off-the-Shelf Serverless” work by Wawrzoniak et al. [6] extends the principles of Boxer to enable unmodified distributed data processing engines, such as Apache Spark and Apache Drill, to operate on AWS Lambda. By introducing a transparent interposition layer, it demonstrates that complex, stateful, communication-heavy applications can run efficiently on serverless platforms, challenging the assumption that serverless is only suited for simple, stateless applications.

2.1.3 SigmaOS

Similarly, SigmaOS by Szekely et al. [8] aims to unify the capabilities of serverless and microservice systems under a single cloud-native operating system, supporting both burst-parallel lightweight tasks and long-running stateful services. These two approaches underline the expanding vision of serverless computing as a viable platform for a broader range of cloud workloads. However, both projects primarily operate at the infrastructure abstraction level

rather than focusing on automated static code transformation, distinguishing their scope from the automation goals pursued in this thesis.

2.1.4 Performance Enhancements in Serverless

FaaS\$T [4] and FaasCache [5] propose improvements that target state management and cold start mitigation in serverless environments, respectively. FaaS\$T introduces an auto-scaling memory caching system that drastically reduces access latency for stateful serverless applications, addressing a common bottleneck for serverless performance. Meanwhile, FaasCache reimagines the problem of function keep-alive as a caching challenge, applying Greedy-Dual caching policies to optimize resource utilization and cold-start mitigation. While neither FaaS\$T nor FaasCache directly tackles microservice-to-serverless conversion, they both address fundamental challenges, specifically state handling and startup efficiency, that critically affect the performance of migrated services.

2.1.5 μ 2sls

Qiu et al. [2] introduced μ 2sls a radically different approach to executing microservice applications on serverless platforms by offering a formally verified execution model. It enables developers to write traditional service logic in Python while providing two additional primitives, transactions and asynchronous calls, that the runtime uses to maintain correctness in the presence of serverless-specific faults, such as re-executions and partial failures. The μ 2sls framework guarantees that serverless executions refine the observable behavior of the original specification. While μ 2sls focuses on ensuring correctness, it assumes developer intervention and restructuring, distinguishing itself from the fully automated conversion approach targeted in this thesis.

2.2 Choice of Serverless Platform: OpenWhisk

A suitable serverless platform had to be chosen to support the research and experimentation conducted in this project. This platform needed to support true Function-as-a-Service capabilities, meaning that the programs it runs must follow strict serverless principles. These include: a) scale-to-zero, where functions automatically stop running when not in use b) ephemeral execution, where each function runs for a short period and shuts down after completing its task c) no server management, meaning the developer does not configure or maintain any servers and d) event-driven invocation, where functions are triggered by requests

or external events. These criteria ensured that the selected platform aligned with the goals of transforming microservices into lightweight serverless actions.

We studied several serverless platforms in the process of defining the most suitable one for the needs of this project. In the following subsections, some of the platform candidates will be presented and the selection of Apache OpenWhisk will be justified.

2.2.1 OpenFaaS

OpenFaaS [9] is a Kubernetes-native serverless platform that emphasizes simplicity, usability, and developer-centric design. It abstracts function deployment through Docker containers and integrates tightly with Kubernetes via the OpenFaaS Operator or with Docker Swarm for lightweight setups. OpenFaaS features native autoscaling, function templates, and a UI/CLI interface for function management. Its strengths lie in its ease of deployment and compatibility with existing container workflows, making it a strong candidate for early experimentation.

However, several limitations emerged upon deeper exploration. While OpenFaaS presents itself as a FaaS system, its architecture fundamentally relies on long-running pods, where functions are persistently active containers, rather than true ephemeral executions. This design reduces cold-start latency but violates the core serverless principle of scale-to-zero, which was essential for this thesis’s benchmarking purposes. Furthermore, OpenFaaS’s eventing model and function management are less modular and more tightly coupled to Kubernetes than required for our automated transformation toolchain. Its reliance on language-specific scaffolding limited its flexibility in supporting arbitrary converted Go [19] microservices. These practical constraints, particularly regarding platform behavior, ultimately led to its exclusion.

2.2.2 Fission

Fission [10] is another Kubernetes-native serverless framework built to offer fast cold-starts, language extensibility, and declarative function deployment using Custom Resource Definitions (CRDs). It emphasizes low-latency execution through the reuse of function containers and supports multiple runtime environments including Python, Go, and Node.js. One of its standout features is Fission Workflows, which allows the chaining of functions in a control-flow manner.

Despite these strengths, Fission’s architecture and development model introduced significant challenges for this thesis. Function definition and deployment are tightly integrated into the Kubernetes control plane via CRDs, which complicates automation efforts for externally transformed code. Fission hides too much of what happens during function execution, which made it hard to control important details like how function programs are deployed, how the environment is set up, and how the runtime behaves. These controls were important for the kind of experiments we needed to run when understanding the conversion of microservices and even potentially automating it. Additionally, the documentation and ecosystem support around Go-based functions in Fission were relatively sparse compared to other platforms. These issues collectively influenced the decision to deprioritize Fission as a deployment platform.

2.2.3 Knative

Knative [11], developed by Google, is a powerful serverless framework built atop Kubernetes, designed to provide a complete set of fundamental tools for deploying event-driven workloads. It introduces key components such as Knative Serving (for autoscaling HTTP workloads), Knative Eventing (for building event-driven applications), and a flexible abstraction for sources events pools. Its appeal lies in its strong adherence to cloud-native design, native Kubernetes integration, and support for complete production deployments in hybrid environments.

However, Knative also represents a relatively heavyweight solution, with a steep operational learning curve, and complex installation requirements including service dependencies like Istio or Kourier. Similarly to the case of OpenFaaS, Knative is also a container-based platform that handles container setup to mimic that of traditional serverless platforms and their core features although primarily being a container management system. These complexities proved to be counterproductive for the goals of this thesis, which required fast iteration cycles, simple repeatable setups, and tight control over the function runtime. The lack of minimal configurations, combined with its emphasis on full-stack deployment approaches, made Knative more suitable for enterprise-scale production environments than research-driven experimentation. As a result, Knative was excluded from further consideration in favor of more lightweight and transparent alternatives.

2.2.4 Apache OpenWhisk

Apache OpenWhisk [12] is an open-source, distributed serverless platform originally developed by IBM and now maintained as part of the Apache Software Foundation. It provides a modular architecture based on Docker and CouchDB, supports function executions in a wide array of languages, and includes a powerful rule-based event triggering system. OpenWhisk is distinct among serverless frameworks due to its support for customizable pluggable runtimes, custom Docker containers, and action chaining, enabling complex workflows and fine-grained control over execution environments. Its CLI tool `wsk` and RESTful APIs make it highly scriptable and offer numerous preconfigured features with interactive platforms applicable within terminal environments, which aligned closely with the project's needs for automating deployment of converted services. OpenWhisk supports features like parameter binding and named packages, which make it easier to group related actions and reuse them across different scenarios. This flexibility was important for setting up and managing the different testing environments used in this research. Additionally, OpenWhisk can run locally, allowing for fast and easy experimentation without needing a cloud setup during development. These attributes made OpenWhisk technically suitable for hosting the converted microservices produced by this project's automated transformation pipeline.

OpenWhisk was ultimately chosen as the serverless execution platform for this thesis due to its balance of flexibility, ease of use and deployment and research-aligned features. Unlike proprietary platforms such as AWS Lambda or heavyweight frameworks like Knative, OpenWhisk enabled full control over deployment mechanics while still adhering to core serverless principles such as statelessness, ephemeral execution, and scale-to-zero semantics. Its support for custom action containers allowed the project to package microservices with complex dependencies, such as MongoDB [18] and Memcached [24] drivers and service initialization code, without compromising execution correctness. Furthermore, its transparent logging, local deployment, and resource observability provided the visibility required for benchmarking converted services against their original microservice implementations. Integration with automation scripts and Makefiles facilitated the deployment of actions without manual overhead, making OpenWhisk an efficiently scriptable and programmable platform for exploring the feasibility of serverless conversion. As such, OpenWhisk was not only a technical fit but a strategic enabler of the thesis's core goal: to explore microservice-to-serverless transformation using repeatable easily implemented methodologies.

The reasoning and criteria taken into consideration for ultimately choosing Apache OpenWhisk over the rest of the platforms and systems mentioned are more clearly demonstrated in the table below:

Table 2.1: Taxonomy table demonstrating the criteria of choosing a serverless platform

Principle / Feature	OpenWhisk	OpenFaaS	Fission	Knative
Scale-to-Zero	Yes	No	Yes	Partial
Ephemeral Execution	True FaaS	Long-lived pods	Container reuse	Simulated via pods
No Server Management	Full abstraction	Requires K8s setup	CRD + cluster access	Full-stack setup needed
Event-Driven Invocation	Triggers and RESTful API	Basic triggers	Workflows and triggers	Advanced eventing
Runtime Transparency	Full	Limited	Hidden	Complex
Local Deployment for Experimentation	Lightweight	Docker/K8s required	No	Complex dependencies
Go Language Support	Native	Available	Limited	Available
Automation-Friendly Architecture	Scriptable CLI and APIs	Template driven	No	Heavy Configuration
Suitable for Research & Prototyping	Ideal	Limited	Unsuitable	Unsuitable

Chapter 3

OpenWhisk Deployment

3.1	OpenWhisk Deployment	20
3.1.1	OpenWhisk Devtools - MacOS	21
3.1.2	OpenWhisk Devtools – Windows Subsystem for Linux: Ubuntu	22
3.1.3	OpenWhisk Documentation Instructions for Local Deployment - MacOS	22
3.1.4	OpenWhisk Documentation Instructions for Local Deployment – WSL Ubuntu	24
3.1.5	OpenWhisk CloudLab Deployment	25

This chapter describes the process of experimenting and finally achieving a stable and repeatable deployment of Apache OpenWhisk. Completing this task was essential for our research, since we needed a non-trivial way of setting up our testing environment that would be consistent and be automatically executed, avoiding constant errors debugging. We aimed at configuring a local setup of interacting containers and serverless functions for simpler development. After trial and error among attempting deployment on different hardware and operating systems, we ultimately choose the route of deploying the platform through Kubernetes [25], configuring a cluster dedicated to OpenWhisk. This was necessary for creating a shared and homogenous environment for both the serverless platform and the microservice application, thus having container-based microservices and converted serverless microservices to work and interact on the same cluster.

3.1 OpenWhisk Deployment

The process of deploying a whole platform as complex as Apache OpenWhisk, from the beginning with no preset environment, proved to be a rigorous process despite being chosen for its lightweight deployment methods. Despite the theoretical reasons as to why OpenWhisk was characterized as the suitable platform adequate for this project, ultimately the choice for its actual utilization was governed by its deployability. In the case the platform proved too difficult or even impossible to be deployed and successfully configured on the systems that

were accessible, another direction would have to be chosen which would potentially mean additional hardships and obstacles occurring.

Initially significant attempts were made at successfully deploying Apache OpenWhisk serverless platform to personal computer hardware hosting different Operating Systems, to more spherically comprehend the ease of both use and configuration of the platform.

3.1.1 OpenWhisk Devtools - MacOS

Specifically, the first attempt at local deployment of OpenWhisk was attempted on Mac Operating System using a version of OpenWhisk from a sub directory of the official Apache documentation for OpenWhisk named “OpenWhisk-devtools” [13]. The repository provided options uniquely developed for local deployment along with other useful tools easing the process of action declaration, development and deployment. The platform is deployed using docker-compose and offers a solution of automation through an assortment of yaml scripts and a core Makefile that simplifies processes in few single commands. It is highlighted that the deployment is compatible on MacOS using the Docker for Desktop [14] application for MacOS systems of Apple Silicon Architecture.

This attempt deemed to be unsuccessful, halting at the point of inability to resolve an error not documented before by the service community of OpenWhisk. The Error occurred in incorrectly running the OpenWhisk invoker container leading to its controller api-gateway getting stuck on repeatedly attempting to reach the invoker node-container, resulting in the deployment halting. Due to not being able to solve the error, another direction had to be taken in the form of attempting the same method on a different Operating System.

```
apigateway-1 | 2024/10/11 07:15:16 done
..apigateway-1 | 2024/10/11 07:15:25 Executing sync cmd: rclone sync minio:api-gateway /etc/api-gateway/
apigateway-1 | 2024/10/11 07:15:26 done
..apigateway-1 | 2024/10/11 07:15:35 Executing sync cmd: rclone sync minio:api-gateway /etc/api-gateway/
apigateway-1 | 2024/10/11 07:15:36 done
..>apigateway-1 | 2024/10/11 07:15:45 Executing sync cmd: rclone sync minio:api-gateway /etc/api-gateway/
apigateway-1 | 2024/10/11 07:15:46 done
..apigateway-1 | 2024/10/11 07:15:55 Executing sync cmd: rclone sync minio:api-gateway /etc/api-gateway/
apigateway-1 | 2024/10/11 07:15:56 done
```

Figure 3.1: Command Line output during OpenWhisk-devtools unsuccessful deployment of OpenWhisk on MacOS

3.1.2 OpenWhisk Devtools – Windows Subsystem for Linux: Ubuntu

Another attempt at this deployment method was executed before following to another OpenWhisk version, on a different Operating System: Ubuntu – specifically on the version offered by the Windows Subsystem for Linux in a partitioned environment hosted in combination with the Windows OS on the same hardware. The OpenWhisk-devtools docker-compose method of deployment also highlights compatibility with any kind of installation of Docker version 1.13 and onward as well as Docker Compose version 1.6 and onward, theoretically deeming the environment eligible for success.

The attempt resulted in another error of a different kind. In this case, the system was able to complete the fetching process of the container images and their internal set up, however issues in regards to container creation rose to the surface while executing the scripted deployment of the system as described within the Makefile. An error reported by the Docker daemon indicated that it was unable to extract a specific image layer and failed with a `NotFound` message related to the content digest. This pointed to a missing or inaccessible image layer, either due to its removal from the remote registry, an internal corruption within the local Docker content store, or inconsistencies in nightly build tags for OpenWhisk services.

Several additional steps were subsequently attempted: a full Docker system prune to clear cached and dangling layers, manual re-pulling of affected images, and substitution of the unstable “:nightly” tag with more stable versioned tags. Despite these efforts, the issue persisted, preventing a successful container orchestration and halting the platform initialization. The failure highlighted the fragility of relying on nightly image tags and underlined the need for controlled Docker environments with maintained version..

[illegible]

Figure 3.2: Command Line output during OpenWhisk-devtools unsuccessful deployment of OpenWhisk on WSL Ubuntu

3.1.3 OpenWhisk Documentation Instructions for Local Deployment - MacOS

In addition to its containerized deployment options from the previous method, Apache OpenWhisk also offers a Java-based standalone deployment mode intended for developers seeking a locally hosted runtime environment. Following the official OpenWhisk documentation, the apache/OpenWhisk GitHub repository [15] was utilized to clone and build the core platform directly from source. This method involves compiling the OpenWhisk controller and invoker services using Gradle and launching them as native Java processes. Requirements for compatible Docker, Node.js and Java software in specific versions was demanded for the process to execute in a successful manner that are stated within the repository documentation.

Despite the intended simplicity of the Java-based deployment, initial attempts to build OpenWhisk from source encountered a critical failure during the Gradle initialization phase. The build process terminated with a semantic error:

```
Unsupported class file major version 67
```

indicating that the Groovy/Gradle toolchain embedded in the OpenWhisk build scripts was incompatible with the Java bytecode compiled using a more recent JDK version such as Java 13 or later. Specifically, this exception stemmed from a mismatch between the version of the Java Development Kit installed on the host system and the expected JDK compatibility defined in OpenWhisk's build configuration, which targets Java 8 or 11. Multiple mitigation attempts were made, including switching the system Java version to Java 11 and ensuring that JAVA_HOME was properly configured. However, the error persisted even under the downgraded JDK, suggesting deeper issues with Groovy's bytecode parsing or potential corruption within the Gradle daemon cache. As a result, the Java-based build method had to be suspended.

```

ioannisantonio@Ioannis-MacBook-Air openwhisk % ./gradlew core:standalone:bootrun
Starting a Gradle Daemon (subsequent builds will be faster)

FAILURE: Build failed with an exception.

* Where:
Settings file '/Users/ioannisantonio/Desktop/OpenWhisk/openwhisk/settings.gradle'

* What went wrong:
Could not compile settings file '/Users/ioannisantonio/Desktop/OpenWhisk/openwhisk/settings.gradle'.
> startup failed:
  General error during semantic analysis: Unsupported class file major version 67

java.lang.IllegalArgumentException: Unsupported class file major version 67
  at groovyjarjarasm.asm.ClassReader.<init>(ClassReader.java:196)
  at groovyjarjarasm.asm.ClassReader.<init>(ClassReader.java:177)
  at groovyjarjarasm.asm.ClassReader.<init>(ClassReader.java:163)
  at groovyjarjarasm.asm.ClassReader.<init>(ClassReader.java:284)
  at org.codehaus.groovy.ast.decompiled.AsmDecompiler.parseClass(AsmDecompiler.java:81)
  at org.codehaus.groovy.control.ClassNodeResolver.findDecompiled(ClassNodeResolver.java:251)
  at org.codehaus.groovy.control.ClassNodeResolver.tryAsLoaderClassOrScript(ClassNodeResolver.java:189)
  at org.codehaus.groovy.control.ClassNodeResolver.findClassNode(ClassNodeResolver.java:169)
  at org.codehaus.groovy.control.ClassNodeResolver.resolveName(ClassNodeResolver.java:125)
  at org.codehaus.groovy.ast.decompiled.AsmReferenceResolver.resolveClassNullable(AsmReferenceResolver.java:57)
  at org.codehaus.groovy.ast.decompiled.AsmReferenceResolver.resolveClass(AsmReferenceResolver.java:44)
  at org.codehaus.groovy.ast.decompiled.AsmReferenceResolver.resolveNonArrayType(AsmReferenceResolver.java:79)
  at org.codehaus.groovy.ast.decompiled.AsmReferenceResolver.resolveType(AsmReferenceResolver.java:70)
  at org.codehaus.groovy.ast.decompiled.MemberSignatureParser.createMethodNode(MemberSignatureParser.java:57)
  at org.codehaus.groovy.ast.decompiled.DecompiledClassNode$2.get(DecompiledClassNode.java:234)
  at org.codehaus.groovy.ast.decompiled.DecompiledClassNode$2.get(DecompiledClassNode.java:231)
  at org.codehaus.groovy.ast.decompiled.DecompiledClassNode.createMethodNode(DecompiledClassNode.java:242)
  at org.codehaus.groovy.ast.decompiled.DecompiledClassNode.lazyInitMembers(DecompiledClassNode.java:199)
  at org.codehaus.groovy.ast.decompiled.DecompiledClassNode.getDeclaredMethods(DecompiledClassNode.java:122)
  at org.codehaus.groovy.ast.ClassNode.tryFindPossibleMethod(ClassNode.java:922)
  at org.codehaus.groovy.ast.ClassNode.tryFindPossibleMethod(ClassNode.java:1280)
  at org.codehaus.groovy.control.StaticImportVisitor.transformMethodCallExpression(StaticImportVisitor.java:252)
  at org.codehaus.groovy.control.StaticImportVisitor.transform(StaticImportVisitor.java:113)
  at org.codehaus.groovy.ast.ClassCodeExpressionTransformer.visitExpressionStatement(ClassCodeExpressionTransformer.java:142)
  at org.codehaus.groovy.ast.stmt.ExpressionStatement.visit(ExpressionStatement.java:40)
  at org.codehaus.groovy.ast.ClassCodeVisitorSupport.visitClassCodeContainer(ClassCodeVisitorSupport.java:110)
  at org.codehaus.groovy.ast.ClassCodeVisitorSupport.visitConstructorOrMethod(ClassCodeVisitorSupport.java:121)
  at org.codehaus.groovy.ast.ClassCodeExpressionTransformer.visitConstructorOrMethod(ClassCodeExpressionTransformer.java:53)

```

Figure 3.3: Command Line output during Java-based standalone unsuccessful deployment of OpenWhisk on MacOS

3.1.4 OpenWhisk Documentation Instructions for Local Deployment – WSL Ubuntu

An additional attempt to deploy Apache OpenWhisk via the Java-based standalone build was carried out within a WSL-based Ubuntu environment on Windows. While this setup was expected to provide a more Linux-native context for running the standalone runtime, it encountered a critical networking failure during the early stages of service initialization. Specifically, the OpenWhisk controller attempted to bind to `http://172.17.0.1:3233`, a Docker bridge address that is typically accessible in standard Linux environments. However, WSL2 does not natively expose or preserve this address in the same manner, leading to a `java.net.BindException: Cannot assign requested address`. The error occurred repeatedly as the invoker attempted to start prewarm containers for node.js actions, all of which failed due to unreachable internal addresses.

Attempts to modify Docker networking or reroute to localhost were unsuccessful due to WSL's namespace handling and its abstraction of the Linux network stack on top of Windows. As a result, the WSL-based standalone deployment was categorized as unreliable for OpenWhisk.

