**IMT School for Advanced Studies, Lucca**
Lucca, Italy

# Efficient Resource Management for Cloud-Native Systems: A Model-Based Optimization Approach

PhD Program in Systems Science

Track in Computer Science and Systems Engineering

XXXVI Cycle

**By**

**Roberto Pizziol**

**2026**

**The dissertation of Roberto Pizziol is approved.**

PhD Program Coordinator: Alberto Bemporad, IMT School for Advanced Studies Lucca

Advisor: Prof. Mirco Tribastone, IMT School for Advanced Studies Lucca

Co-Advisor: Dr. Emilio Incerto, IMT School for Advanced Studies Lucca

The dissertation of Roberto Pizziol has been reviewed by:

Dr. Giovanni Quattrocchi , Politecnico di Milano

Dr. Lishan Yang, George Mason University

IMT School for Advanced Studies Lucca
2026

# Contents

# List of Figures

# List of Tables

# Acronyms

**API** Application Programming Interface. 10, 16, 23, 39, 63

**APM** Application Performance Monitoring. 39

**CPU** Central Processing Unit. viii, xvi, 1, 5–7, 12, 17–19, 21, 23, 25, 26, 33, 34, 39, 40, 42, 45–47, 49, 50, 52–55, 57–59, 61–63, 65, 67, 68, 70, 71, 74, 82, 85, 91

**CTMC** Continuous-Time Markov Chain. 24

**DAG** Directed Acyclic Graph. 23, 33, 65, 78

**DL** Deep Learning. 32

**FaaS** Function-as-a-Service. v–vii, ix, xvi, 1–4, 6, 8, 13–19, 22, 30, 32–35, 62–65, 68–70, 77, 78, 81, 82, 84, 87, 88, 90, 91

**GCP** Google Cloud Platform. 56

**GCR** Google Cloud Run. viii, 3, 6, 18, 19, 62, 69, 70, 73–77, 79–82, 85–88

**GKE** Google Kubernetes Engine. ix, 12, 44, 51, 52, 60, 87

**HPA** Horizontal Pod Autoscaler. vi, viii, ix, 3, 5, 7, 38, 44, 51–53, 56, 58–61, 85, 87, 88

**HTTP** Hypertext Transfer Protocol. 7, 10, 14, 23, 45, 53, 63, 65, 70, 72

**IaaS** Infrastructure-as-a-Service. 14, 15

**IPOPT** Interior Point Optimizer. 4, 27, 28, 42

**K8s** Kubernetes. 3, 5

**KKT** Karush-Kuhn-Tucker. 28

**KPI** Key Performance Indicator. 40, 41, 58

**LQN** Layered Queueing Network. vii, xvi, 2, 4–6, 8, 20–22, 24, 27, 28, 31, 33, 34, 36, 37, 39–44, 46, 47, 51, 53–56, 60, 63–68, 71–73, 82, 84–87, 89, 91

**MAE** Mean Absolute Error. 54, 57

**MARL** Multi-Agent Reinforcement Learning. 34

**ML** Machine Learning. 32

**MPC** Model Predictive Control. 3, 4, 40, 87

**NLP** Non-Linear Programming. 4, 26–28, 85

**NP** Non-Deterministic Polynomial-time. 27

**ODE** Ordinary Differential Equation. xvi, 24, 25, 41, 46, 63, 65, 68

**PaaS** Platform-as-a-Service. 14, 15

**QN** Queueing Network. v, 20, 21, 31, 37

**QoS** Quality of Service. 2, 7, 12, 38, 86

**RAM** Random-access memory. 46, 53, 77

**REST** Representational State Transfer. 10

**RL** Reinforcement Learning. 32

**SLO** Service-Level Objective. 25, 32

**SPN** Stochastic Petri Net. 33

**vCPU** virtual Central Processing Unit. 53, 77

**VM** Virtual Machine. 77

**WAN** Wide Area Network. 90

# Acknowledgements

...

No generative AI was used to produce original research or text for this work; its use was restricted solely to stylistic and linguistic assistance.