For this thesis' experimental setup, an already preconfigured CloudLab profile was utilized that incorporates the deployment process described by the repository mentioned, in a Kubernetes environment. The profile utilized the Kind tool [17] for containerization of Kubernetes – Kubernetes in Docker – to first achieve a functional iteration of a Kubernetes Cluster on the reserved family of machine nodes from CloudLab. Executing an OpenWhisk deployment on top of the Kube Cluster is an optional functionality of the profile, that if configured to do so deploys the version of OpenWhisk from the corresponding CloudLab repository, with Helm [18] as a package manager tool.

Helm provided a declarative and reusable mechanism to manage configuration and deployment lifecycle. The official OpenWhisk Helm chart abstracts away low-level Kubernetes YAML definitions by encapsulating them into modular templates. Helm enhanced the ease of system configuration at great scale, with its automated options in partial or whole redeployment of the environment in a few easily comprehensible commands, significantly decreasing workflow complexity. This allowed for easier provisioning of OpenWhisk in Kubernetes environments and made it easier to experiment with scaling, and resource allocation by simply modifying Helm value files. Helm's declarative and repeatable deployment structure, allowed for scalable and reproducible experimentation, which was essential for the benchmarking and debugging in the iterative refinement phases of this research.

This implementation of the deployment workflow is default to be implemented on three nodes/machines rented from CloudLab's maintained node clusters, for this project's needs the m510 machines were primarily utilized as they are the type of node the profile has been most tested on. The topology of the partitioned node cluster is consisted of one controller node and two invoker nodes at minimum.

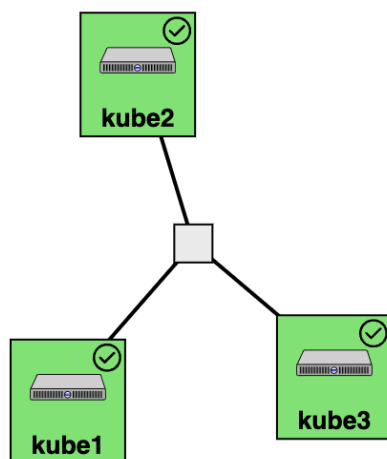


Figure 3.5: Topology View of a Deployed CloudLab Experiment using the kubernetes Profile

While interacting with the environment prepared by the profile in multiple iterations of its execution, occasional inconsistent behaviors were encountered often regarding the OpenWhisk's CLI tool: wsk configuration not functioning as expected as well as issues in relation to incorrect declaration of authentication parameters demanded by OpenWhisk such as API_HOST and AUTH_KEY. This behavior was avoided by making sure the startup processes of all nodes, especially the controller nodes finish executing successfully by CloudLab, despite the timely wait times approaching approximately 10 to 12 minutes.

ID	Node	Type	Cluster	Status	Startup
kube1	ms0834	m510	Utah	ready	Finished
kube2	ms0843	m510	Utah	ready	Finished
kube3	ms0814	m510	Utah	ready	Finished

Figure 3.6: List View of a Deployed CloudLab Experiment using the kubernetes Profile that has finished startup process execution

ID	Node	Type	Cluster	Status	Startup
kube1	ms0931	m510	Utah	ready	Running
kube2	ms0919	m510	Utah	ready	Finished
kube3	ms1217	m510	Utah	ready	Finished

Figure 3.7: List View of a Deployed CloudLab Experiment using the kubernetes Profile that is currently executing startup processes

Ultimately, the configuration hosted on CloudLab proved to be the most fitting for our purposes. Not only did it provide an automated, error-free method of deployment of the platform that could be executed remotely as a background process, it fully prepared a significantly useful and well-orchestrated Kubernetes Cluster. The cluster would serve as an overall execution environment for both traditional microservice applications and their serverless iterations, in a homogenous setting.

Chapter 4

DeathStarBench Microservice Deployment

4.1	Choice of DeathStarBench Service	29
4.1.1	Social Network Application	30
4.1.2	Hotel Reservation Application	30
4.2	Deployment Experimentation.....	31
4.2.1	Hotel Reservation Helm-Chart Deployment.....	31
4.2.2	Social Network Helm-Chart Deployment.....	33
4.3	Selected Service: Hotel Reservation	34

4.1 Choice of DeathStarBench Service

The DeathStarBench Suite of microservice programs offers several fully functional and maintained collections of services that truly follow the microservice architecture of distributed web applications, assuming them as suitable candidates for understanding the principles of said architecture. Each application is designed to reflect real-world service architectures with complex internal dependencies, databases, and network communication.

Among its most prominent applications is the Social Network, which emulates a social media platform featuring services for user timelines, post composition, media management, and social graphs. Additionally, the suite includes the Hotel Reservation system designed to replicate an end-to-end hotel booking service, implementing search, reservation, user profile, geo-location, and rate management services. Also included are workloads such as an E-Commerce platform for simulating online retail, a Media Service representing content delivery and review systems, and an Edge Coordination service that manages distributed IoT devices such as drones.

All applications offer an adequate number of features that increase the complexity of the interaction environment and propose a stable system of internal services that utilize common

supporting technologies found in real-world usage. The two primary candidates that were ultimately explored for the main conversion and benchmarking environment demanded for this project were the Social Network and Hotel Reservation microservice applications. Other DeathStarBench applications, such as the Media Service, E-Commerce, and Drone Coordination benchmarks, were excluded due to factors such as higher system complexity, lack of clear documentation, or reliance on less-supported technologies and languages, which made them less suitable for controlled experimentation and automation within the scope of this thesis. The two main services were chosen for the reason being their complementary characteristics and relevance to the research objectives.

4.1.1 Social Network Application

The Social Network application represents a complex polyglot microservice ecosystem, consisted of a high number of independent services in comparison to the other optional microservice systems, with deep reliance on caching and memory management. Each service was coupled with their own personalized supporting service for handling these aspects, which presented an opportunity to outsource state related dependencies of standalone microservices to those supporting technologies. The utilization of Apache Thrift as the RPC protocol streamlines the application's internal communications and poses as an attractive factor for handling the intricacies of also communicating with an external entity such as OpenWhisk.

4.1.2 Hotel Reservation Application

The Hotel Reservation application offers a clean, modular architecture entirely implemented in Go, aligning with one of the languages natively supported by OpenWhisk which was a great benefit. Its limited number of services makes it manageable for initial experiments while still embodying realistic patterns of service orchestration, database interaction, and RPC-based communication. Using the same programming language across all services makes the system easier to understand and work with during early research, especially when studying how services function and depend on each other. However, this also means that the system uses gRPC, a language-specific communication method tied to Go. Replacing gRPC with a method compatible with OpenWhisk, like REST APIs, can be challenging because it's closely built into how the services communicate.

4.2 Deployment Experimentation

Similar to the approach followed on finalizing the choice of serverless platform deployment, stability and consistency of the process were the core deciding factors for future utilization of the service chosen on the research. The applications are not limited to a single deployment option, having available alternative approaches at deployment using different tools and platforms such as Docker with Docker-Compose, Openshift as well as implementations through Kubernetes by using the Helm-Chart package manager. The inclusion of a deployment implementation through Helm-Chart enhances the incentive to utilize a shared cluster manager between both OpenWhisk and the deployed microservice application in question. This would allow the design of a more streamlined environment setup process, easily repeated and replicated, while at the same time alleviating the difficulty of managing two independent systems hosting OpenWhisk and the application.

Therefore, Helm routed deployments were a priority while tuning the research environment, which had to appropriately configured in order to maintain a stable platform for experimentation. Despite the benefit of the active maintenance nature of the DeathStarBench microservice applications by the development team, the Helm-Chart documentation appeared to be lacking in specific areas and the deployment configuration introduced a variety of new issues in relation to startup processes of both Social Network and Hotel Reservation applications. Resolving these issues was crucial for the continuation of the project as well as deciding for the final selection of the microservice application to be studied under this research.

4.2.1 Hotel Reservation Helm-Chart Deployment

Initial attempts to deploy the Hotel Reservation microservices using the Helm chart provided within the DeathStarBench repository revealed multiple misconfigurations that caused deployment failures across several components. The current unchanged configuration from the repository itself led to CrashLoopBackOff errors in all core services.

NAME	READY	STATUS	RESTARTS	AGE
consul-hotelres-754f44b5cc-x9rzq	1/1	Running	0	106s
frontend-hotelres-5459cb9f55-cfbtz	0/1	CrashLoopBackOff	2 (26s ago)	106s
geo-hotelres-5f7855f5f5-ls7rm	0/1	CrashLoopBackOff	3 (22s ago)	106s
jaeger-hotelres-99cd86797-9f558	1/1	Running	0	106s
memcached-profile-1-hotelres-5bcb679d5-85jpk	1/1	Running	0	106s
memcached-rate-1-hotelres-7b479c8957-f6qft	1/1	Running	0	106s
memcached-reserve-1-hotelres-668cb94967-vr42p	1/1	Running	0	106s
mongodb-geo-hotelres-5cbbb68c4c-vjzqj	1/1	Running	0	104s
mongodb-profile-hotelres-6bb48b7555-g6s45	1/1	Running	0	106s
mongodb-rate-hotelres-7f7fd44595-jd6tf	1/1	Running	0	105s
mongodb-recommendation-hotelres-58b75d7c95-bv8zq	1/1	Running	0	106s
mongodb-reservation-hotelres-67d8854b87-rmm5q	1/1	Running	0	106s
mongodb-user-hotelres-5b55f5b5c9-jm7qf	1/1	Running	0	105s
profile-hotelres-5478c658bf-zl2pj	0/1	RunContainerError	3 (8s ago)	105s
rate-hotelres-69b86b9bdd-h5zz9	0/1	CrashLoopBackOff	2 (25s ago)	105s
recommendation-hotelres-7b75f7644c-r5j5c	0/1	CrashLoopBackOff	2 (27s ago)	105s
reservation-hotelres-7c7d557c67-gq7dz	0/1	RunContainerError	3 (7s ago)	106s
search-hotelres-5dc66b4d54-xn69m	0/1	CrashLoopBackOff	2 (26s ago)	105s
user-hotelres-7b5d6476f6-9j47w	0/1	CrashLoopBackOff	3 (20s ago)	106s

Figure 3.6: CrashLoopBackOff Errors occurred from out-of -box Helm-Chart deployment of Hotel Reservation application

```
Warning Failed 71s (x5 over 2m49s) kubelet Error: failed to start container "hotel-reserv-frontend": Error response from daemon: OCI runtime create failed: container_linux.go:380: starting container process caused: exec: "./frontend": stat ./frontend: no such file or directory: unknown
Warning BackOff 29s (x9 over 2m15s) kubelet Back-off restarting failed container
```

Figure 3.7: Pod error description logged after the occurrence of CrashLoopBackOff error

Moreover, there was a misalignment between the service definitions and their corresponding MongoDB instances. These misconfigurations stemmed from the use of outdated environment variables and service discovery names inherited from the original Docker Compose deployment environment. MongoDB and other service connection strings, for example, were fetched from a config.json file that contained a configuration map. The map explicitly instructs the system on how to handle the structure and mapping of the service names to align with the Kubernetes DNS-based resolution model. The issue was traced to Helm's values.yaml file misconfiguration, unique for each particular service, that failed to set the environment directory of execution correctly. More specifically, services were referencing binaries under the default path `./<service-name>` instead of the expected `/go/bin/<service-name>`.

```
{{- define "hotelreservation.templates.service-config.json" }}
{
  "consulAddress": "consul-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:8500",
  "jaegerAddress": "jaeger-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:6831",
  "FrontendPort": "5000",
  "GeoPort": "8083",
  "GeoMongoAddress": "mongodb-geo-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:27018",
  "ProfilePort": "8081",
  "ProfileMongoAddress": "mongodb-profile-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:27019",
  "ProfileMemcAddress": "{{ include "hotel-reservation.generateMemcAddr" (list . .Values.global.memcached.HACount "memcached-profile" 11213) }}",
  "RatePort": "8084",
  "RateMongoAddress": "mongodb-rate-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:27020",
  "RateMemcAddress": "{{ include "hotel-reservation.generateMemcAddr" (list . .Values.global.memcached.HACount "memcached-rate" 11212) }}",
  "RecommendPort": "8085",
  "RecommendMongoAddress": "mongodb-recommendation-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:27021",
  "ReservePort": "8087",
  "ReserveMongoAddress": "mongodb-reservation-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:27022",
  "ReserveMemcAddress": "{{ include "hotel-reservation.generateMemcAddr" (list . .Values.global.memcached.HACount "memcached-reserve" 11214) }}",
  "SearchPort": "8082",
  "UserPort": "8086",
  "UserMongoAddress": "mongodb-user-{{ include "hotel-reservation.fullname" . }}.{{ .Release.Namespace }}.svc.{{ .Values.global.serviceDnsDomain }}:27023"
}
{{- end }}
```

Figure 3.8: Configuration Map found in config.json file showcasing environment variable mapping to Kubernetes-compliant service names

Ultimately, the resolution involved rewriting each values.yml file for all services, fixing the container command directory of execution from `./<service-name>` to `/go/bin/<service-name>`. Additionally, correctly specifying the mount path from which the configuration map is referenced had to be corrected, from simply `config.json` to `/workspace/config.json`. These changes in values.yml file structure correctly set the environment of deployment for all sub-services of the application and directs them to extract the appropriate names for referencing the rest of the services from the configuration map. After this simple but crucial adjustment, the service was successfully deployed and fully functional.

```
name: user

ports:
  - port: 8086
    targetPort: 8086

container:
  command: ./user
  image: deathstarbench/hotel-reservation
  name: hotel-reserv-user
  ports:
    - containerPort: 8086

configMaps:
  - name: service-config.json
    mountPath: config.json
    value: service-config
```

Figure 3.9: Incorrect values.yml file for User Service

```
name: user

ports:
  - port: 8086
    targetPort: 8086

container:
  command: /go/bin/user
  image: deathstarbench/hotel-reservation
  name: hotel-reserv-user
  ports:
    - containerPort: 8086

configMaps:
  - name: service-config.json
    mountPath: /workspace/config.json
    value: service-config
```

Figure 3.10: Correct values.yml file for User Service

4.2.2 Social Network Helm-Chart Deployment

The Social Network application, as larger and more complex, also demanded a few critical interventions for deployment via Helm. Initial deployment attempts encountered issues specifically within the Horizontal Pod Autoscaler (HPA) template files. The service chart inherited auto-scaling configurations from a base file named `_baseHPA.tpl`, which declared global-level scaling parameters. These caused Helm to fail due to conflicting or missing references within non-global value declarations, particularly when default values were not explicitly overridden in the values.yml file. Resolving the issue required removing HPA configurations at the individual service level, ensuring the template logic evaluated valid

metrics references. The solution that ultimately allowed the application to execute was achieved by adjusting these if condition statements within the `_baseHPA.tpl` file:

```
{{- if or
$.Values.global.hpa.targetMemoryUtilizationPercentage (and
.Values.hpa .Values.hpa.targetMemoryUtilizationPercentage) }}
to
{{- if or
$.Values.global.hpa.targetMemoryUtilizationPercentage}}
, and also
{{- if or $.Values.global.hpa.targetCPUUtilizationPercentage
(and .Values.hpa .Values.hpa.targetCPUUtilizationPercentage)
}}
to
{{- if or
$.Values.global.hpa.targetCPUUtilizationPercentage}}.
```

4.3 Selected Service: Hotel Reservation

Although both the Hotel Reservation and Social Network applications from the DeathStarBench suite were ultimately deployed successfully within the Kubernetes cluster using Helm, the Hotel Reservation application was selected as the primary subject for serverless transformation in this thesis due to its greater structural alignment with the conversion goals and constraints of the project. Specifically, Hotel Reservation offers a more uniformly implemented service architecture, with all microservices written in Go, simplifying code analysis and transformation pipeline.

In contrast, the Social Network application's polyglot architecture, with services written in six different languages, introduced enhanced complexity and difficulty in familiarizing with them and keeping track of language-specific intricacies. Furthermore, the Social Network's higher inter-service communication density and media processing layers made it a less suitable candidate for initial conversion attempts and performance benchmarking. Incidentally, Hotel Reservation's more homogeneous codebase, clearer service boundaries, and manageable complexity offered an ideal and approachable development environment. This allowed the research to focus on the core challenges of microservice-to-serverless transformation without being interrupted by language heterogeneity and complexity.

Chapter 5

Hotel Reservation Microservice Application

Overview

5.1	Overview	35
5.1.1	Breakdown of Core Services	36
5.1.2	Communication Protocols and Runtime Behavior	36
5.1.3	Service Code Structure and Components	37
5.1.4	Relevance and Observability	39
5.2	User Service Breakdown	39
5.2.1	User Service Structure	40

This chapter describes the process of analyzing and comprehending the structure and functionality of the Hotel Reservation Microservice Application as well as some of its consisting subservices.

5.1 Overview

The Hotel Reservation application from DeathStarBench is a representative, production-grade microservice benchmark that models the backend logic of a hotel booking platform. It is designed to test and evaluate modern microservice architectures under realistic workloads and demonstrates key patterns of distributed application design. The system is composed of eight stateless microservices, each responsible for a distinct part of the booking pipeline, such as user authentication, profile management, geo-location resolution, hotel search, rate calculation, reservation handling, and recommendation generation. As established, each service is implemented in Go, reflecting a uniform language choice that simplifies orchestration and tooling.

The application integrates widely used backend technologies including MongoDB for persistent storage and Memcached for caching, ensuring high performance and scalability under load. These components are containerized and orchestrated in cloud-native environments

such as Kubernetes, making the application suitable for studying the deployment, execution, and transformation of microservices in realistic settings.

5.1.1 Breakdown of Core Services

Each microservice in the Hotel Reservation system serves an independent purpose and communicates with other services to fulfill broader user requests. The Geo Service maps user-provided addresses or coordinates to internal region identifiers, leveraging static datasets and spatial resolution logic. The Search Service uses this information to query nearby hotels based on location and availability, pulling a hotel dataset from a MongoDB service. The Rate Service retrieves dynamic pricing information for specific hotels and dates, often combining data from both databases and in-memory caches like Memcached to reduce latency. The Reservation Service is responsible for handling new bookings and cancellations, interfacing with storage layers to ensure consistency. The Recommendation Service analyzes user profiles and booking histories to suggest hotels tailored to individual preferences. Each of these services can operate independently and collaborates through defined interfaces, forming a coordinated.

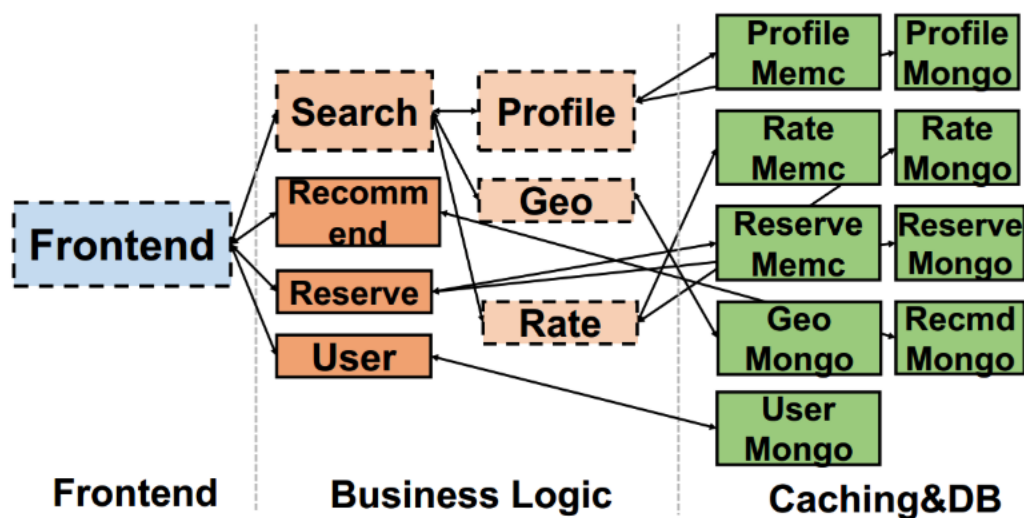


Figure 4.1: Graphical Representation of Hotel Reservation Microservice Application Internal Service Communication Structure

5.1.2 Communication Protocols and Runtime Behavior

Communication between services in the Hotel Reservation application is implemented using gRPC, a high-performance, open-source RPC framework developed by Google. GRPC uses

Protocol Buffers, `proto3`, as its interface definition language, enabling strongly typed inter-service communication. Each microservice defines its service interface in a `.proto` file, which is compiled into Go bindings using the `protoc` compiler and associated gRPC plugins. These bindings generate both the client and server-side interfaces, simplifying implementation and maintaining consistency across the system. Runtime communication occurs over HTTP/2. Additionally, the application employs distributed tracing frameworks like Jaeger to monitor request propagation across services, using gRPC interceptors to automatically propagate tracing metadata with each call. This architecture ensures both low-latency communication and observability.

5.1.3 Service Code Structure and Components

The microservices in Hotel Reservation follow a standardized internal code structure that emphasizes modularity. Each service contains a `main.go` file that acts as the entry point, initializing configuration, logging, tracing, and service registration before launching the gRPC server. Core logic is encapsulated in a `server.go` file, which implements the generated gRPC interface by wiring business logic into the service methods. The Protocol Buffers are defined under a `proto/` directory and compiled into `*_pb.go` and `*_grpc.pb.go` files during the build process. Database configuration for each service is found within the `db.go` file, dedicated to initializing and interacting with the MongoDB backend that stores user credential data

MongoDB integration is handled via the official Go driver [21], typically abstracted behind internal helper functions for loading and querying data. Memcached clients are initialized in services that benefit from fast in-memory caching, such as rate or search-related services. Environment variables are used for injecting runtime parameters such as ports, database addresses, and external service URLs, through JSON-based config files ensuring consistency across deployment environments.



Figure 4.2: Tree structure of services directory of Hotel Reservation Application

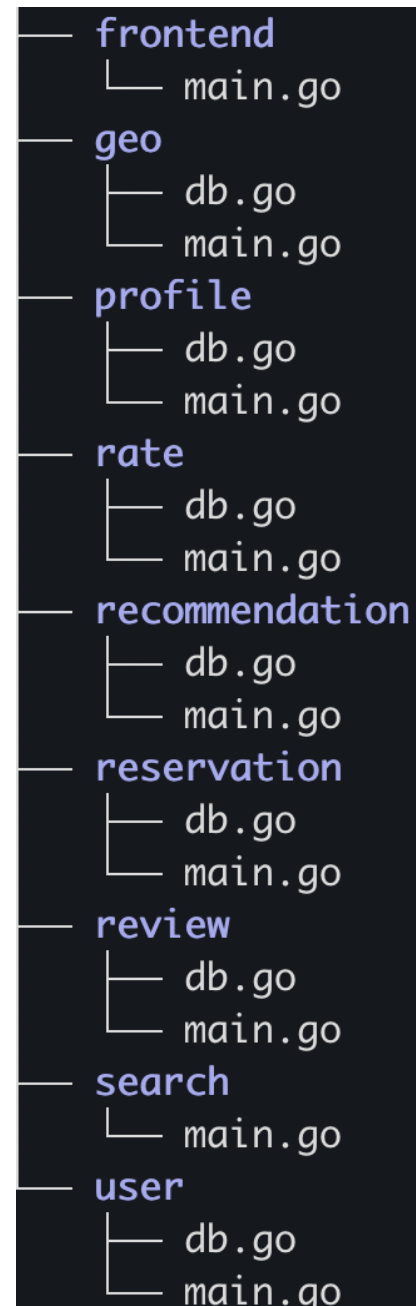


Figure 4.3: Tree structure of cmd directory of Hotel Reservation Application

The architecture of the Hotel Reservation application leverages external supporting technologies such as MongoDB and Memcached for persistent storage and caching, allowing services to offload much of their data handling and memory requirements. Each core service is paired with its own dedicated MongoDB instance, used exclusively for storing and retrieving its operational data, while Memcached instances provide efficient access to frequently used information for some more load-heavy services. This design proves advantageous when converting services to serverless functions, as it enables the externalization of state. Even if a service contains some internal logic that temporarily maintains state, the presence of these

external systems offers a clear path for restructuring such logic into a stateless execution model. The serverless functions can interact with these supporting services on demand, maintaining the expected behavior while conforming to the stateless nature required by the serverless platform.

```
memcached-profile-1-hotelres-5bcbb679d5-ngx5k  
memcached-rate-1-hotelres-7b479c8957-dxf8t  
memcached-reserve-1-hotelres-668cb94967-fzf79  
mongodb-geo-hotelres-5cbbb68c4c-xrgjr  
mongodb-profile-hotelres-6bb48b7555-c74sm  
mongodb-rate-hotelres-7f7fd44595-xq9vn  
mongodb-recommendation-hotelres-58b75d7c95-vv5k6  
mongodb-reservation-hotelres-67d8854b87-4wc9z  
mongodb-user-hotelres-5b55f5b5c9-6c2zf
```

Figure 4.4: MongoDB pod instances for all services

5.1.4 Relevance and Observability

The Hotel Reservation application is particularly well-suited for research on microservice transformation due to its modularity, language uniformity, and realistic data interactions. Its structure mimics common enterprise deployments, where services are decoupled, independently deployable, and communicate through well-defined channels. The use of widely adopted tools like gRPC, MongoDB, and container-based deployment makes the system approachable while still maintaining production-level complexity. These features make Hotel Reservation not only a strong benchmark for performance and scalability evaluation, but also an ideal candidate for experimental transformation into serverless architectures, where clear service boundaries, statelessness, and transparent communication are essential for success.

5.2 User Service Breakdown

In order to comprehend the requirements of converting a microservice to a serverless function, a service had to be chosen from the Hotel Reservation application for deep analyzation of its internal structure and functionality. Understanding the core logic and how it is implemented in the lower levels of code development as well as using specific technologies such as MongoDB and Memecached protocols, was of high significance in discovering a robust conversion process and methodology.

Therefore, a service that included the base level components used by the majority of the services in the application while also maintaining a lower level of logical complexity uniquely implemented for its specific functionality had to be selected. This would help identify common patterns used throughout the system, while avoiding issues caused by deeply specialized functionality. The service that satisfied these demands was the User Service, combining simplicity of service-specific logic implementation with a complete model of service architecture.

Beyond its structure, the User Service was also a good fit in terms of performance and scalability. Since it handles user login and authentication, it is called frequently and must respond quickly. Its operations are short and efficient, mostly involving basic input checks and database lookups. This means it uses few resources and does not require long processing time. The service also experiences sudden increases in traffic, such as when many users log in at once, which makes it a good test case for studying how serverless platforms like OpenWhisk scale to meet demand. Because of this, converting the User Service gave useful results both for testing correctness and for understanding performance under realistic workloads.

5.2.1 User Service Structure

As mentioned, each service encapsulates its functionality in a collection of files that serve a different purpose adhering to a modular and layered file structure, ultimately combining when executed to a complete system. At the entry point lies the `main.go` file, located in the `cmd/user/` directory, which calls upon the services provided by the other files implemented in the system. This file is responsible for bootstrapping the service: it reads configuration parameters from a `config.json` file:

```
jsonFile, err := os.Open("config.json")
```

establishes a MongoDB connection using a custom `initializeDatabase` function which is implemented in the `db.go` file:

```
log.Info().Msg("Initializing DB connection...")
```

```
mongoClient, mongoClose :=
initializeDatabase(result["UserMongoAddress"])
defer mongoClose()
```

utilizing said parameters extracted from the configuration map found in config.json file and initializes tracing and service discovery through Jaeger and Consul respectively. It then launches the gRPC server via a call to `srv.Run()`:

```
srv := &user.Server{ Port: servPort, IpAddr: servIP, Tracer: tracer,
Registry: registry, MongoClient: mongoClient }
log.Fatal().Msg(srv.Run().Error())
```

The `db.go` file in the User Service of the Hotel Reservation application is dedicated to initializing and interacting with the MongoDB backend that stores user credential data. It defines a `User` struct with BSON field tags to map Go struct fields to MongoDB document keys:

```
type User struct {
    Username string `bson:"username"`
    Password string `bson:"password"`
}
```

The core function, `initializeDatabase`, accepts a MongoDB connection string, constructs a URI, and creates a new client using the official MongoDB Go driver. It then populates the database with synthetic test users by generating 500 usernames and corresponding passwords, each hashed using the SHA-256 algorithm for secure storage:

```
sum := sha256.Sum256([]byte(password))
newUsers = append(newUsers, User{
    fmt.Sprintf("Cornell_%x", suffix),
    fmt.Sprintf("%x", sum),
})
```

These users are inserted in bulk into the "user" collection within the "user-db" database. After the insertion, a cleanup function is returned to disconnect the MongoDB client when the service shuts down.

Service logic is implemented in the `server.go` file, located under `services/user`. Understanding the contents of this file was crucial for pinpointing the basic functionality of the service that needed to be extracted and fit into the serverless function model. The file implements three core methods:

The `Run` method handles the service's gRPC configuration which will accept requests from other entities by setting up a gRPC server instance and binds it to a TCP port. It also registers the service with Consul for dynamic service discovery.

In addition, the `loadUsers` method is implemented with the purpose of fetching all the user data stored inside the corresponding MongoDB database for the User service: `mongodb-user`, and storing them in a map data structure that matches each instance of a username with the appropriate password:

```
collection := client.Database("user-db").Collection("user")
curr, err := collection.Find(context.TODO(), bson.D{})
if err != nil {
    log.Error().Msgf("Failed get users data: ", err)
}

var users []User
curr.All(context.TODO(), &users)
if err != nil {
    log.Error().Msgf("Failed get users data: ", err)
}
```

The usage of this function adds a level of state to the lifetime environment of the system. As the function is called upon in the `Run()` method mentioned before, as a startup process, the entirety of the user information is loaded to the memory allocated to the service while it is alive and is kept in an easily accessible data structure for instant reference when user validation requirements are demanded from the service from outside entities. The Hotel Reservation application in general does not offer a method for users to create new accounts with customized user data, therefore the user service is designed to preload the user information to memory and use the map data structure to instantly access it in any instance the service is called upon to execute user validation. This design is justified since there is no need for the map to be updated with new user data introduced to the application environment. At a microservice level, this

choice severely decreases latency levels for the access to user data and nullifies additional latencies that occur when that data must be fetched.

This approach introduces state within the main service logic, as the user data remains in memory for the lifetime of the service. While this design significantly improves performance by eliminating repeated database queries, it conflicts with the stateless model required by serverless platforms like OpenWhisk. In a serverless setting, each function invocation must operate independently, with no persistent in-memory state between calls. Replacing map with a stateless alternative would require each request to fetch user data directly from the database or a shared external cache. This can lead to increased latency, especially under heavy load. To address these drawbacks, solutions such as FaaS-oriented caching systems for example, FaaSCache mentioned before, can be utilized. These systems provide shared and fast-access memory across function instances, helping to maintain low-latency access to frequently used data.

Lastly, the CheckUser method is implemented, encapsulating the main functionality of the User Service as a whole, validating user information after referencing the appropriate preconfigured user data. The CheckUser method is the sole gRPC endpoint exposed by the User Service, defined in the Protocol Buffers file as:

```
rpc CheckUser(Request) returns (Result);
```

When invoked, it receives a Request message containing the username and password fields. Inside the method, the server performs a deterministic transformation of the submitted password using the SHA-256 cryptographic hash function to ensure secure storage and comparison. This is implemented in Go as follows:

```
sum := sha256.Sum256([]byte(req.Password))
pass := fmt.Sprintf("%x", sum)
```

The hash is converted to a hexadecimal string to match the format stored in the database. The service then retrieves the correct hashed password for the requested user from its in-memory map that was populated during initialization from MongoDB. If the username is found and the hashes match, the Correct field of the response is set to true, otherwise it remains false:

```
if true_pass, found := s.users[req.Username]; found {
```

```
    res.Correct = pass == true_pass  
}
```

Chapter 6

OpenWhisk Go Runtime

6.1	OpenWhisk Go Runtime	45
6.2	Example of Action provided by OpenWhisk Go Runtime Repository.....	47

After conducting deeper investigation, it came to our understanding that OpenWhisk does not compile nor does it accept programs compiled using the default program compiler commonly utilized for regular go applications, but instead performs compilation of the action program by parsing it through a customized go runtime specifically designed for OpenWhisk actions. This chapter further explains how the runtime functions and its utility in our research.

6.1 OpenWhisk Go Runtime

This customized runtime acts as an invocation driver, using a lightweight controller that handles function calls through a standard protocol called ActionLoop. Unlike traditional Go execution where the application controls the main event loop and server lifecycle, OpenWhisk's Go runtime shifts this responsibility to the platform's internal handler, which acts as an execution shell that invokes a user-defined function, equivalent to what would be the program's main function, in response to JSON-formatted input events. This ActionLoop model, followed by the platform enforces a clear separation of computation and transport: OpenWhisk manages invocation routing, environment setup, and container lifecycle, while the user's core function is responsible solely for processing input and returning a structured output.

The user-defined action invoked by the shell must follow a strict signature structure, including having a user given name that must start with an uppercase letter and identical input parameter and return types in the form of a `map[string]interface{}`, for example:

```
func Main(args map[string]interface{}) map[string]interface{ }
```

This signature reflects the platform's expectation of a JSON-in / JSON-out interaction model. The parameter `args` represents the input payload, which is automatically populated by the OpenWhisk runtime by parsing the incoming invocation request as a flat JSON object. The `map[string]interface{} type` in Go allows the function to handle arbitrary key-value pairs, supporting a wide variety of payload schemas without requiring a strict predefined structure. The return value of the `Main` function is also a `map[string]interface{}`, which is serialized to JSON and emitted to standard output by the runtime. This response becomes the activation result, which is sent back to the caller and stored in the platform's activation logs. The response must be a well-formed JSON otherwise, the platform may consider the invocation failed.

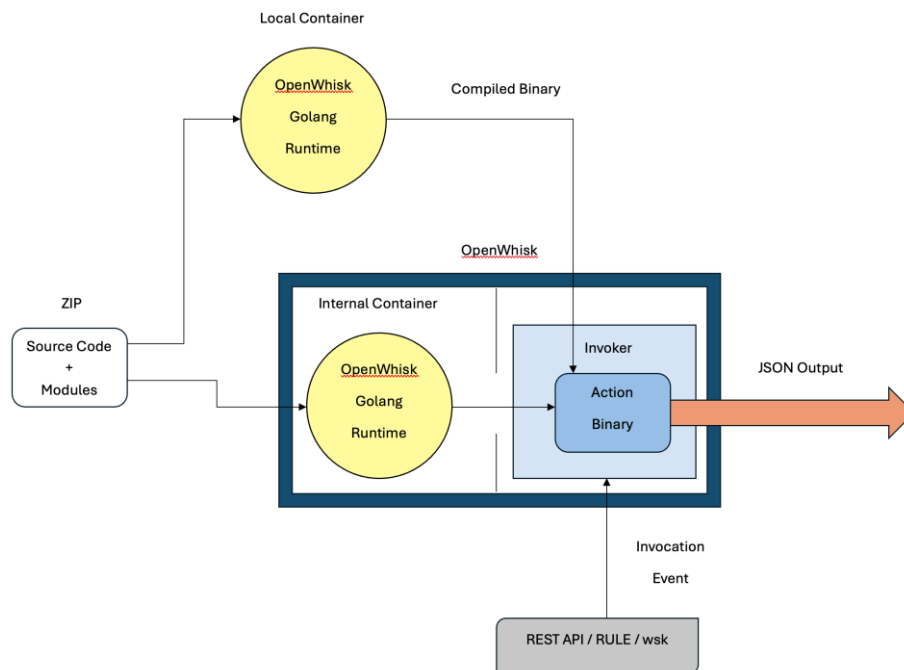


Figure 6.1: Figure demonstrating action creation compilation-creation-invocation workflow

The runtime is bundled into a minimal Docker image that serves as the execution context for all Go actions. The image containing the runtime can be momentarily deployed as a Docker container, for directly handling compilation of the program locally into the proper binary form, before into feeding into the platform for invocation which can be executed in a single command as so:

```
docker run -i action-golang-v1.17:nightly -compile Main < user_v1.go
> exec
```

The runtime supports two primary modes of action deployment: precompiled binary actions and source-based actions. In the first case, the developer compiles their Go function into a

binary before deployment using an instance of the runtime container itself locally and packages it into a ZIP archive. In the second, the developer provides the Go source code and module files, and OpenWhisk compiles the code within the runtime container at execution time. This flexibility allows developers to either rely on OpenWhisk’s internal build system or retain full control over the build process themselves, depending on their development workflow or the specific needs of their application. In both cases, the deployed package must conform to the runtime’s structural and naming conventions for successful execution.

In addition, the runtime environment imposes certain expectations during execution. It initializes containers using a minimal base image where the compiled binary must exist at the container root and is demanded to be statically compiled for Linux amd64. This binary must be named exactly “*exec*”, as this name is hardcoded in the container entrypoint script that the ActionLoop wrapper uses to launch the function. Through failures observed during the experimentation, it was determined that a failure to name the binary *exec* or a mismatch in architecture leads to silent container crashes or cryptic errors.

6.2 Example of Action provided by OpenWhisk Go Runtime Repository

By reviewing the `openwhisk-runtime-go` repository [22], complete examples of functional and more importantly comprehensible action programs were found with additional guidelines to their correct deployment to OpenWhisk. Significantly insightful for the user action service was the example showcasing the appropriate program structure and deployment method for an action program that required external go modules to function called “*module-main*”. In the example, the modules would have to be downloaded and integrated to the compilation process by utilizing the `go.mod` and `go.sum` files, just as they are used in the default building method of a traditional go program and packaged properly when sent to the runtime for compilation.

One of the core lessons derived from the *module-main* example is the support for zip-based action uploads in OpenWhisk, which allows us to submit either compiled executables or raw Go source files in a ZIP archive. Unlike trivial OpenWhisk actions which consist of a single, dependency-free source file, the *module-main* example showcased the end-to-end process of bundling and compiling a multi-file Go project with module dependencies and uploading it as either source or binary in ZIP format.

The example in question, other than the Go module files needed for their inclusion in the compilation process, included the main.go file which implemented the main logic of the action and a Makefile for automating frequently utilized commands. The main.go file in the module-main example provides a complete Go action that demonstrates how to implement a modular, stateless function suitable for OpenWhisk deployment. Its structure follows the requirements imposed by the actionloop protocol. The correctly defined OpenWhisk Go action is structured as so:

```
import (
    "github.com/rs/zerolog"
    "github.com/rs/zerolog/log"
)

func init() {
    zerolog.TimeFieldFormat = ""
}

// Main function for the action
func Main(obj map[string]interface{}) map[string]interface{} {
    name, ok := obj["name"].(string)
    if !ok {
        name = "world"
    }
    log.Debug().Str("name", name).Msg("Hello")
    msg := make(map[string]interface{})
    msg["module-main"] = "Hello, " + name + "!"
    return msg
}
```

This the expected structure of an OpenWhisk Go action, particularly highlighting the role of the init() block, the required function signature, and the output format. The init() function is used here to configure the behavior of the imported library before any action logic runs. The core logic resides in the Main function, which is required to have the specific signature structure mentioned before. This structure defines a generic, JSON-compatible interface both for input and output, in line with OpenWhisk's actionloop protocol. The function reads a value from the input map, validates its type as a string, and constructs a response stored in a new map[string]interface{}. This output map is then returned directly to the invoker, demonstrating the required practice of returning results as JSON-encoded key-value pairs

The accompanying Makefile in the module-main example played a crucial role in automating and simplifying the deployment process of the Go-based OpenWhisk action. It encapsulates a complete workflow that includes packaging the source files, compiling the binary using the official OpenWhisk Go runtime container, and deploying the resulting action to the platform using the wsk CLI tool. By abstracting these steps into a single, repeatable command sequence, the Makefile eliminates manual overhead, ensures consistent configuration, and reduces the potential for user error during action creation and testing. This automated approach proved highly valuable in the development of this thesis, both for deploying converted services and for understanding the structural expectations of the runtime environment.

Chapter 7

Methodology of Conversion

7.1	Identifying and Isolating the Target Service Logic	50
7.2	Ensuring Statelessness.....	51
7.3	Refactoring into the OpenWhisk Runtime Model	51
7.4	Managing External Dependencies	52
7.5	Deployment.....	52

This chapter formalizes the practical process developed to convert individual microservices within a larger service-oriented application into OpenWhisk-compatible serverless functions. The methodology is distilled from iterative experimentation, the successful transformation of key services, and lessons learned from debugging, deployment, and integration. It is structured to serve as a repeatable guideline for future transformations, ensuring compatibility with OpenWhisk’s runtime expectations while preserving the original service logic and external dependencies.

7.1 Identifying and Isolating the Target Service Logic

The conversion process begins with identifying the target microservice’s to be converted core service logic. This requires a focused analysis of the service’s codebase typically consisting of the request processing, business rules, and data interaction components. This logic must be isolated from unrelated elements such as long-running server loops, or tightly coupled middleware. The goal is to identify the minimum executable subset of code that can process a well-defined input and produce a corresponding output, ideally in a stateless manner. This extracted logic is then restructured into a standalone function that conforms to OpenWhisk’s expectations.

Determining this minimum executable subset is often an empirical and iterative process. Apart from the code's structure its runtime behavior requires examination to ensure that all dependencies, such as helper functions and external service calls, are correctly recognized. This may involve trial-and-error testing, where different combinations of code segments are isolated and evaluated for correctness and completeness. The validity of the isolated logic is typically confirmed by comparing its output against the original microservice under a variety of input scenarios, ensuring functional equivalence.

7.2 Ensuring Statelessness

To comply with the serverless execution model required by OpenWhisk, all converted actions must operate in a stateless manner. This means that no data, state, or execution context can persist between action invocations. During the transformation process, it is essential to identify and eliminate any stateful constructs within the original service code, such as in-memory data structures or long-lived resource handlers. These patterns must be avoided, as the serverless environment instantiates a fresh execution context for each request, with no guarantee of continuity.

To maintain functionality without relying on stateful behavior, all data that must persist or be shared across invocations should be managed through external systems. Databases and caching services should be used to store data and information that would otherwise have been held in memory. The action must retrieve all necessary input at the beginning of execution and complete its processing using only the data it receives or fetches from persistent sources. By designing the action in this way, it remains compatible with the serverless platform's guarantees of stateless behavior.