# Vita

| | |
|---|---|
| **January 23, 1993** | Born, Rome, Italy |
| **2018** | Bachelor Degree in Computer Engineering<br>Final mark: 110/110<br>University of Rome "Tor Vergata" |
| **2020** | Master Degree in Computer and Information Engineering<br>Final mark: 109/110<br>University of Rome "Tor Vergata" |
| **2026** | PhD Scholarship<br>IMT School for Advanced Studies Lucca |

# Publications

1. E. Incerto, R. Pizziol, & M. Tribastone, "$\mu$Opt: An Efficient Optimal Autoscaler for Microservice Applications," in *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 67-76, 2023.
2. E. Incerto, R. Pizziol, G. Russo Russo & M. Tribastone, "WasteLess: An Optimal Provisioner for Self-Adaptive Second-Generation Serverless Applications," in *2025 IEEE/ACM International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 61-72, 2025.
3. E. Incerto, R. Pizziol, & M. Tribastone, "Efficient Microservice Autoscaling through $\mu$Opt," submitted to *IEEE Transactions on Services Computing* (under review).
4. E. Incerto, R. Pizziol, G. Russo Russo & M. Tribastone, "Enhancing Performance-Cost Trade-off in Second-Generation FaaS Systems Through WasteLess," submitted to *IEEE Transactions on Services Computing* (under review).

# Presentations

1. ...

# Abstract

Efficient resource management is a critical challenge in cloud-native systems, demanding a delicate balance between performance and operational cost. For microservices, this translates into the complex task of devising an efficient autoscaler that can adapt to dynamic workloads. In second-generation Function-as-a-Service (FaaS) systems, it becomes a difficult trade-off between cost-efficiency and performance degradation, complicated by the manual tuning of the concurrency limit. This thesis develops a unified, model-based optimization methodology to address these challenges. The approach leverages LQNs, made computationally efficient by a fluid approximation technique. By encoding the resulting ODE system as constraints within a non-linear optimization problem, we rapidly compute optimal configurations. This methodology is realized in two distinct frameworks: $\mu$Opt, an online autoscaler for microservices that iteratively adjusts CPU and thread-pool resources; and WasteLess, an offline provisioner that performs a single optimization to jointly configure the CPU, memory, and concurrency limit for second-generation FaaS applications. Extensive validation on production-grade cloud platforms proves the methodology's effectiveness against both industry and state-of-the-art research benchmarks. $\mu$Opt improves application performance by up to 9% while reducing resource consumption by up to 30%. For FaaS systems, WasteLess reduces operational costs by an average of 38% against non-concurrent deployments, without performance loss. When compared to the other benchmarks, it lowers latency by up to 72%, avoiding the severe performance penalties inherent in their methods.

# Introduction

## Motivation and Problem Statement

Cloud-native computing has emerged as the dominant paradigm for developing modern software systems. By abstracting away underlying infrastructure, this architectural shift enables applications to fully leverage the elasticity and distributed nature of the cloud, achieving unprecedented levels of scalability and resilience (Dragoni et al., 2017; Alonso et al., 2023).

Within this domain, microservices and Function-as-a-Service (FaaS) represent the prevailing implementation strategies. These architectures decompose complex applications into smaller, independent components, enabling modular development and granular resource management.

However, these innovations introduce profound operational challenges alongside their opportunities. The efficiency of such systems depends on the precise configuration of highly sensitive parameters. In the microservices domain, this requires the dynamic adjustment of CPU quotas, memory limits, and replica counts to match fluctuating workloads (Incerto, Tribastone, and Trubiani, 2018; Luciano Baresi, Hu, et al., 2021). Current industrial solutions often struggle to adapt quickly enough, leading to performance penalties during sudden load variations (Podolskiy, Jindal, and Gerndt, 2018; Razavi et al., 2022). Similarly, in the serverless context—particularly with second-generation FaaS—performance is critically dependent on the *concurrency limit* and memory allocation.

Failing to correctly tune these parameters results in a detrimental trade-off: over-provisioning incurs unnecessary operational costs, while under-

provisioning leads to severe Quality of Service (QoS) degradation (Eivy and Weinman, 2017; F. Liu and Niu, 2023; Nima Mahmoudi and Hamzeh Khazaei, 2020). Consequently, finding robust and automated solutions to these resource management challenges is of great importance for the operational sustainability of modern cloud systems.

## Contributions

This thesis addresses these challenges by developing a unified, model-based optimization methodology. By combining the expressive power of Layered Queueing Networks (LQNs) with the computational efficiency of the *fluid approximation*, the proposed framework enables efficient, proactive resource allocation that maximizes application performance while minimizing operational costs.

This methodology is realized through two novel frameworks, *$\mu$Opt* and *WasteLess*, which apply this common core to the specific domains of microservices and FaaS systems, respectively.