7.3 Refactoring into the OpenWhisk Runtime Model

Once isolated, the service logic is rewritten to follow the required function signature:

```
func FunctionName(args map[string]interface{})map[string]interface{}
```

This form is dictated by OpenWhisk's Go runtime, which uses the actionloop protocol. The logic inside the function must handle all input validation, type assertions, and data

transformations inline. External service interactions must also occur within this single function call or through reusable global initializations, such as inside `init()`. Any communication mechanisms that rely on tightly coupled RPC protocols must be removed or bypassed, as OpenWhisk actions are isolated units that cannot directly participate in non-HTTP communication patterns. Instead, all data required by the function should be delivered through the input JSON map, which acts as the sole interface between the invoker and the action. The function must terminate quickly, avoid background processes, and return all meaningful output in a JSON-compatible map to ensure proper execution and integration within the OpenWhisk runtime.

7.4 Managing External Dependencies

In cases where the service requires external dependencies or additional modules such as caching or database drivers, the action in our case is developed using Go modules through the configuration of `go.mod` and `go.sum` files. The action's packaging process must include these modules, either through source-based deployment, where OpenWhisk compiles the code inside a container or binary deployment, where the action is compiled externally using a Docker image and then uploaded. In both methods, the packaging of the module files is executed through bundling them into ZIP archive files ensuring that all necessary components are transferred together in a format compatible with OpenWhisk's action creation process.

7.5 Deployment

After packaging the action files appropriately, the final step in the transformation process is deployment to the OpenWhisk platform. This involves registering the action with the platform's control interface, specifying the required runtime, and supplying either the compiled binary or source archive. Each action is associated with a name and optionally a package and must declare the correct entry point corresponding to the function name implemented in the source. Once deployed, the action becomes fully accessible within the OpenWhisk environment and can be invoked directly through CLI commands, API calls, or as part of a larger event-driven workflow defined within the system. The success of this process confirms that the action is correctly structured, runtime-compliant, and ready for integration within the broader serverless application.

Overall, we outline a structured methodology for transforming microservice components into OpenWhisk-compatible serverless actions. It covers isolating service logic, establishing statelessness by externalizing stateful features, adapting it to the runtime execution model, and managing external dependencies in a platform-compliant manner. By emphasizing modularity, statelessness, and clear communication through JSON-based input and output, the methodology enables reliable deployment of serverless functions that integrate seamlessly into distributed application architectures. These principles form the foundation for broader service transformation efforts and enable subsequent integration and evaluation within a real-world, cloud-native environment.

Chapter 8

Conversion of User Service

8.1	User Service Conversion Attempt.....	54
8.1.1	Core Function Signature.....	57
8.1.2	Database Logic	57
8.1.3	Authentication Logic	57
8.1.4	Deployment Attempt	58

8.1 User Service Conversion Attempt

Following the detailed insights obtained from the module-main example in the official openwhisk-runtime-go repository, the conversion of the original User microservice into a fully compliant OpenWhisk action was carried out with structural and functional adjustments to meet the platform's runtime expectations. Additionally, as established when analyzing the service and indicated by the gRPC protocol buffer method exposure, the only substantial functionality that is available to other service entities is implemented in the CheckUser method which performs basic user validation. Therefore, the action developed must implement functionally equivalent logic to the one present in CheckUser function. The action was developed in the following structure:

```
import (  
    "context"  
    "crypto/sha256"  
    "fmt"  
    "log"  
    "time"  
  
    "go.mongodb.org/mongo-driver/bson"  
    "go.mongodb.org/mongo-driver/mongo"  
    "go.mongodb.org/mongo-driver/mongo/options"  
)
```



```

// MongoDB connection
var mongoClient *mongo.Client

func init() {
    // MongoDB URI for connection
    mongoURI := "mongodb://mongodb-user-
hotelres.ianton04.svc.cluster.local:27023"
    clientOptions := options.Client().ApplyURI(mongoURI)

    ctx, cancel := context.WithTimeout(context.Background(),
10*time.Second)
    defer cancel()

    client, err := mongo.Connect(ctx, clientOptions)
    if err != nil {
        log.Fatalf("Failed to connect to MongoDB: %v", err)
    }
    mongoClient = client
    log.Println("Connected to MongoDB")
}

// OpenWhisk entry point
func User(event map[string]interface{}) map[string]interface{} {
    // Validate input parameters
    username, ok := event["username"].(string)
    if !ok || username == "" {
        return map[string]interface{}{
            "error": "Invalid or missing username",
        }
    }

    password, ok := event["password"].(string)
    if !ok || password == "" {
        return map[string]interface{}{
            "error": "Invalid or missing password",
        }
    }

    // Query MongoDB for the user

```

```

    collection := mongoClient.Database("user-
db").Collection("user")
    filter := bson.M{"username": username}

    var result struct {
        Username string `bson:"username"`
        Password string `bson:"password"`
    }

    err := collection.FindOne(context.TODO(),
filter).Decode(&result)

    // Handle user not found
    if err == mongo.ErrNoDocuments {
        return map[string]interface{}{
            "status": "success",
            "message": "User not found",
            "valid": false,
        }
    } else if err != nil {
        // Handle other database errors
        return map[string]interface{}{
            "error": fmt.Sprintf("Database error: %v", err),
        }
    }

    // Hash the input password using SHA-256
    hashedInputPassword := sha256.Sum256([]byte(password))
    hashedPassword := fmt.Sprintf("%x", hashedInputPassword)

    // Compare hashed passwords
    passwordIsValid := hashedPassword == result.Password

    return map[string]interface{}{
        "status": "success",
        "message": "User validation complete",
        "username": username,
        "valid": passwordIsValid,
        "hashedPassword": result.Password,
    }

```

}

8.1.1 Core Function Signature

The `user.go` file was designed around the required functional interface `func User(map[string]interface{}) map[string]interface{}`, replacing the original gRPC-based handler models with OpenWhisk's actionloop-compatible signature. The action receives invocation data as a flat JSON object, which is unmarshalled into a `map[string]interface{}` by the runtime and passed to the function. Internally, the function performs type-checking on the "username" and "password" parameters and ensures fault-tolerant input parsing by returning structured error responses when input validation fails.

8.1.2 Database Logic

The database query logic was retained from the original service but refactored to execute entirely within the function scope, using a globally initialized MongoDB client defined in the `init()` function. This approach aligns with OpenWhisk's container reuse model, allowing the database connection to persist across multiple invocations, within the time frame the action container is alive before initial invocation, that being about 10-15 minutes. The `init()` block itself was constructed to safely establish a connection to the MongoDB service using a Kubernetes service URI `mongodb-user-hotelres.ianton04.svc.cluster.local:27023`. The MongoDB Go driver `mongo.Connect` is used in conjunction with BSON decoding to retrieve user credentials from the "user" collection within the "user-db" database.

Sufficient error handling is conducted regarding the unsuccessful retrieval of user information from the database storage, or any other kind of errors produced by the Mongo service. Proper message communication from the action is integral for understanding the state of the action at any point of interaction to ensure stability.

8.1.3 Authentication Logic

Passwords are verified using a SHA-256 hashing function and compared to the stored hash, with the result returned in a JSON-encoded response that abides to OpenWhisk's output model. In the response we include primarily status and message keys validating the complete execution of user validation task as well as a Boolean key value called "valid" that states the result of validation, true for password match and false for mismatch.

Notably, all state transitions, error handling, and logic branching are encapsulated within the single `User` function, respecting OpenWhisk's strict lifecycle boundaries and stateless invocation model.

Accompanying the new refined version of the converted action, we also constructed a Makefile coupled with the action program for proper compilation and deployment heavily influenced by the structure of the module-main example and adapted accordingly to reflect the new action name, package, and compilation needs.

8.1.4 Deployment Attempt

Deploying the converted `User` action can now be executed through the specialized Makefile responsible for handling environment configuration for an automated process. By simply running in sequence the appropriate commands for deployment and testing we can easily execute a process that is preconfigured to locally compile the `user.go` program containing the action, using the `action-golang` runtime from a temporary docker container. Afterwards, the action in the form of a packaged zip file containing the compiled action binary as well as its required modules is deployed to the OpenWhisk platform and then is invoked after proper action creation. These tasks implemented in the Makefile are done through the `wsk` CLI tool and are configured to be responsive and streamlined in terms of command feedback and output structure.

Initially, running the deployment command returns positive results regarding the creation of the service within the OpenWhisk platform environment. As evident in the figure 4.9 the expected sequence of tasks outlined by the Makefile rules interchange successfully. Initially the appropriate package is created within OpenWhisk to host the upcoming action to be created and the files demanded for compilation are packaged in a zip file that is then loaded to the activated docker container running the OpenWhisk go runtime. Lastly the zipped output from

the runtime containing the successfully compiled binary is used along with its demanded modules to create an action named checkUser.

```
ianton04@kubel1:~/thesis-serverless/testServerlessUser/forBuild$ make deploy
wsk -i package update user-service
ok: updated package user-service
touch package.done
zip User-src.zip -qr user.go go.mod go.sum
sudo docker run -i openwhisk/action-golang-v1.17:nightly -compile User <User-src.zip >User-bin.zip
wsk -i action update user-service/checkUser User-bin.zip --main User --docker openwhisk/action-golang-v1.17:nightly
ok: updated action user-service/checkUser
```

Figure 4.9: OpenWhisk output when creating action using Makefile command “make deploy”

These results indicate a smooth execution of the commands outlined in the Makefile ruling as well as positive feedback from OpenWhisk relating to successfully creating the action, however this output is not sufficient to ensure proper compatibility of the action program with the platform’s execution environment. We then run the command for executing an invocation of the checkUser action with a predefined test sample of input. By doing so, as showcased in figure 4.10 we achieve successful invocation of the action with correctly returned results. The action delivers a properly structured message and status value informing us about the completion of the user validation process as well as information regarding the actual validity of the data provided, evident in the value of the “valid” key.

The “hashedPassword” and “username” keys present were added for debugging purposes after encountering an issue with validating expected username and password values stored in the mongo database, which was discovered to simply be misconception of the way the username and password were paired when uploaded to mongo.

```
wsk -i action invoke user-service/checkUser -P test.json -r | tee -a test.out
{
  "hashedPassword": "84d9c4b849506b6d8f8075a9000e7e0a254be71060ea889fad3c88395988f4fc",
  "message": "User validation complete",
  "status": "success",
  "username": "Cornell_30",
  "valid": true
}
```

Figure 4.10: Successful action invocation result of checkUser action

The successful response confirms that the compiled and deployed Go action not only adheres to OpenWhisk’s structural and runtime constraints but also performs the intended application logic with precision. These results collectively demonstrate the end-to-end compatibility and correctness of the transformed microservice action and validate the underlying architectural and implementation decisions that guided its construction.

Chapter 9

Integration of Action with Microservice Application

9.1	GPRC Invocation of Original User Service	61
9.2	The checkUserHTTP function.....	62
9.3	Building new Image of the Application	65

This chapter showcases the process followed to properly integrate the newly created serverless function implemented as an OpenWhisk action, to the already functioning environment of the Hotel Reservation microservice application. The communication method with OpenWhisk is constructed to seamlessly invoke the action at any point demanded by the application. Additionally, a new functional image of the application is built for the purpose of integrating additions of code responsible for application to action communication.

With the conversion of the individual User Service to a serverless OpenWhisk action complete and verified for correctness, the next stage involves integrating this new component into the existing HotelReservation application. This process requires carefully refactoring service invocations to align with the OpenWhisk model while maintaining full compatibility with the frontend and user-facing APIs.

The original structure of the Hotel Reservation application calls upon the CheckUser function which was converted into a serverless action to execute user validation, only from within the frontend service. The server.go file containing application logic of the Frontend service makes the call to the User service in every instance of handler function that is called when specifically invoking the usage of one of the services by the applications API. Essentially, at every request received by the frontend service, requiring the utilization of another service from the collection of business logic services, the appropriate handler function is called upon which that itself internally communicates with the User service to perform user validation.

9.1 GPRC Invocation of Original User Service

The original CheckUser method is called upon from only one other service from the collection of services active when the application is deployed, that being the Frontend service specifically present within Frontend's server.go file. The method call can be traced to all handler functions implemented for all the main services in the server file which is done so to always validate the user that invokes the functionalities provided by each service. This indicates that the action to be developed will have to be invoked equally at the same points the function was called upon, thus exposing the route to seamlessly incorporate the serverless function within the already existing microservice implementation constructing a hybrid system between the two architectures.

The body of the request received by Frontend in every case contains username and password values that are properly parsed and extracted from the request payload and fed into a request to the User Service. The communication between the services is made possible by the gRPC remote procedure call protocol that all services of the application adhere to, to maintain a uniform method of exchanging information and utilizing each services functions making them available to one-another. An example of this can be seen present in the reviewHandler function responsible for handling user review requests in the frontend service and includes a user authentication step utilizing gRPC:

```
username, password := r.URL.Query().Get("username"),
r.URL.Query().Get("password")
    if username == "" || password == "" {
        http.Error(w, "Please specify username and password",
http.StatusBadRequest)
        return
    }

    // Check username and password
    recResp, err := s.userClient.CheckUser(ctx, &user.Request{
        Username: username,
        Password: password,
    })
    if err != nil {
```

```

        http.Error(w, err.Error(),
http.StatusInternalServerError)
        return
    }

    str := "Logged-in successfully!"
    if recResp.Correct == false {
        str = "Failed. Please check your username and password. "
    }

```

It begins by parsing the username and password parameters directly from the request URL. This direct extraction from the query string ensures that both credentials are available for the subsequent verification process. An initial check confirms that neither value is empty and if the parameters are valid, the handler proceeds to authenticate the user via a gRPC request, invoking the `CheckUser` method from the user microservice through the pre-initialized `userClient`.

This call encapsulates the provided credentials into a `user.Request` protobuf message and dispatches it using gRPC. The response returned, held in `recResp`, includes a `Correct` boolean field indicating the result of the authentication. If the credentials are invalid or the gRPC call fails, an HTTP error is returned to the client.

9.2 The `checkUserHTTP` function

The original user service uses gRPC for communication with other services, exposing internal methods through protocol buffers. While this architecture simplifies cross-service interaction within a microservice ecosystem, it poses compatibility issues when converting services into OpenWhisk actions, which are invoked via HTTP-based REST APIs. To enable integration of the OpenWhisk-based `checkUser` action within the existing application, the gRPC-based authentication mechanism was replaced with an HTTP invocation method. This transition required decoupling the frontend service from the gRPC interface of the original user microservice and introducing a new function, `checkUserHTTP`, responsible for sending requests to the serverless action endpoint as well as parsing the json response sent by the action and returning it to the application:

```

func checkUserHTTP(username, password string) (bool, error) {

```



```

url :=
"https://10.10.1.1:31001/api/v1/namespaces/guest/actions/user-
service/checkUser?blocking=true&result=true"

// Create JSON payload
reqBody, _ := json.Marshal(map[string]string{
    "username": username,
    "password": password, // Send as plain string (no local
hashing)
})

// Create HTTP request
req, _ := http.NewRequest("POST", url, bytes.NewBuffer(reqBody))
req.Header.Set("Content-Type", "application/json")
req.SetBasicAuth("23bc46b1-71f6-4ed5-8c54-816aa4f8c502",
"123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP")
// Replace with actual auth

// Create custom HTTP client to ignore SSL certificate
verification
tr := &http.Transport{
    TLSClientConfig: &tlsGo.Config{InsecureSkipVerify: true},
}
client := &http.Client{Transport: tr}

// Send HTTP request
resp, err := client.Do(req)
if err != nil {
    return false, err
}
defer resp.Body.Close()

// Read the raw response body
body, err := io.ReadAll(resp.Body)
if err != nil {
    return false, err
}

//Debug: Print Full JSON Response
fmt.Println("Full JSON Response:", string(body))

```

```

// Define a struct to parse the JSON response
var jsonResponse struct {
    HashedPassword string `json:"hashedPassword"`
    Message         string `json:"message"`
    Status          string `json:"status"`
    Username        string `json:"username"`
    Valid          bool   `json:"valid"`
}

// Parse JSON into the struct
err = json.Unmarshal(body, &jsonResponse)
if err != nil {
    return false, fmt.Errorf("failed to parse JSON response:
%v", err)
}

// Print extracted values for debugging
fmt.Printf("Extracted - Username: %s | Valid: %v | Message: %s |
Hashed Password: %s\n",
    jsonResponse.Username, jsonResponse.Valid,
    jsonResponse.Message, jsonResponse.HashedPassword)

// Return valid status and hashed password
return jsonResponse.Valid, nil
}

```

This change enables seamless access to the action from the frontend service, aligning the hybrid system with serverless invocation semantics.

The function begins by defining the target URL which includes OpenWhisk's RESTful API gateway path. A JSON request body is constructed using the provided credentials. The request body is encoded in JSON and includes the username and password fields, which are marshaled into a format expected by the deployed action.

A custom HTTP client is configured to bypass SSL verification and basic authentication credentials to specify authentication host and key demanded by the OpenWhisk endpoint are added to authorize the call. To accommodate the self-signed certificate used in the local

environment, a custom HTTP transport is created with InsecureSkipVerify enabled, and the request is executed using a modified HTTP client. Upon receiving the response, the function reads the raw response body and unmarshals it into a temporary struct with expected fields such as Username, HashedPassword, Status, Message, and Valid:

Finally, the function returns the Valid field to indicate the success or failure of user validation. This implementation ensures full compatibility with OpenWhisk's expected input/output model and enables the frontend service to validate credentials through an external serverless function without relying on a persistent RPC channel. The integration was completed by substituting previous `userClient.CheckUser` gRPC calls existent in the service handler functions with synchronous calls to `checkUserHTTP`, enabling the frontend to maintain identical business logic while shifting its user authentication layer to a serverless environment.

9.3 Building new Image of the Application

Following the integration of the new HTTP-based user authentication logic via the `checkUserHTTP` function, the frontend service required recompilation to include the modified `server.go` implementation. To accomplish this, a new Docker image of the microservice application was constructed. The Dockerfile used for this process compiled the Go-based application binary with the updated logic and produced a statically linked executable. The resulting image was tagged as `ianton04/hotel-reservation:latest` and pushed to Docker Hub to make it available for deployment within the Kubernetes cluster.

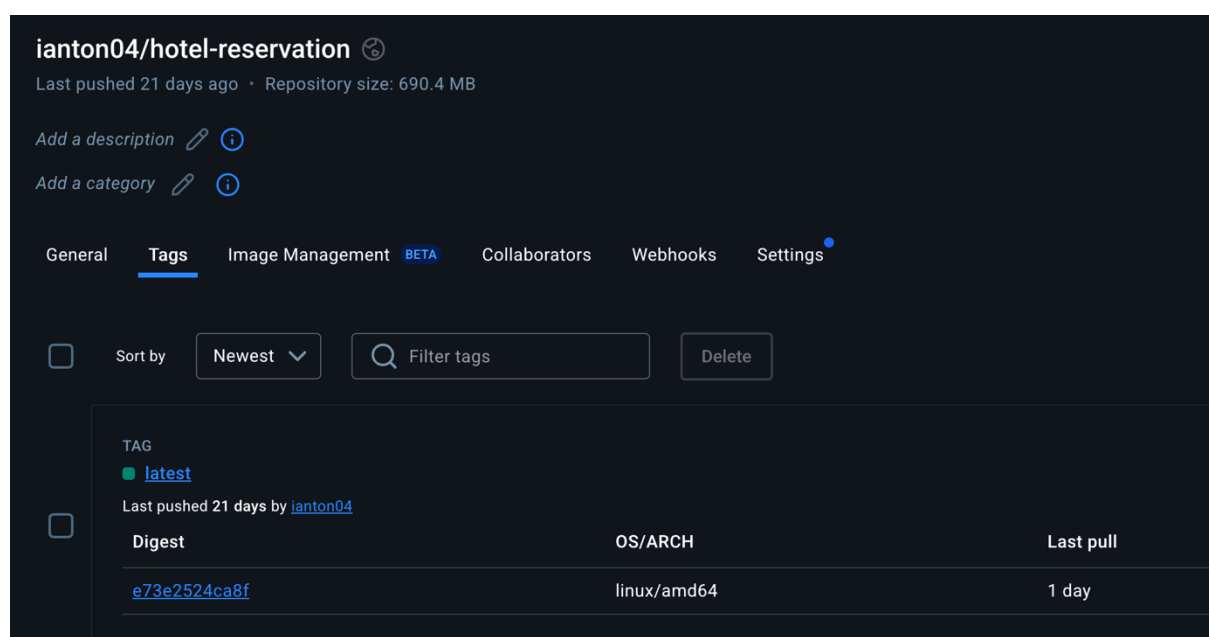


Figure 6.1: Newly Built Application image uploaded to Dockerhub under the name ianton04/hotel-reservation with the tag latest

To ensure that the Helm deployment reflected this new version of the application, the `values.yaml` file located in the `frontend` chart directory of the Helm chart (`hotelReservation/helm-chart/hotelreservation/charts/frontend/`) was modified accordingly. Compared to the original version of the file, which referenced the default `deathstarbench/hotel-reservation` image, the new configuration specified the updated image registry, tag, and pull policy:

```
image: ianton04/hotel-reservation
tag: latest
imagePullPolicy: Always
```

This ensured that Helm would always pull the most recent image from the Docker registry at deployment time. Upon invoking Helm to redeploy the application (`helm install` or `helm upgrade` depending on the cluster state), Kubernetes fetched and deployed the new image to the cluster. The updated frontend pod now included the newly implemented logic that redirected user verification requests to the OpenWhisk action endpoint instead of the original gRPC user service.

To ensure the correct re-deployment of the application, now containing the logic added to the `frontend server.go` file by fetching the new image we built for this purpose, we query the Frontend service using the application api to access the user handler. An HTTP POST request is constructed directed to the frontend service pod IP address and the appropriate port allocated for it in the application's architecture. The request also carries username and password values to be utilized for validation within the action itself. Here we see a request containing valid and invalid user validation data:

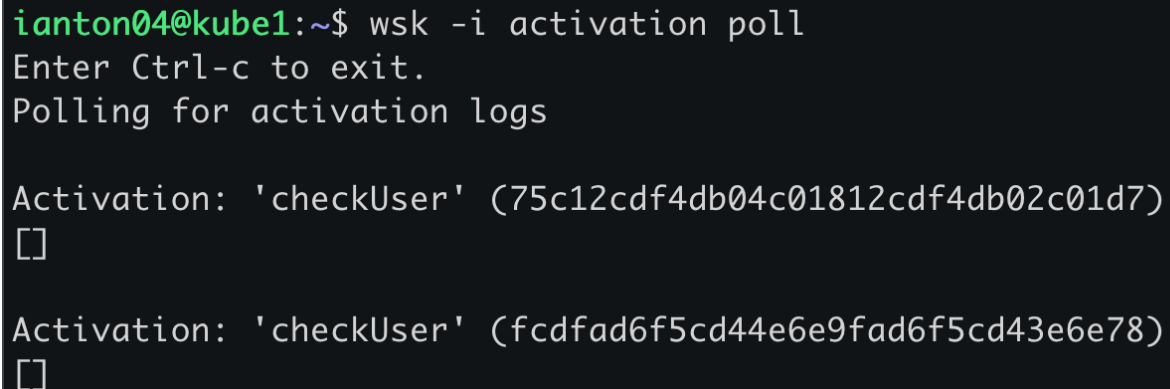
```
ianton04@kubel1:~$ curl -X POST "http://10.108.209.74:5000/user?username=Cornell_30&password=0000000000"
{"message":"Login successfully!"}
ianton04@kubel1:~$ curl -X POST "http://10.108.209.74:5000/user?username=Cornell_30&password=0000000000"
{"message":"Failed. Please check your username and password. "}
```

Figure 6.2: HTTP POST requests to the Frontend Service instructed to communicate with the user action through calling the `userHandler` function with both valid and invalid user data

Using the OpenWhisk command from its CLI tool that logs the action activations in real time:

```
wsk -i activation poll
```

We can ensure the action is correctly invoked through the microservice application by making the post request after seeing the list of action activations being logged live as we send it.



```
ianton04@kube1:~$ wsk -i activation poll
Enter Ctrl-c to exit.
Polling for activation logs

Activation: 'checkUser' (75c12cdf4db04c01812cdf4db02c01d7)
[]

Activation: 'checkUser' (fcdfad6f5cd44e6e9fad6f5cd43e6e78)
[]
```

Figure 6.3: List of checkUser action activations recorded after making requests to the action through the Frontend Service

The integration of the OpenWhisk-based serverless function into the existing microservice application demonstrates the viability of combining traditional service architectures with event-driven execution models. By replacing the gRPC-based user verification flow with an HTTP-based invocation of a stateless action, the application maintains its functional requirements while benefiting from the scalability and modularity of serverless infrastructure. The updated deployment workflow, supported through image versioning and Helm chart adjustments, ensures seamless incorporation of new logic into the Kubernetes-managed environment.

Chapter 10

Conversion of Reservation Service

10.1	Reservation Service Conversion.....	68
10.2	Reservation Service Integration	73
10.3	Reservation Service Invocation.....	73

This chapter describes the conversion of another service belonging in the Hotel Reservation Suite, the Reservation service. After designing a robust methodology of converting microservices to actions and also experimenting in applying through converting the User Service, we chose to implement it on a different service with non-trivial service-specific logic. The User service, as mentioned, follows a fairly simple implementation executing a trivial task which was beneficial for understanding the service and approaching initial steps of the conversion process. However, in order to ensure universal application of the conversion methods constructed, a more logically intense service was put to test.

10.1 Reservation Service Conversion

Reservation involves multiple database interactions, cache validation and updates, and conditional failure paths based on room availability. Due to its transactional nature and role in the core functionality of the hotel reservation system, converting it into a serverless-compatible function was essential for testing our transformation workflow for applicability and versatility.

The converted action, named `makeReservation`, follows the same methodology used in previous service transformations and is implemented as follows:

```
func Makereservation(event map[string]interface{})
map[string]interface{} {
    customerName, ok1 := event["customerName"].(string)
    hotelId, ok2 := event["hotelId"].(string)
    inDate, ok3 := event["inDate"].(string)
```

```

    outDate, ok4 := event["outDate"].(string)
    roomNumber, ok5 := event["roomNumber"].(float64) // JSON numbers
are float64

    if !ok1 || !ok2 || !ok3 || !ok4 || !ok5 {
        return map[string]interface{}{
            "error": "Invalid or missing parameters",
        }
    }

    reservationCollection := mongoClient.Database("reservation-
db").Collection("reservation")
    numberCollection := mongoClient.Database("reservation-
db").Collection("number")

    // **Convert dates**
    inDateTime, _ := time.Parse("2006-01-02", inDate)
    outDateTime, _ := time.Parse("2006-01-02", outDate)
    indate := inDateTime.Format("2006-01-02")

    memc_date_num_map := make(map[string]int)

    for inDateTime.Before(outDateTime) {
        count := 0
        inDateTime = inDateTime.AddDate(0, 0, 1)
        outdate := inDateTime.Format("2006-01-02")

        // **Memcached Check**
        memcKey := fmt.Sprintf("%s_%s_%s", hotelId, indate, outdate)
        item, err := memcClient.Get(memcKey)
        if err == nil {
            count, _ = strconv.Atoi(string(item.Value))
            memc_date_num_map[memcKey] = count + int(roomNumber)
        } else if err == memcache.ErrCacheMiss {
            // **Check MongoDB**
            filter := bson.M{"hotelId": hotelId, "inDate": indate,
"outDate": outdate}
            var reserve []struct {
                Number int `bson:"number"`
            }

```

```

        cur, err := reservationCollection.Find(context.TODO(),
filter)

        if err != nil {
            log.Printf("Failed to get reservation data: %v",
err)

            return map[string]interface{}{
                "error": "Database error",
            }
        }
        defer cur.Close(context.TODO())
        cur.All(context.TODO(), &reserve)

        for _, r := range reserve {
            count += r.Number
        }
        memc_date_num_map[memcKey] = count + int(roomNumber)
    } else {
        return map[string]interface{}{
            "error": fmt.Sprintf("Memcached error: %s", err),
        }
    }
}

// **Check Hotel Capacity**
memcCapKey := hotelId + "_cap"
item, err = memcClient.Get(memcCapKey)
hotelCap := 0
if err == nil {
    hotelCap, _ = strconv.Atoi(string(item.Value))
} else if err == memcache.ErrCacheMiss {
    var num struct {
        Number int `bson:"numberOfRoom"`
    }
    err = numberCollection.FindOne(context.TODO(),
bson.M{"hotelId": hotelId}).Decode(&num)
    if err != nil {
        return map[string]interface{}{
            "error": fmt.Sprintf("Failed to find hotel
capacity: %v", err),
        }
    }
}

```



```

        hotelCap = num.Number
        memcClient.Set(&memcache.Item{Key: memcCapKey, Value:
[]byte(strconv.Itoa(hotelCap))})
    }

    if count+int(roomNumber) > hotelCap {
        return map[string]interface{}{
            "status": "failed",
            "message": "Room unavailable",
            "reason": count+int(roomNumber),
            "count": count,
            "hotelCap": hotelCap,
            "hotelId": hotelId,
        }
    }

    indate = outdate
}

// **Update Memcached**
for key, val := range memc_date_num_map {
    memcClient.Set(&memcache.Item{Key: key, Value:
[]byte(strconv.Itoa(val))})
}

// **Insert Reservation into MongoDB**
inDateTime, _ = time.Parse("2006-01-02", inDate)
indate = inDateTime.Format("2006-01-02")

for inDateTime.Before(outDateTime) {
    inDateTime = inDateTime.AddDate(0, 0, 1)
    outdate := inDateTime.Format("2006-01-02")

    _, err := reservationCollection.InsertOne(context.TODO(),
bson.M{
        "hotelId":      hotelId,
        "customerName": customerName,
        "inDate":        indate,
        "outDate":       outdate,
        "number":        int(roomNumber),

```

```

    })
    if err != nil {
        return map[string]interface{}{
            "error": fmt.Sprintf("Failed to insert reservation:
%v", err),
        }
    }
    indate = outdate
}

return map[string]interface{}{
    "status": "success",
    "message": "Reservation confirmed",
    "hotelId": hotelId,
}
}

```

beginning with isolating the core logic responsible for booking validation. The input interface is restructured into a `map[string]interface{}` format to comply with OpenWhisk's Go runtime expectations. To support this format, all incoming parameters such as the hotel identifier, check-in and check-out dates, customer name, and number of rooms are dynamically parsed from a JSON payload, with appropriate type assertions and validation checks applied to guard against malformed or missing data.

Once the inputs are processed, the action executes a loop over the specified date range, checking room availability for each consecutive day by querying Memcached for previously recorded bookings. If a cache miss occurs, it performs a fallback query to MongoDB to count existing reservations from the database and populate the cache accordingly. It then retrieves the total room capacity for the target hotel and compared it against the cumulative requested bookings to ensure availability. If the request is valid, the action proceeds to update the reservation count in Memcached and insert the finalized reservation records into MongoDB. To maintain performance and minimize redundant operations, global clients for MongoDB and Memcached were initialized within the `init()` function, following the runtime's execution model for warm container reuse:

```

func init() {
    // MongoDB Setup

```

```

    mongoURI := "mongodb://mongodb-reservation-
hotelres.ianton04.svc.cluster.local:27022"
    clientOptions := options.Client().ApplyURI(mongoURI)
    ctx, cancel := context.WithTimeout(context.Background(),
10*time.Second)
    defer cancel()

    client, err := mongo.Connect(ctx, clientOptions)
    if err != nil {
        log.Fatalf("Failed to connect to MongoDB: %v", err)
    }
    mongoClient = client
    log.Println("Connected to MongoDB")

    // Memcached Setup
    memcClient = memcache.New("memcached-reserve-1-
hotelres.ianton04.svc.cluster.local:11214")
    log.Println("Connected to Memcached")
}

```

10.2 Reservation Service Integration

To integrate the newly converted makeReservation OpenWhisk action into the existing application, modifications were applied to the frontend/server.go file. Following the same pattern applied for the User service's case, a new helper function named makeReservationHTTP was defined, encapsulating the logic for invoking the action via a direct HTTP request to the OpenWhisk API endpoint. Similarly to the checkUserHTTP method, this function serializes the required input parameters into a JSON payload, sends the request using a customized HTTP client with basic authentication, and parses the JSON response for further handling. In the reservationHandler function, the original gRPC call to the reservation service was replaced by a call to makeReservationHTTP. To apply these changes in a deployment environment, the container image of the frontend service was rebuilt to include the updated codebase. Upon redeployment via Helm, the cluster loaded the new container image, activating the modified reservation flow through OpenWhisk.

10.3 Reservation Service Invocation

In similar fashion to the process followed for the user service, a Makefile was constructed using the one provided by the “module-main” action creation example, as a template and was made to fit to the specific attributes of the makeReservation action. Utilizing the Makefile we are instantly able to create within OpenWhisk’s environment and invoke to evaluate both functionality and accuracy of test results. By running commands “make deploy” which prompts action creation and “make test” which prompts action invocation using an input test sample for successful reservation:

```
{
  "customerName": "JohnDoe",
  "hotelId": "1",
  "inDate": "2025-03-10",
  "outDate": "2025-03-12",
  "roomNumber": 1
}
```

we receive the following result as action return output, which matches the appropriate output expected by the system in case of success:

```
wsk -i action invoke reservation-service/makeReservation -P test.json -r | tee -a test.out
{
  "hotelId": "2",
  "message": "Reservation confirmed",
  "status": "success"
}
```

Figure 4.10: Successful action invocation result of makeReservation action

Chapter 11

Automation of Conversion Process

11.1	Introduction	75
11.2	Initial Attempts.....	75
11.3	Adopting Generative AI for Code Conversion.....	76
11.4	Tool Architecture and Workflow	76
11.5	Tool Architecture and Workflow	77
11.6	Observations.....	79
11.7	Closing Remarks	79

In this chapter, we examine the feasibility of automating the conversion process of a microservice application program to a serverless OpenWhisk action. We demonstrate the intricacies of the automation process and our solution: a tool leveraging generative artificial intelligence.

11.1 Introduction

As established throughout the course of this research, transforming traditional Go-based microservices into serverless functions compatible with Apache OpenWhisk requires an adequate amount of manual effort. While a consistent methodology was constructed the conversion process remained non-trivial. Each service exhibited subtle semantic differences and specific structuring of internal logic. Factors such as these, prompted significant problems in the development of a rule-based transformation mechanism that would be able to handle a diverse range of services.

11.2 Initial Attempts

Initial efforts toward automation leveraged methods such as Abstract Syntax Tree traversal and source-to-source transformations. The idea was to analyze and isolate specific areas of code that would be either be placed in different function blocks but would primarily remain unchanged in form, for example MongoDB and Memcached connection configuration and client initialization in the `init()` function or be removed completely in cases of gRPC communication logic. However, these approaches quickly reached their limits.

AST analysis proved useful for syntactic disassembly and for characterizing source code structures in preparation for transformation. However, it lacked the semantic understanding necessary to fully comprehend service logic and contextual dependencies. In many cases, services implemented logic that relied on external helper functions, shared resource access, or chained gRPC calls, patterns that may be recognizable at the syntactic level, but difficult to reliably reinterpret or transform using rule-based logic alone. Even in the case of simpler service functions such as `CheckUser`, which exhibit relatively straightforward logic and minimal internal complexity, static code analysis failed to produce generalizable transformation rules. This limitation highlighted the need for a more context-aware and adaptable approach to automation.

11.3 Adopting Generative AI for Code Conversion

Given these challenges, the solution space expanded toward generative artificial intelligence, specifically large language models (LLMs). The capacity of LLMs to infer context needs, refactor code, and generate structured output from descriptive prompts made them a compelling candidate for the task. These models offer semantic understanding and flexibility, features absent in syntactic approaches, which are essential when dealing with services that do not adhere to a uniform template. More importantly, LLMs can use specific context from carefully constructed prompts to shape the output so it matches the required structure of OpenWhisk actions.

11.4 Tool Architecture and Workflow

The tool developed for this purpose is a command-line interface, which accepts as input a YAML configuration file describing important information about the context of conversion, that is utilized to construct an accurate and clear prompt meant to instruct the model on how to

perform the conversion. It also accepts the CheckUser converted action a reference example in order to further enable the model to understand the structure we want to achieve. The CLI invokes an LLM endpoint, in this case: OpenAI's gpt-4-turbo-preview model, and submits the constructed prompt.

The CLI is implemented in Go and organized into modular components that handle configuration parsing, environment variable loading for the API key, prompt construction, LLM interaction, and output parsing. Users invoke the CLI using a simple “convert” command with two required flags: the path to the config file (--config) and the source code to be converted (--input). Internally, the tool reads the YAML configuration which defines critical attributes such as:

- `function_to_convert`: Name of function to be located and converted from the provided input file
- `action_name`: name to assign the main action function to be executed by OpenWhisk
- `dependencies`: supporting services utilized that need to be handled such as MongoDB or Memcached
- `input_parameters`: a list of value pairs that specify the name and type input parameters to prevent inaccuracies
- `output_keys`: an example return structure for the json element to be returned from the action
- `reference_function`: name of the file that contains the converted action to be used as an example
- `notes`: natural language giving additional instructions to be applied to the instruction prompt

The tool assembles a detailed prompt using this metadata, then sends it as a chat completion request to the OpenAI API. The configuration file enables the user to adapt the conversion prompt to any chosen service with ease and enables more accurate service-specific configuration to avoid leaving too many components of the process for the model to speculate and potentially hallucinate on. The LLM response includes the converted action packaged in a go file named `converted_action.go`.

11.5 Tool Architecture and Workflow

To validate the effectiveness of this approach, the MakeReservation microservice was selected

as a benchmark case for automated conversion. This service is non-trivial, offering adequate complexity to test the model's ability to adapt to difficult service-specific logic, multiple dependencies combining MongoDB interactions and cache updates via Memcached. The prompt utilized for this has, apart from additional parameters specified in the configuration file has the following structure:

"You are converting a Go microservice function into an Apache OpenWhisk action.

The goal is to maintain all core logic (e.g. MongoDB operations, Memcached access, input validation, loops, inserts) and ensure the result compiles directly on OpenWhisk with minimal manual modification.

Guidelines:

- Implement all MongoDB and Memcached calls as shown, if there are any.*
- Handle all necessary input validation and error reporting.*
- DO NOT summarize the code or skip logic steps.*
- Use the reference action below to match structure, naming, and initialization logic."*

Additionally, the following notes section was added to the prompt through the configuration file, providing further instructions:

"The output Go action must follow OpenWhisk conventions, using func Function(map[string]interface{}) map[string]interface{}, with the name "Function" being a placeholder for a function name that its first letter must be Uppercase.

Do not include a main function as its not needed in OpenWhisk.

Do not omit error checking, Memcached or MongoDB logic, even if partial or nested.

You must maintain the same logic and structure as the original function, adjusting it to fit the OpenWhisk action format which can be seen from the reference action.

The function should be able to handle the input parameters and return the output keys as specified.

Do not put placeholders in the code, do not use "..." or "TODO" comments, the code must be complete and functional, nothing must be missing.

Compact all MongoDB and Memcached connection logic into a single function called: init(), see example from reference service"

Upon execution, the AI-generated action closely matches the behavior and structure of the original logic. All major functional blocks were preserved: cache reads and fallbacks with Memcached, room availability calculations, iteration over date ranges, and reservation insertions into MongoDB. Moreover, the structure of the result closely followed the reference action, including the correct usage of global initializations in the init() function block, the

OpenWhisk-compliant function signature, and return payload formatting. Comparisons between the manual and AI-generated versions revealed only minor formatting and ordering differences, with functional parity preserved.

This successful outcome highlights the utility of LLMs in handling complex transformations that exceed the capabilities of static tooling. By inserting context effectively and providing structure within the prompt, the tool effectively guided the LLM to produce a usable and overall correct output. An even more refined version of the tool could benefit development greatly accelerating the conversion process and enabling scalable transformation of more services.

11.6 Observations

Through experimenting with different prompt and input structures, it was derived that the tool functions the best when provided with smaller token-wise and more accurate inputs. For example, inserting the entire `server.go` file of the reservation service and having it figure out the function to convert specifically, just because of the added context with irrelevant information, confused the model resulting in incomplete action generations with a lot of placeholders. This fact could become an obstacle with more complex services with dependencies spread throughout the program file. The overall input that is sent to the model is already quite large, including prompt, metadata from the configuration file and additional example action program, therefore aiming to be as concise as possible providing only relevant information was greatly beneficial. This behavior will also prove to be beneficial in terms of cost. Utilizing the OpenAI models through the developer API is not free at any tier, and even though the cost is minimal, input token size is always something to consider.

11.7 Closing Remarks

The development and deployment of the tool demonstrate the feasibility and practical benefits of incorporating generative AI into the automation of serverless action generation. While not yet a fully general-purpose solution for all service types, this approach significantly reduces the manual burden and allows developers to scale their conversion efforts with consistency. Nonetheless, the tool provides a strong foundation for AI-assisted transformation of microservice architectures into function-as-a-service platforms.

Chapter 12

Operational Integrity and Performance Evaluations

12.1	Benchmarking	80
12.2	Evaluation of Operational Integrity	81
12.3	Performance Evaluation	83
12.3.1	Reservation Service	84
12.3.2	User Service	87
12.3.3	Mixed Service Workload	90
12.3.4	Automatically Converted Reservation Service	93
12.4	Conclusion.....	95

In this chapter, we describe the process of evaluating the performance of the hybrid microservice-serverless system that was created with the converted actions being integrated to the rest of the application. The system is evaluated both in terms of effectiveness, producing equal results to the original versions of the services as well as in terms of latency especially at later percentiles approaching the performance expressed by the unchanged application.

12.1 Benchmarking

To evaluate the performance implications of transforming microservices into serverless functions, a structured benchmarking methodology was established using the wrk workload generator. The benchmarking targeted three representative workload profiles: one calling the user service, one calling the newly constructed reservation service, and a synthetic mixed workload which combines calls to multiple services, simulating a more realistic usage pattern within the hotel reservation domain.