Figure 1 outlines the research trajectory and the progressive refinement of our methodology. The investigation originated with $\mu$Opt, initially designed for vertical scaling in microservices. This foundational core—integrating LQN modeling with fluid approximations—was then adapted to the FaaS domain to realize the first version of WasteLess. Subsequently, both frameworks were significantly expanded: $\mu$Opt was extended to encompass horizontal scaling, while WasteLess was enhanced to handle complex application topologies. These advanced iterations are presented in two journal articles currently under review, collectively bridging the operational gap between dynamic service scaling and serverless resource provisioning.

To provide a high-level structural overview, Figure 2 compares the two frameworks within the unified architecture. While they share the same theoretical core (fluid LQNs and non-linear optimization), the diagram highlights their fundamental difference in operational scope: $\mu$Opt functions as an online, iterative controller for dynamic runtime adaptation (left loop), whereas WasteLess operates as a one-shot, offline provisioner

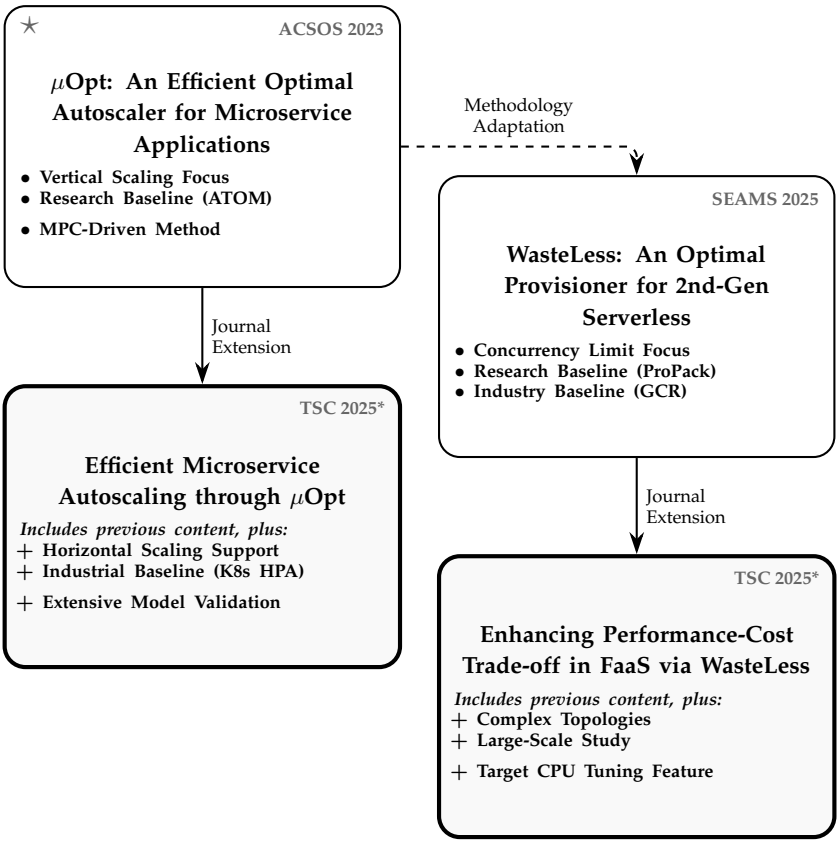



**Figure 1:** Roadmap of the thesis papers. The research originated with $\mu$Opt (Best Paper Award ⋆), focusing on vertical scaling, and was significantly extended in the journal version. The methodology was adapted to the FaaS domain in WasteLess—chronologically positioned between the $\mu$Opt studies—and further extended in a subsequent journal submission. (* Under review).

for deployment-time configuration (right loop).

The specific characteristics and contributions of each framework are detailed below, highlighting how the shared methodological core is adapted to address the distinct constraints of their respective domains.