The wrk tool was selected for its flexibility in simulating HTTP workloads and its ability to be extended with Lua scripting. For each benchmark scenario, a dedicated Lua script was written to define the request format, target endpoint, and method parameters. These scripts were orchestrated to execute under three predefined workload intensities: light, medium, and high. These load levels were configured as follows:

- Light: 5 concurrent connections at 30 requests per second
- Medium: 15 concurrent connections at 100 requests per second
- High: 30 concurrent connections at 250 requests per second

Each test was executed for a fixed duration of 60 seconds, with the number of threads set to 4.

To ensure a fair and meaningful evaluation, the workload levels used in benchmarking were carefully adjusted to match the system's capabilities. A key constraint was the absence of horizontal pod autoscaling in the microservice deployment, which meant that pushing the system to very high loads would have led to artificial slowdowns unrelated to the architectural differences being studied. As a result, the light, medium, and high load levels were selected based on manual testing and observation. By steadily increasing testing values, these were found to create adequate stress without causing system instability. Additionally, the Hotel Reservation application is part of a research-based benchmark suite not intended for commercial-scale traffic, which further supports the use of modest, but also representative load levels for accurate comparison.

The three tests for each type of load were executed under circumstances reflecting the system's state regarding architecture. Experiments were conducted with the previous scripts having the application's structure maintain its original form as a pure microservice application, as well as two other cases: the structure following the hybrid serverless schema, communicating with the OpenWhisk actions `checkUser` and `makeReservation`, while all action containers are uninitialized, and the container environment is cold and another case where action containers were prewarmed.

12.2 Evaluation of Operational Integrity

The first step in the testing phase was to assess the operational stability of the serverless components and their ability to sustain load when invoked from within the application's

frontend. During these evaluations, however, a critical limitation was encountered. Upon executing medium and high-intensity test scripts, the system consistently returned errors indicating excessive invocation rates as can be seen in Figure 7.1.

```

Extracted - Username: | Valid: false | Message: | Hashed Password:
Full JSON Response: {"code": "YEajUky5wzwa0wsqwfCkVwXtkijpZxw", "error": "Too many requests in the last minute (count: 4973, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
makeReservation request: {"customerName": "Cornell_65", "hotelId": "29", "inDate": "2015-04-21", "outDate": "2015-04-23", "roomNumber": 1}
Full JSON Response: {"code": "0Qm6IHNl3ZfToRNOSKfSgWYh1408f8Ns", "error": "Too many requests in the last minute (count: 4970, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
Full JSON Response: {"code": "p0CF0Ku0QkrHVX6PbXV4EBHQ3nDVoxXC", "error": "Too many requests in the last minute (count: 4972, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
makeReservation request: {"customerName": "Cornell_460", "hotelId": "22", "inDate": "2015-04-17", "outDate": "2015-04-20", "roomNumber": 1}
makeReservation response: {"code": "QJMtXbnMogz9KfXZw1EAYfQZHY1JZTJY", "error": "Too many requests in the last minute (count: 4969, allowed: 60)."}
Full JSON Response: {"code": "j83bKVpLXqbyU7JM6c0aGQTln66PNW4E", "error": "Too many requests in the last minute (count: 4974, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
makeReservation request: {"customerName": "Cornell_256", "hotelId": "65", "inDate": "2015-04-22", "outDate": "2015-04-24", "roomNumber": 1}
makeReservation response: {"code": "Ww1mIpekL2DWAGCT1KG8IzMDc0f8AtcF", "error": "Too many requests in the last minute (count: 4976, allowed: 60)."}
makeReservation response: {"code": "YIx0JD6BuRwFVY8sUn9aQgWc1lEZSLaL", "error": "Too many requests in the last minute (count: 4975, allowed: 60)."}
Full JSON Response: {"code": "TUEP2B3Q9rEPRIEd0FeRmXyNlJN4mXNc", "error": "Too many requests in the last minute (count: 4977, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
makeReservation request: {"customerName": "Cornell_64", "hotelId": "52", "inDate": "2015-04-18", "outDate": "2015-04-23", "roomNumber": 1}
makeReservation response: {"code": "fekznPUmCTY2VlSya97Dopyeex8LYBZ1", "error": "Too many requests in the last minute (count: 4978, allowed: 60)."}
makeReservation response: {"code": "ThT3BLghXImTYRk76e7XEpnPRpSeXwdz", "error": "Too many requests in the last minute (count: 4979, allowed: 60)."}
Full JSON Response: {"code": "pChnlgClnW7ncM4JlTuUr0LleJM2Ndbm", "error": "Too many requests in the last minute (count: 4981, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
Full JSON Response: {"code": "FwxZiZlpTIqprMBAqZzvd5PYwTn3T9F", "error": "Too many requests in the last minute (count: 4980, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
Full JSON Response: {"code": "OI2c1K9EublacmpklP4qpNqbbMh4vFEQ", "error": "Too many requests in the last minute (count: 4982, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
makeReservation request: {"customerName": "Cornell_152", "hotelId": "63", "inDate": "2015-04-15", "outDate": "2015-04-20", "roomNumber": 1}
Full JSON Response: {"code": "Jz60aQUhDlcsGZ67jYhtDiWTUqDl1HXN", "error": "Too many requests in the last minute (count: 4983, allowed: 60)."}
Extracted - Username: | Valid: false | Message: | Hashed Password:
makeReservation request: {"customerName": "Cornell_166", "hotelId": "17", "inDate": "2015-04-19", "outDate": "2015-04-20", "roomNumber": 1}
makeReservation response: {"code": "ac2HFh1gHvdSKcW75tRR47DgJHkl01Z", "error": "Too many requests in the last minute (count: 4984, allowed: 60)."}
makeReservation response: {"code": "xHoRkZyy5VeHkwQKW7ZGYe8FbInwko", "error": "Too many requests in the last minute (count: 4985, allowed: 60)."}
Full JSON Response: {"message": "User not found", "status": "success", "valid": false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:

```

Figure 7.1: Frontend service's pod logs showcasing request limitation errors from action outputs while conducting benchmarking experiments

These messages, visible in the action responses, originated from OpenWhisk's internal throttling mechanism that enforces per-minute invocation limits as a protective measure. The system was registering over 4,000 invocations per minute against a default policy that allowed only 60. This bottleneck effectively prevented accurate measurement of the action's scalability and latency under pressure and signaled a misalignment between the platform's rate-limiting configuration and the load characteristics of the benchmark.

To address this issue, the OpenWhisk configuration was modified to lift the restrictive default limits. Specifically, the Helm deployment configuration file `mycluster.yaml`, from the OpenWhisk-Cloudlab configuration repository, was edited to increase the allowed number of concurrent and per-minute action invocations. The `limits.actionsInvokesPerminute` parameter was raised to 10000, and the `limits.actionsInvokesConcurrent` was similarly increased to 5000. These changes ensured that the serverless platform could support a high-throughput scenario without throttling test traffic. Additionally, memory and duration limits for action execution were expanded.

After updating the configuration file, the OpenWhisk system was redeployed via Helm, applying the new resource parameters across the cluster. This process included reinitializing the controller and invoker pods and verifying the applied limits through platform introspection tools. Once the platform had been restarted and stabilized, benchmarking was resumed under identical load conditions. The absence of rate-limit errors in subsequent test runs, which can be observed in Figure 7.2, confirmed the effectiveness of the reconfiguration enabling valid performance measurements across all load tiers and implementations.

```

● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_66","hotelId":"48","inDate":"2015-04-11","outDate":"2015-04-12","roomNumber":1}
makeReservation response: {"hotelId":"48","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_325","hotelId":"79","inDate":"2015-04-11","outDate":"2015-04-16","roomNumber":1}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_51","hotelId":"13","inDate":"2015-04-22","outDate":"2015-04-23","roomNumber":1}
makeReservation response: {"hotelId":"13","message":"Reservation confirmed","status":"success"}
makeReservation response: {"hotelId":"79","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_332","hotelId":"60","inDate":"2015-04-14","outDate":"2015-04-16","roomNumber":1}
makeReservation response: {"hotelId":"60","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_354","hotelId":"22","inDate":"2015-04-14","outDate":"2015-04-17","roomNumber":1}
makeReservation response: {"hotelId":"22","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_289","hotelId":"25","inDate":"2015-04-18","outDate":"2015-04-19","roomNumber":1}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_123","hotelId":"74","inDate":"2015-04-10","outDate":"2015-04-12","roomNumber":1}
makeReservation response: {"hotelId":"25","message":"Reservation confirmed","status":"success"}
makeReservation response: {"hotelId":"74","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_460","hotelId":"64","inDate":"2015-04-19","outDate":"2015-04-22","roomNumber":1}
makeReservation response: {"hotelId":"64","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_51","hotelId":"13","inDate":"2015-04-22","outDate":"2015-04-23","roomNumber":1}
makeReservation response: {"hotelId":"13","message":"Reservation confirmed","status":"success"}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_436","hotelId":"48","inDate":"2015-04-20","outDate":"2015-04-22","roomNumber":1}
● Full JSON Response: {"message":"User not found","status":"success","valid":false}
Extracted - Username: | Valid: false | Message: User not found | Hashed Password:
makeReservation request: {"customerName":"Cornell_332","hotelId":"60","inDate":"2015-04-14","outDate":"2015-04-16","roomNumber":1}
makeReservation response: {"hotelId":"48","message":"Reservation confirmed","status":"success"}
makeReservation response: {"hotelId":"60","message":"Reservation confirmed","status":"success"}

```

Figure 7.2: Frontend service's pod logs showcasing expected output behavior after conducting benchmark experiments

12.3 Performance Evaluation

After ensuring a stable environment for benchmarking experimentation that accurately handles our chosen workload intensity tiers in comparison to the system's expected behavior, the next step in the testing phase was to assess the performance of the serverless implementation relative to the original microservice-based system. This was done by analyzing latency across different percentiles under the different load profiles established previously: light, medium, and high. The goal was to capture how the system responds not only under normal circumstances but also during peak stress, and to observe the implications of cold starts in the serverless context, as well as the benefits of warm invocations. These measurements were visualized through

percentile-based latency distribution charts, which enabled direct comparison of the three deployment modes: traditional microservice, cold-start serverless, and warm-start serverless.

Warming was achieved by applying a short high intensity load on the hybrid system, prompting it to create a large amount of action containers. The system was then let to briefly cool down for 5 to 10 seconds, enough time for OpenWhisk to execute container termination for initialized but unutilized action containers. After that amount of time each load profile described previously was executed so the system would maintain only the required amount of containers.

Cold starts were ensured by waiting the predefined amount of time between 10 to 12 minutes after each load profile execution. This guaranteed all versions of action containers were terminated completely without inherited resources or preserved connections to supporting services. The goal was to account for the time demanded to both create the actions container for each invoked action and execute the `init()` function commands, typically establishing MongoDB and Memcached connections with their appropriate services. As discussed later, latency values affected by these factors reached alarmingly high amounts, presenting a serious disadvantage for cold executions.

12.3.1 Reservation Service

The reservation service revealed greater performance divergence across implementations, particularly at higher loads and percentiles. While microservices demonstrated strong performance at median and 90th percentile latencies under all loads, they were outpaced by warm serverless actions in tail latency behavior. Under medium load, warm executions achieved superior performance in the 99.9th and higher percentiles, with latency remaining under 184 ms, while microservices rose above 500 ms. Even under high load, warm actions sustained lower 99.999th percentile latencies (202 ms) than their microservice counterparts (308 ms), suggesting that OpenWhisk's stateless invocation model can produce more consistent high-percentile responses in resource-isolated conditions. Cold-start actions, however, incurred substantial delays, consistently exceeding 30 seconds at peak percentiles and rendering them impractical for time-sensitive reservation flows.

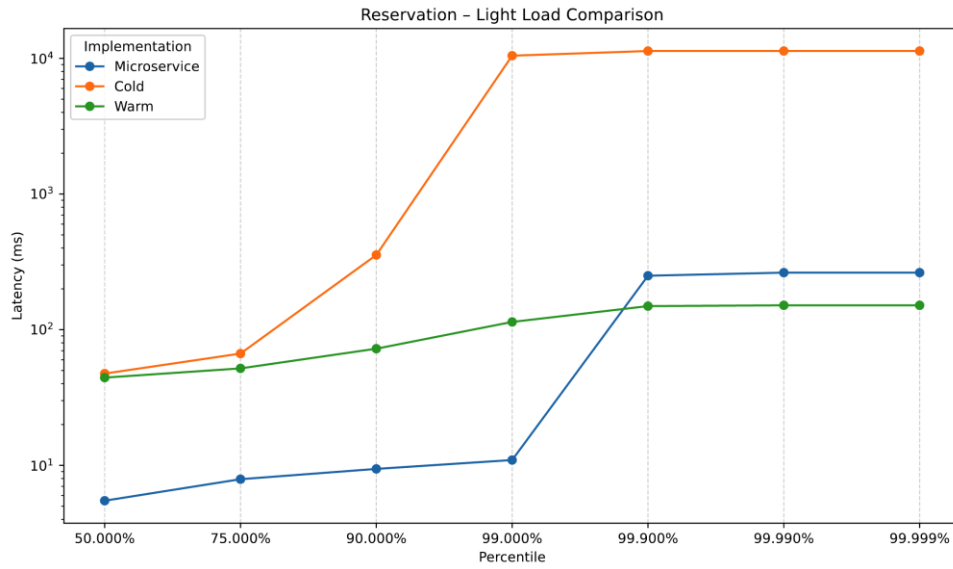


Figure 7.3: Manually Converted Reservation Service Benchmarks Latencies – Light
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

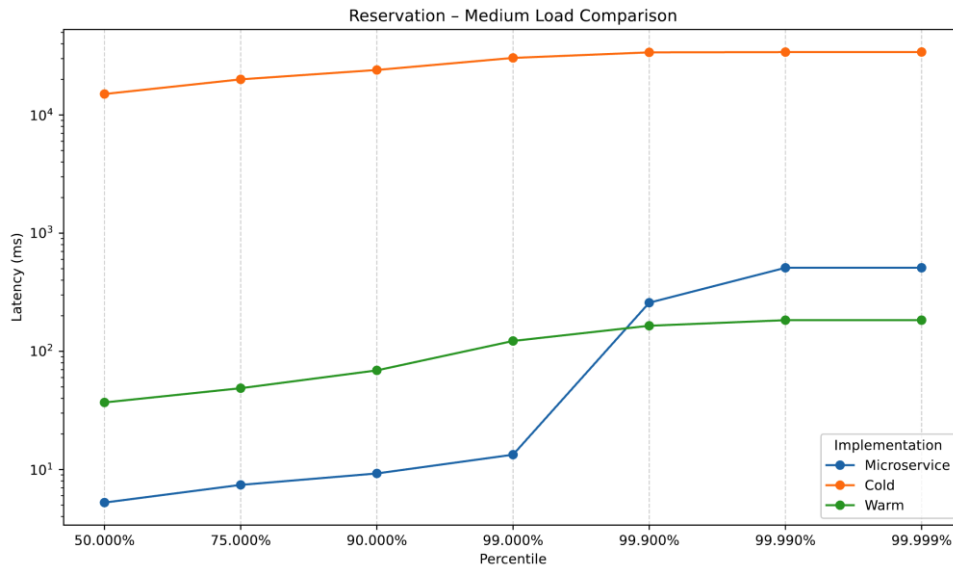


Figure 7.4: Manually Converted Reservation Service Benchmarks Latencies – Medium
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

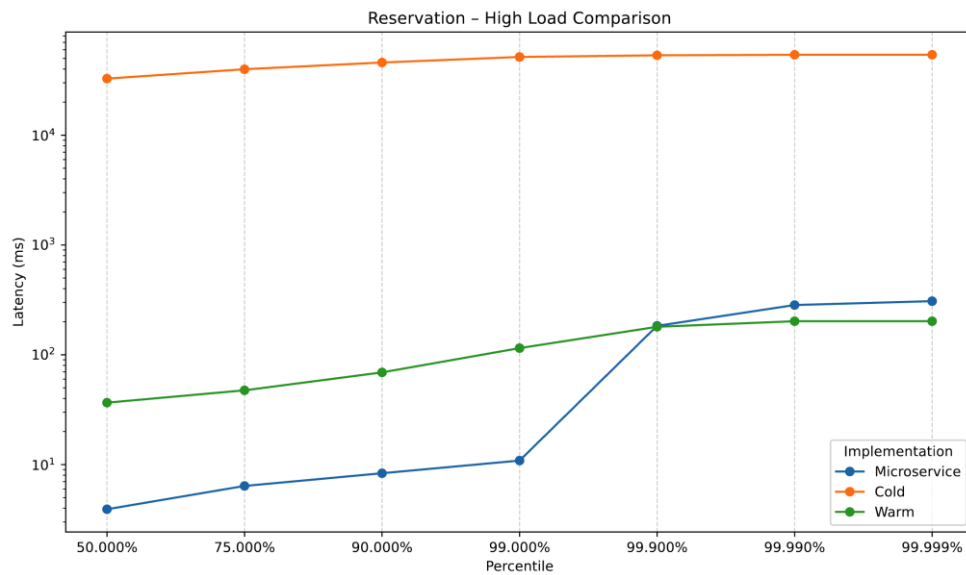


Figure 7.5: Manually Converted Reservation Service Benchmarks Latencies – High
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

In the microservice version, average latency stays very low across all loads, while the 99.999th percentile latency peaks at medium load, possibly due to a concurrency bottleneck. In the serverless cold setup, both average and high-end latencies rise sharply with load the 99.999th percentile reaches over 54 seconds under heavy traffic. This shows that cold starts add a large delay when containers have to be created anew. The warm serverless setup performs much better, with both latency values growing only slightly as the load increases. This confirms that warm containers can handle traffic more smoothly without long delays.

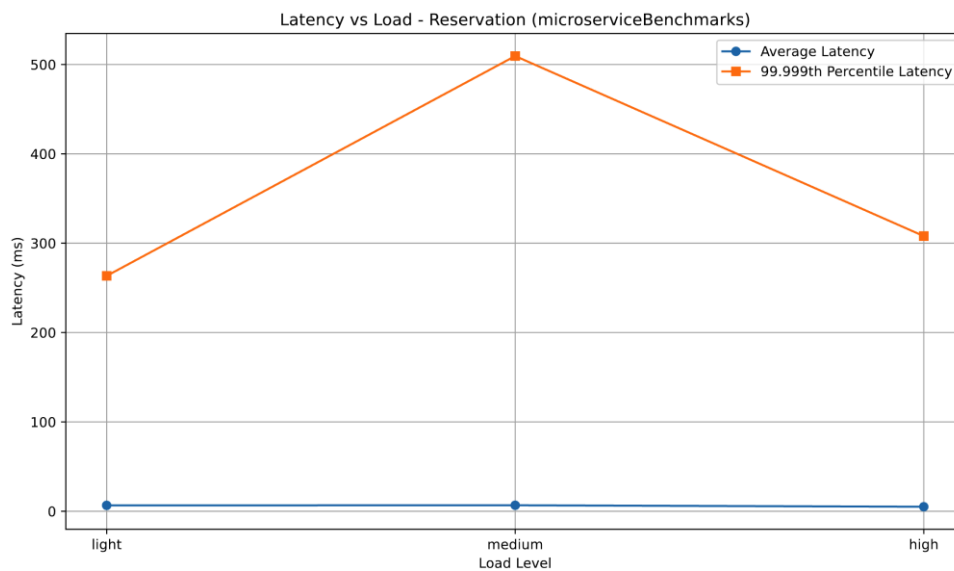


Figure 7.6: Reservation Service Average and High Percentile Latency Comparison – Microservice -
X-Axis: Latency in ms, Y-Axis: Load Level

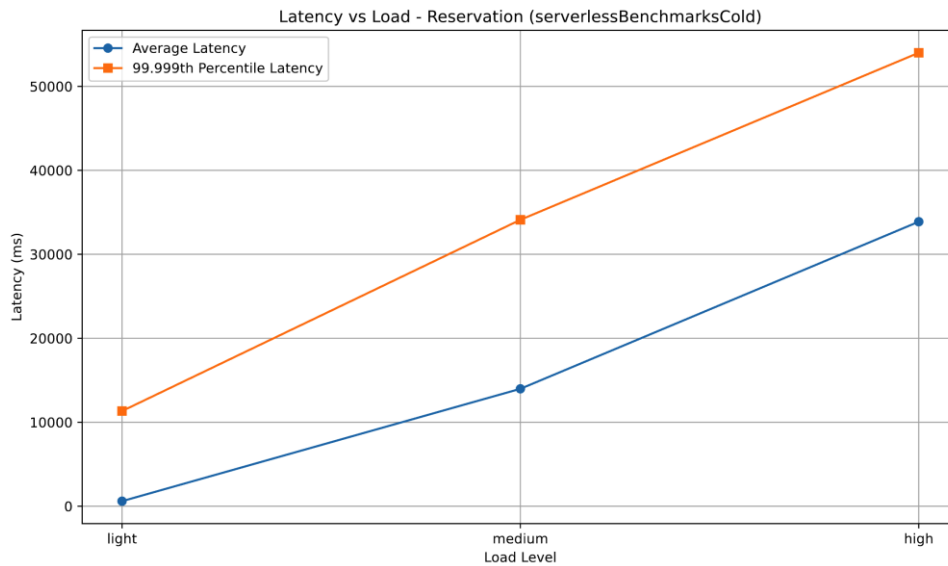


Figure 7.7: Reservation Service Average and High Percentile Latency Comparison – Serverless Cold
Start - X-Axis: Latency in ms, Y-Axis: Load Level

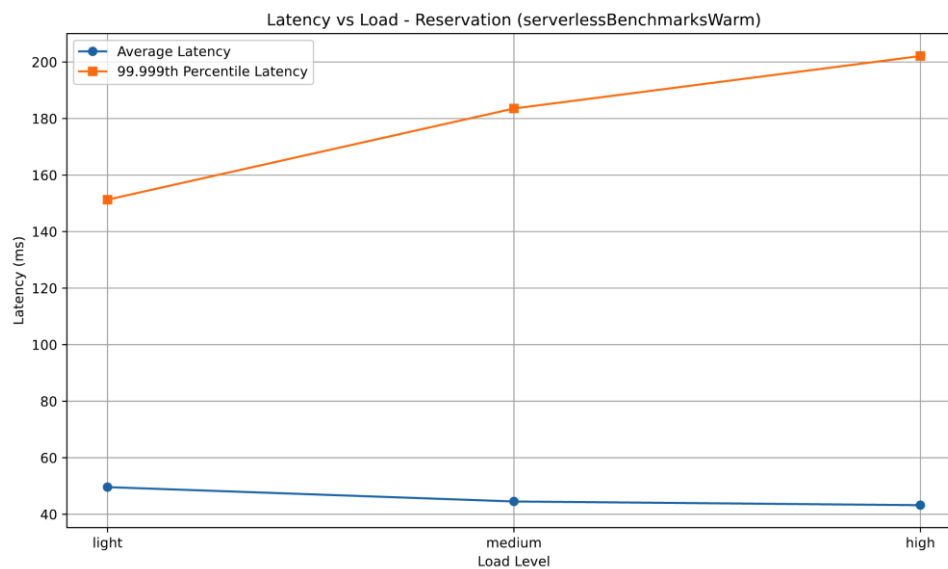


Figure 7.8: Reservation Service Average and High Percentile Latency Comparison – Serverless Warm
Start - X-Axis: Latency in ms, Y-Axis: Load Level

12.3.2 User Service

The user service exhibited consistently low latency across all implementations and load tiers, with minimal variability at lower percentiles. Under light and medium loads, both the microservice and serverless warm implementations delivered near-identical performance, maintaining latency well below 30 ms even at the 99.999th percentile. At high load, the warm serverless function sustained a stable tail latency profile, peaking at only 40 ms, compared to the microservice’s 16.58 ms. However, the warm execution showed better percentile

progression, maintaining a smoother rise across percentile boundaries. In contrast, the cold start serverless version underperformed significantly, with latency escalating to over 50 seconds under high load at the 99.999th percentile.

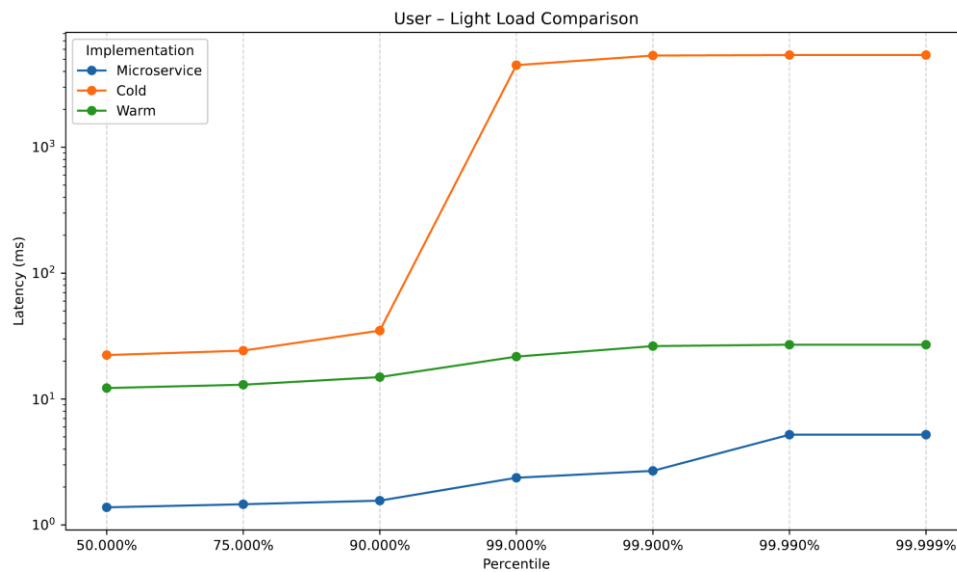


Figure 7.10: User Service Benchmarks Latencies – Light

X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

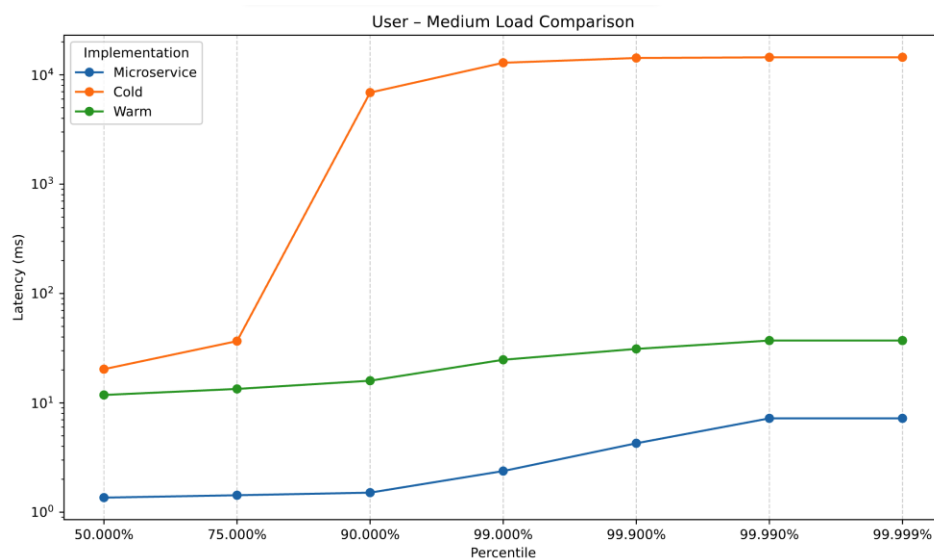


Figure 7.11: User Service Benchmarks Latencies – Medium

X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

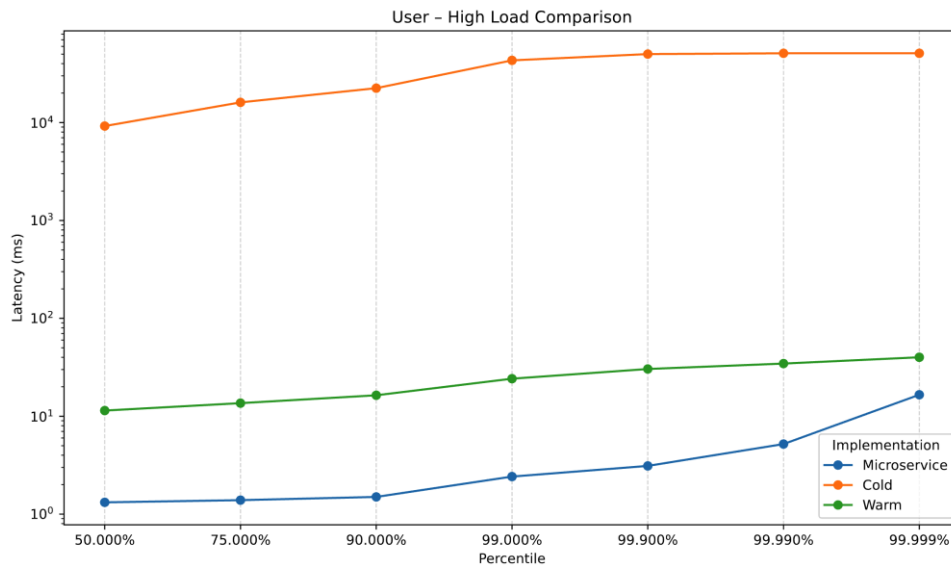


Figure 7.12: User Service Benchmarks Latencies – High
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

Microservices stay consistently fast, and even in the serverless cold case, average latency remains manageable. However, under high load, cold serverless latency spikes to over 50 seconds at the highest percentile, showing how much cold starts can slow things down. Warm serverless actions, however, stay close to the microservice results, even under heavy traffic. This suggests that small services benefit a lot from being kept “warm” and ready to run, avoiding startup delays.

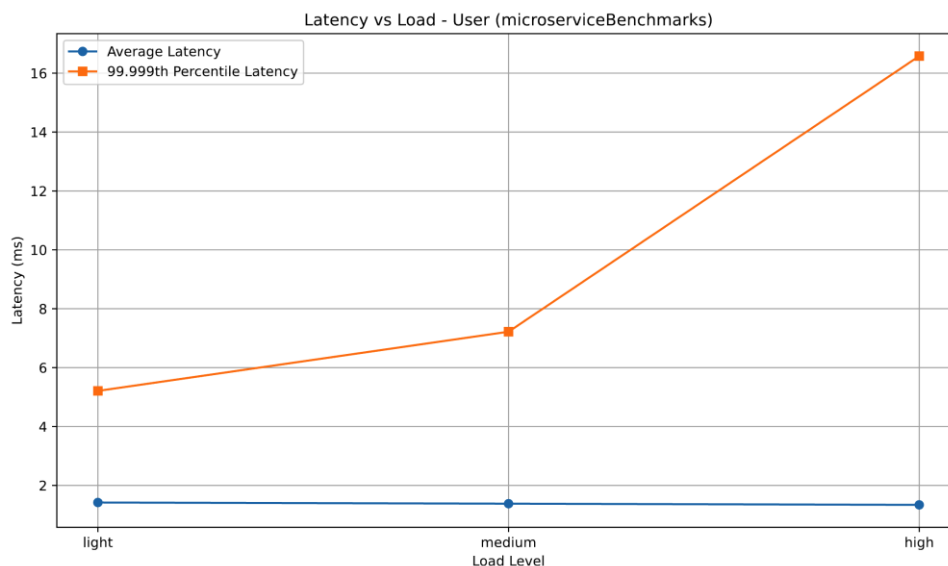


Figure 7.13: User Service Average and High Percentile Latency Comparison – Microservice –
X-Axis: Latency in ms, Y-Axis: Load Level

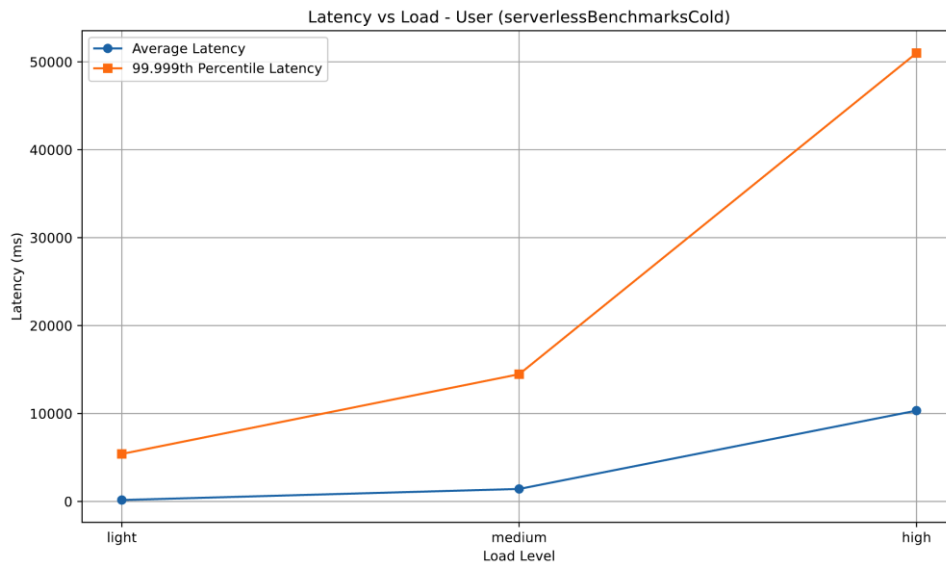


Figure 7.14: User Service Average and High Percentile Latency Comparison – Serverless Cold Start –
X-Axis: Latency in ms, Y-Axis: Load Level

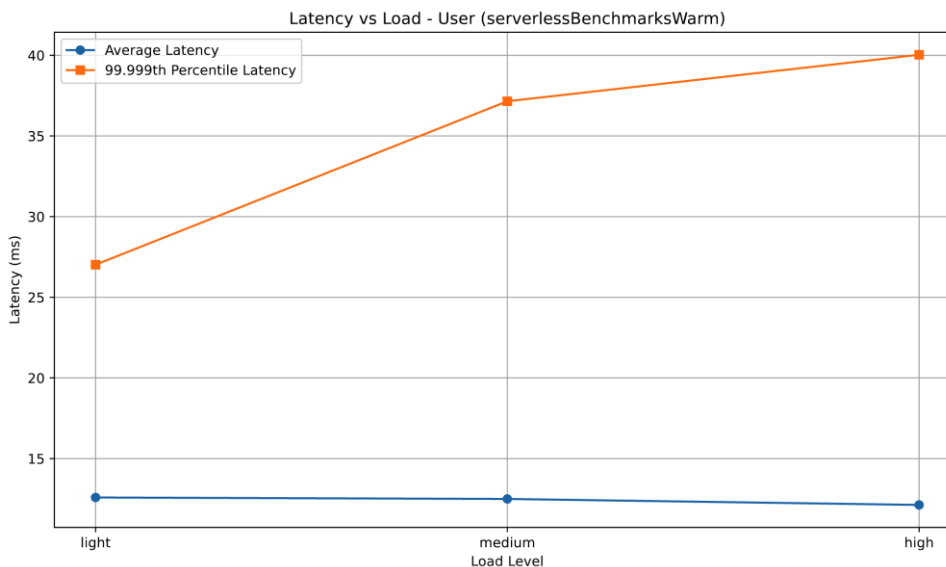


Figure 7.15: User Service Average and High Percentile Latency Comparison – Serverless Warm Start
X-Axis: Latency in ms, Y-Axis: Load Level

12.3.3 Mixed Service Workload

The mixed service, representing a complex invocation pattern across several microservices, provided insight into end-to-end system behavior under varying load. At light and medium load, both microservice and warm serverless implementations performed within acceptable latency boundaries, with 99.999th percentiles recorded at 87 ms and 85 ms respectively. Under high load, warm serverless actions slightly outperformed microservices in upper percentile stability, achieving 87 ms at 99.999% compared to microservice performance peaking around 251 ms. The cold-start configuration, on the other hand, demonstrated critical latency

bottlenecks with values reaching up to 52 seconds, reinforcing the need to avoid cold invocations in multi-service coordinated workloads.

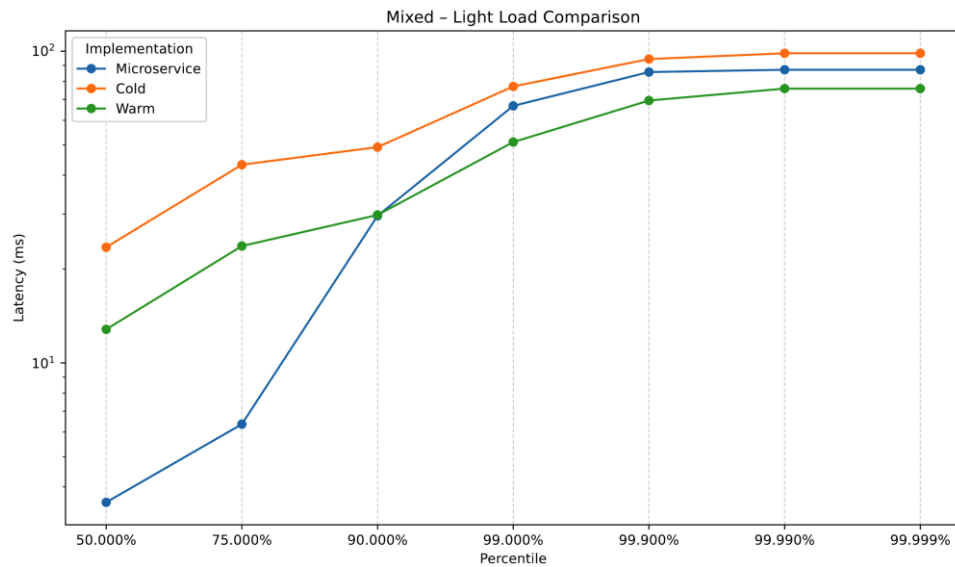


Figure 7.16: Mixed Service Workload Benchmarks Latencies – Light
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

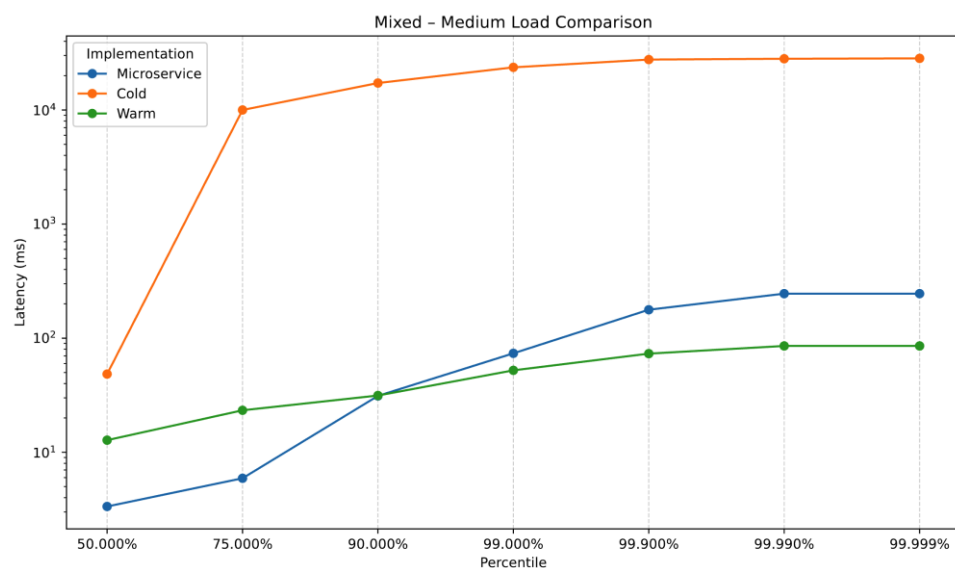


Figure 7.17: Mixed Service Workload Benchmarks Latencies – Medium
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

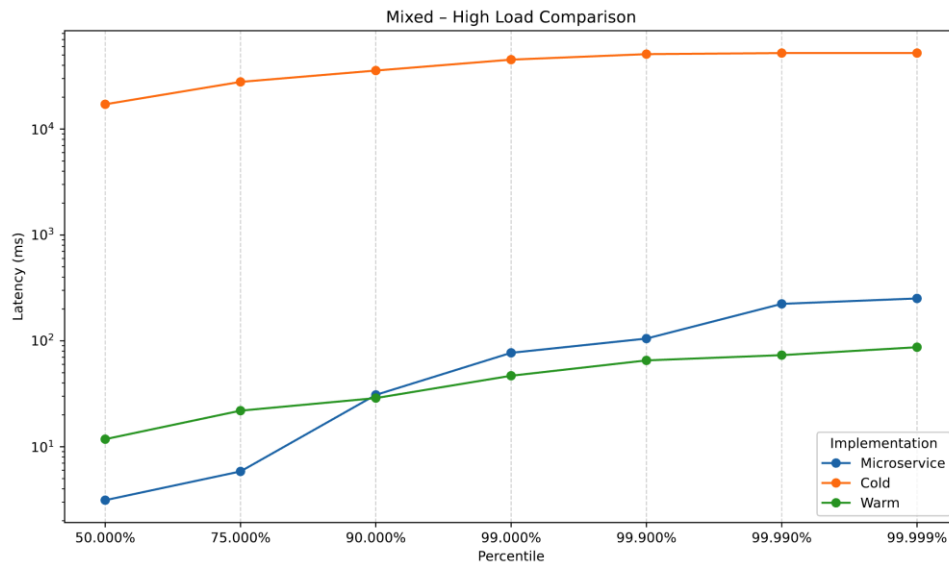


Figure 7.18: Mixed Service Workload Benchmarks Latencies – High
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

Microservices can handle even high loads fairly well, but the cold serverless results show a rise in both average and tail latencies, reaching over 50 seconds at the 99.999th percentile. This is because cold starts must happen for multiple actions across services. On the other hand, the warm serverless configuration has stable latencies even under load, the highest percentile stays below 90 milliseconds.

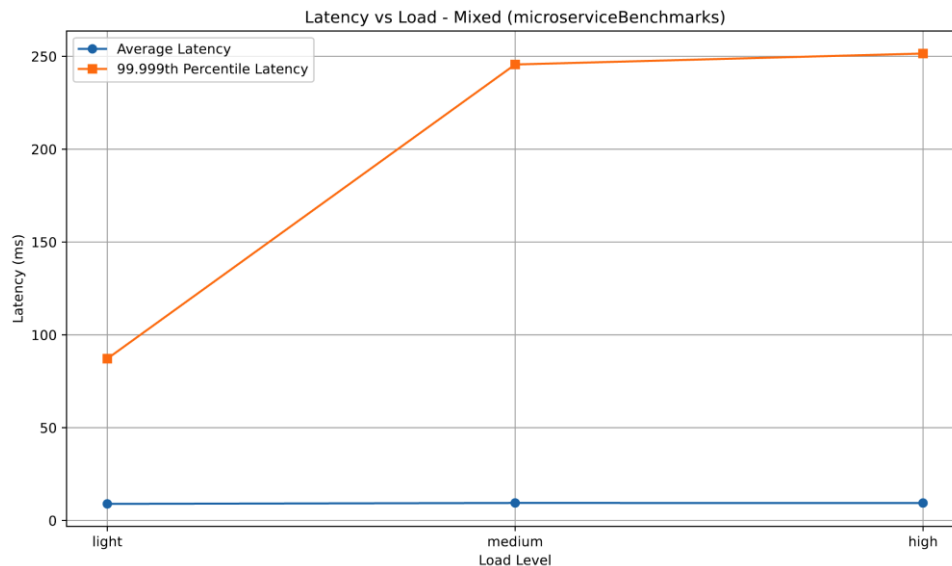


Figure 7.19: Mixed Service Workload Average and High Percentile Latency Comparison – Microservice- X-Axis: Latency in ms, Y-Axis: Load Level

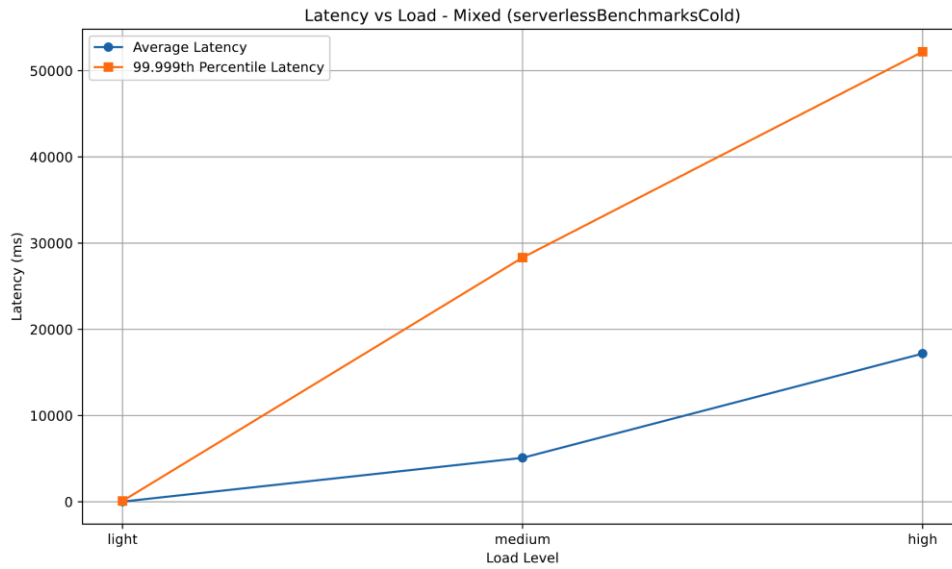


Figure 7.20: Mixed Service Workload Average and High Percentile Latency Comparison – Serverless Cold Start - X-Axis: Latency in ms, Y-Axis: Load Level

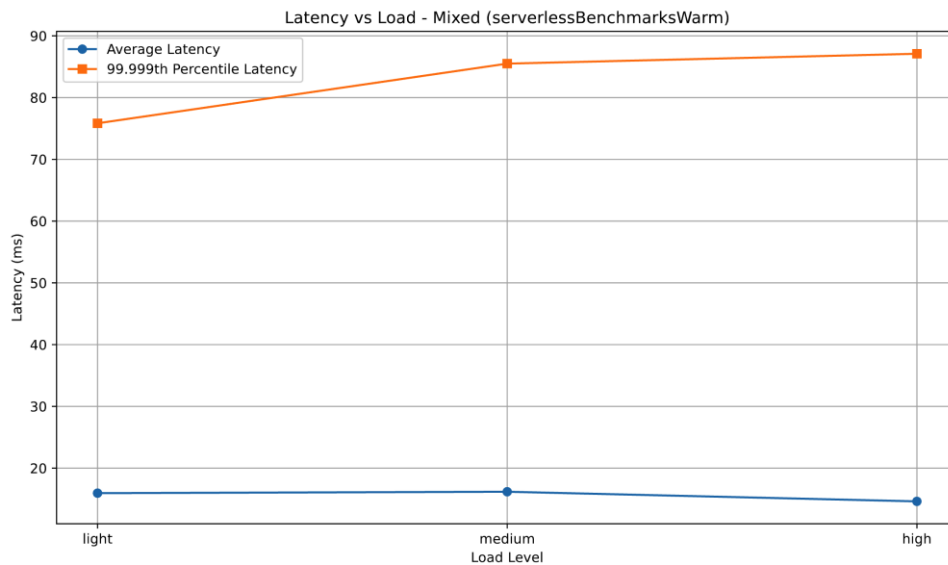


Figure 7.21: Mixed Service Workload Average and High Percentile Latency Comparison – Serverless Warm Start - X-Axis: Latency in ms, Y-Axis: Load Level

12.3.4 Automatically Converted Reservation Service

The benchmarking results from the automatically converted makeReservation action show that the AI tool we constructed for the particular task was successful in producing a version that performs almost identically to the manually converted one. When we tested it under the same load profiles as the manually converted version, the action displayed very similar latency patterns across all percentiles. Specifically, in the warm execution the latency remained low and consistent, just like in the manual version. This suggests that the AI-generated function

handled input processing, logic maintenance, and supporting service communication in a correct and efficient way.

Even in cold starts, where the serverless platform has to initialize containers from scratch, the latency values closely matched those demonstrated by the manual benchmarks. Though cold starts are expected to have higher delays, the consistent behavior through the different percentiles shows that the tool correctly handled initial startup steps like loading dependencies and connecting to databases. The microservice results stayed the same as before, confirming the hybrid system worked as expected without being affected by the change in the action's implementation.

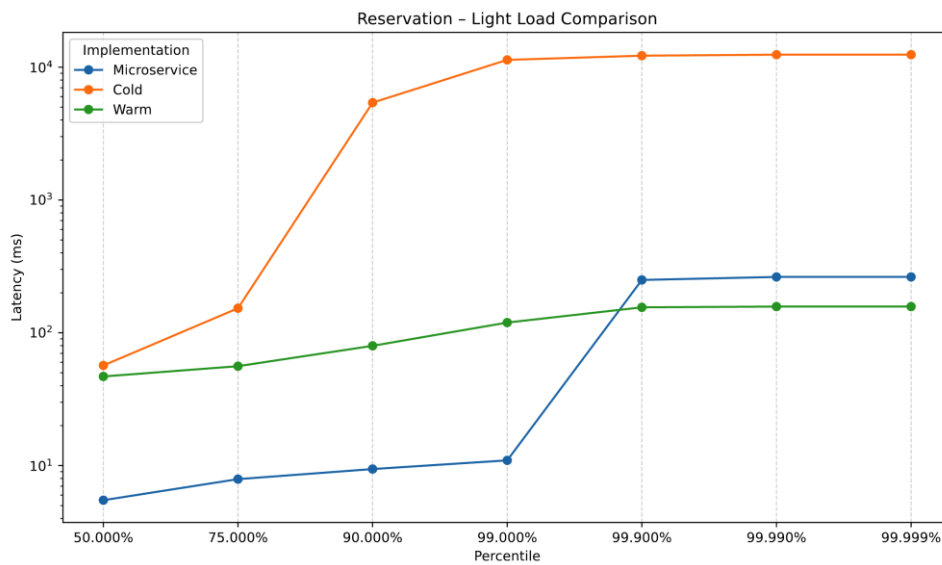


Figure 7.22: Automatically Converted Reservation Service Benchmarks Latencies – Light
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

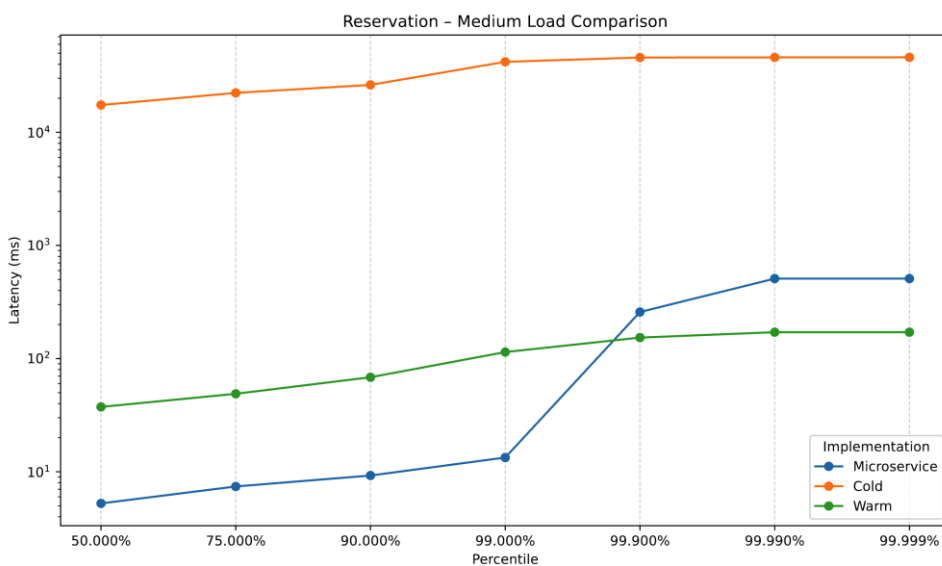


Figure 7.23: Automatically Converted Reservation Service Benchmarks Latencies – Medium
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

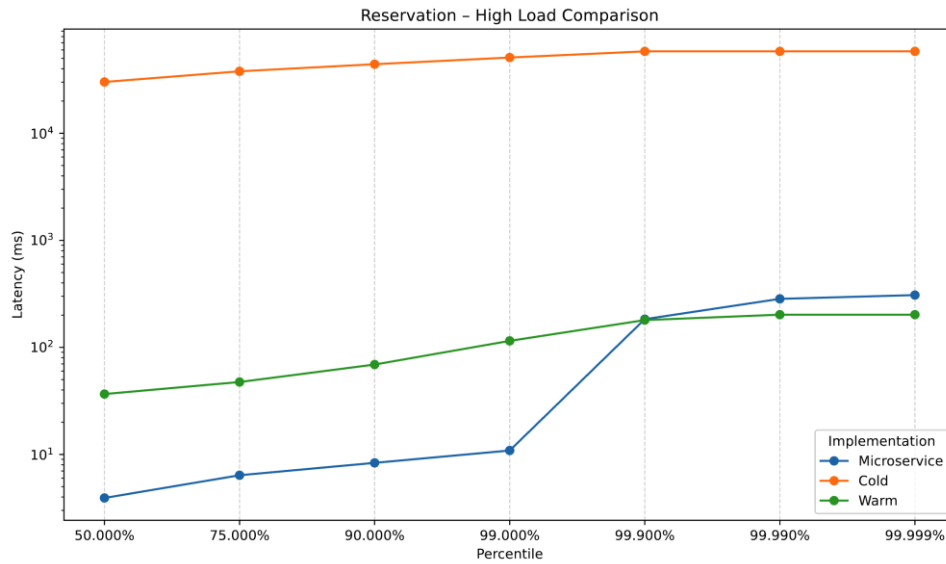


Figure 7.24: Automatically Converted Reservation Service Benchmarks Latencies – High
X-Axis: Logarithmic Latency in ms, Y-Axis: Latency Percentiles

Overall, these results confirm that the AI-based method for converting microservices to serverless functions is reliable and accurate. It successfully reproduces the logic and performance of the original service, proving that the tool works well and the LLM utilized by it can infer and accurately transform the logic we require. Therefore, our method can be trusted for future automated transformations.

12.4 Conclusion

The benchmarking evaluation of the transformed serverless architecture versus the original microservice-based implementation reveals critical insights into the performance characteristics and trade-offs from each approach. Across all tested services user, reservation, and mixed it is evident that serverless cold starts present a major latency bottleneck, deeming such configurations unsuitable for low-latency workloads. Cold-start executions frequently exceeded tens of seconds in response time, especially under medium to high load, which severely impacts system responsiveness.

However, the warm serverless configuration consistently demonstrated competitive performance, often matching or exceeding the microservice implementation at the upper latency percentiles. Particularly in high-stress scenarios, warm actions exhibited greater

stability and predictability in tail latencies, suggesting that OpenWhisk’s stateless execution model, when optimized for reuse, can yield robust and low-variance responses.

The microservice architecture, while still delivering the most consistent average-case latency, showed signs of degradation in extreme percentile cases. This highlights the importance of evaluating not just average performance but also latency distribution tails, which have significant implications for user-perceived responsiveness.

The automatically converted implementation of the reservation service also demonstrated promising results. With corresponding results to its manual counterpart, the automated implementation behaved the same at all latency percentiles at each architectural state, especially serverless warm and cold. These results validate the choice of LLM utilization in our research both in terms of efficiency and effectiveness.

In summary, these results underscore the viability of warm serverless deployments as an effective and scalable alternative to traditional microservices provided that cold-start penalties are adequately mitigated. The hybrid approach explored in this research illustrates how targeted service transformation can preserve system correctness while leveraging the benefits of function-as-a-service platforms for elasticity and functional versatility.

Chapter 13

Conclusions

13.1	Conclusion.....	97
13.2	Future Work	98

13.1 Conclusion

This work demonstrates that it is indeed possible to extract and reimplement microservice logic as fully functional, independently deployable serverless actions. By converting core components of the Hotel Reservation application, such as user authentication and reservation management, into stateless OpenWhisk actions, this research proves that serverless platforms can accommodate complex service logic originally designed for interconnected microservices. The resulting hybrid system, integrating both microservice and serverless elements, was deployed and evaluated under realistic workloads. Performance testing showed that warm-start serverless functions could not only maintain functional parity with their microservice counterparts, but in some cases, deliver superior latency characteristics in higher percentiles, validating the feasibility and competitiveness of the hybrid model.

Through repeated application and refinement, a structured methodology emerged for transforming Go-based microservices into serverless functions. This methodology emphasized the isolation of business logic, elimination of stateful dependencies, reformulation of function structure into OpenWhisk-compatible formats, and the externalization of supporting services such as databases and caches. By applying this process across distinct services and confirming the successful replacement of microservice behavior within the live application, the thesis establishes a transferable and repeatable approach. The methodology's practical value was further reinforced by its alignment with the needs of real-world deployment scenarios, where maintainability and platform compliance are critical.

Building on this foundation, the thesis culminated in the development of a CLI-based automation tool leveraging generative AI. This tool demonstrated the capacity to produce valid, high-fidelity OpenWhisk actions from annotated microservice source code, using structured prompts and reference examples as guidance. The quality of the generated outputs, particularly in the reproduction of the complex makeReservation logic, validates the role of LLMs as powerful tools in code transformation workflows. The integration of semantic code understanding into the automation process overcomes the limitations of static analysis tools, opening the door to scalable, intelligent service conversion. In this way, the thesis provides both a proven manual methodology and a forward-looking, AI-assisted path for migrating microservices to serverless computing, enabling more flexible and efficient application architectures.

13.2 Future Work

One promising avenue for future work involves the integration of the conversion automation process into the Blueprint [23] framework, an architecture-aware tool for generating modular and reconfigurable microservice applications. Blueprint’s compositional design and parameterized configuration of services make it a fitting environment for embedding conversion logic directly into the service generation pipeline. By leveraging the existing Blueprint implementation of the Hotel Reservation application, which includes clean service boundaries and explicit service definitions, it becomes feasible to automate not just the transformation of isolated services but the end-to-end adaptation of an entire application toward a serverless-ready structure. Embedding LLM-powered transformation capabilities into Blueprint could streamline the deployment of hybrid systems.

Another area of exploration lies in merging predictive traffic analytics with dynamic serverless orchestration strategies. While this thesis established the performance viability of warm serverless actions, their benefits are dependent on container availability. Future systems could integrate predictive models that analyze historical and real-time traffic patterns to anticipate service demand and proactively warm specific actions in advance. These pre-warmed actions could be maintained only for the necessary window, shutting down automatically afterward to preserve cost-efficiency. Such predictive warm-start scheduling could optimize latency under load without incurring the resource overhead of always-on provisioning, bringing serverless deployments closer to both performance and economic optimality in production-grade systems.

References

- [1] A. Kansal, A. Wolman, J. Nelson, S. Ghanbari, and E. de Lara, “Boxer: Interposition for Transparent Cloud Elasticity,” in *Proc. 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 763–776, 2020.
- [2] H. Qiu, Y. Mao, Z. Qian, and M. Qiu, “Executing Microservice Applications on Serverless Correctly,” in *Proc. 1st Workshop on Serverless Computing (WoSC '17)*, 2017.
- [3] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso, “Imaginary Machines: A Serverless Model for Cloud Applications,” in *Proc. 2nd Workshop on Serverless Systems, Applications and Methodologies (SESAME '24)*, ACM, 2024.
- [4] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, “FaaS\$: A Transparent Auto-Scaling Cache for Serverless Applications,” in *Proc. ACM Symposium on Cloud Computing (SoCC '21)*, pp. 1–16, 2021.
- [5] A. Fuerst and P. Sharma, “FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching,” in *Proc. 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, pp. 1–15, 2021.
- [6] M. Wawrzoniak, G. Moro, R. Bruno, A. Klimovic, and G. Alonso, “Off-the-shelf Data Analytics on Serverless,” in *Proc. 14th Conference on Innovative Data Systems Research (CIDR '24)*, 2024.
- [7] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless Computing: An Investigation of Factors Influencing Microservice Performance,” in *Proc. IEEE International Conference on Cloud Engineering (IC2E '18)*, pp. 159–169, 2018.
- [8] A. Szekely, A. Belay, R. Morris, and M. F. Kaashoek, “Unifying Serverless and Microservice Tasks with SigmaOS,” in *Proc. 29th ACM Symposium on Operating Systems Principles (SOSP '24)*, pp. 1–15, 2024.
- [9] OpenFaaS, “OpenFaaS Documentation,” [Online]. Available: <https://docs.openfaas.com>. [Accessed: Apr., 2025].

- [10] Fission, “Fission Documentation,” [Online]. Available: <https://docs.fission.io>. [Accessed: Apr., 2025].
- [11] Knative, “Knative Documentation,” [Online]. Available: <https://knative.dev/docs/>. [Accessed: Apr., 2025].
- [12] Apache Software Foundation, “Apache OpenWhisk Documentation,” [Online]. Available: <https://OpenWhisk.apache.org/documentation.html>. [Accessed: Apr, 2025].
- [13] OpenWhisk-Devtools <https://github.com/apache/OpenWhisk-devtools>
- [14] Docker for Desktop <https://www.docker.com/products/docker-desktop>
- [15] Apache OpenWhisk <https://github.com/apache/OpenWhisk>
- [16] CloudLab profile for deploying OpenWhisk via Kubernetes - [cloudlab-OpenWhisk](https://github.com/CU-BISON-LAB/cloudlab-OpenWhisk)
<https://github.com/CU-BISON-LAB/cloudlab-OpenWhisk>
- [17] K. Sriraman, A. Austruy, J. Shun, and C. Delimitrou, “DeathStarBench: A Benchmark Suite for Cloud Microservices,” in *Proc. 2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–12, 2020. [Online]. Available: <https://github.com/delimitrou/DeathStarBench>
- [18] MongoDB Inc., “MongoDB: The Developer Data Platform,” [Online]. Available: <https://www.mongodb.com>. [Accessed: May, 2025].
- [19] The Go Authors, “The Go Programming Language,” [Online]. Available: <https://golang.org>. [Accessed: May, 2025].
- [20] Google, “gRPC: A High-Performance, Open-Source Universal RPC Framework,” [Online]. Available: <https://grpc.io>. [Accessed: May, 2025].
- [21] MongoDB Inc., “The Official MongoDB Go Driver,” [Online]. Available: <https://pkg.go.dev/go.mongodb.org/mongo-driver>. [Accessed: May, 2025].

[22] Apache Software Foundation, “openwhisk-runtime-go: Go Runtime for Apache OpenWhisk,” GitHub, 2024. [Online]. Available: <https://github.com/apache/openwhisk-runtime-go>.

[23] V. Anand, D. Garg, A. Kaufmann, and J. Mace, “Blueprint: A Toolchain for Highly-Reconfigurable Microservices,” in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 973–987.

[24] B. Fitzpatrick, *gomemcache: Go Memcached client library*, GitHub repository, <https://github.com/bradfitz/gomemcache>, Accessed: May, 2025.

[25] Kubernetes Authors, *Kubernetes: Production-Grade Container Orchestration*, GitHub repository and documentation, <https://kubernetes.io/>, Accessed: May, 2025.