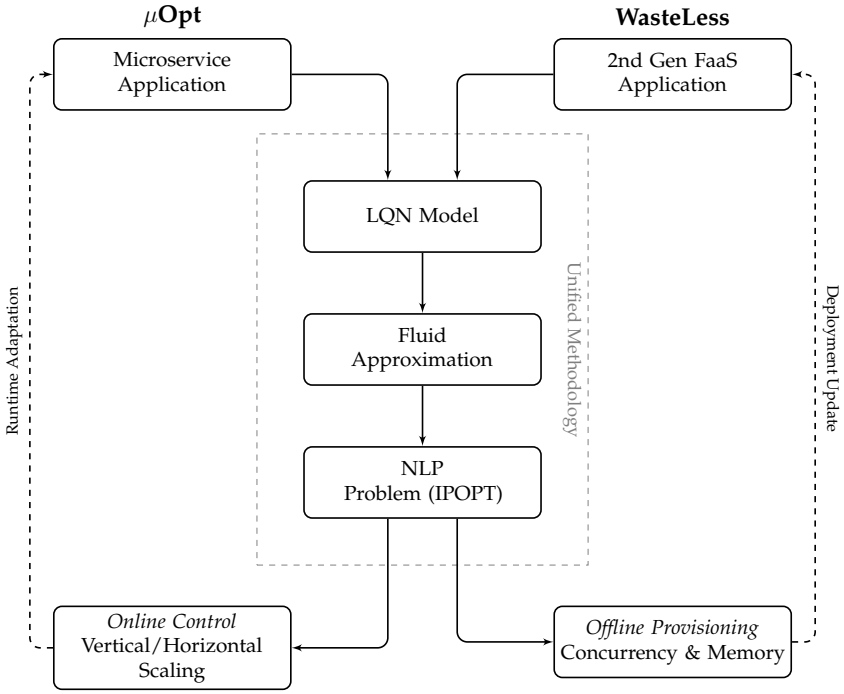



**Figure 2:** Conceptual overview of the unified methodology. The diagram illustrates the two distinct operational loops driven by the shared core. $\mu$Opt (left): An online control loop for dynamic microservice scaling. WasteLess (right): An offline provisioning cycle for FaaS configuration.

## $\mu$Opt: An Efficient Autoscaler for Microservices

In the microservices domain, we introduce $\mu$Opt as the runtime realization of our unified methodology. Functioning as a Model Predictive Control (MPC) loop, the framework iteratively updates the fluid LQN model with live monitoring data and solves the resulting non-linear optimization problem to determine optimal resource allocations. This proactive approach allows $\mu$Opt to anticipate resource needs and adapt to workload fluctuations more effectively than traditional heuristic-based methods.

This work, presented in Chapter 3, is based on the research published in (Incerto, Pizziol, and Tribastone, 2023) and its extended journal ver-

sion (Incerto, Pizziol, and Tribastone, 2025a). The key contributions are:

- The design and implementation of $\mu$Opt, a computationally efficient autoscaler that solves a non-linear optimization problem to minimize operational costs subject to real-time performance constraints.
- A compact encoding of the fluid LQN model as constraints within the optimization problem, enabling rapid exploration of the configuration space.
- A comparative analysis against a state-of-the-art research baseline (ATOM) in a vertical scaling context, demonstrating that the framework is orders of magnitude faster, improves application throughput by up to 9%, and reduces allocated cores by up to 25%.
- A comprehensive validation of the model's accuracy on a production-grade Kubernetes (K8s) cluster using a diverse testbed of randomly generated applications, demonstrating the model's generalizability.
- An extensive benchmark against the industry-standard Horizontal Pod Autoscaler (HPA), showing that the framework reduces resource consumption by up to 30% while strictly adhering to performance targets in a horizontal scaling context.

## WasteLess: An Optimal Provisioner for Second-Generation FaaS

In the serverless domain, we introduce WasteLess as the offline provisioning realization of our unified methodology. Unlike runtime autoscalers, WasteLess leverages the fluid LQN model to perform a one-shot, joint optimization of CPU, memory, and the critical concurrency limit. This approach determines a static, optimal configuration that maximizes instance density while respecting performance constraints, effectively guiding the platform's native scheduler without the overhead of continuous online control.

This work, presented in Chapter 4, is based on the research published in (Incerto, Pizziol, Russo Russo, et al., 2025d) and its extended journal

version (Incerto, Pizziol, Russo Russo, et al., 2025a), which is currently under review. The main contributions are:

- The design and implementation of WasteLess, a resource provisioning framework that automates the complex task of configuring CPU, memory, and concurrency for modern FaaS applications.
- A novel optimization encoding based on the fluid approximation of an LQN model to efficiently determine the optimal balance between performance and cost for a given FaaS application topology.
- A comprehensive evaluation on the Acmeair benchmark, showing that WasteLess reduces operational costs by an average of 38% against non-concurrent deployments without performance loss, while significantly outperforming both commercial (Google Cloud Run (GCR)) and research (ProPack) baselines in terms of latency.
- A large-scale evaluation on complex, randomly generated topologies including asynchronous and parallel execution patterns, demonstrating the framework's robustness and scalability with latency reductions of up to 72% against platform defaults.

## Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 1** provides the foundational background, introducing cloud-native systems, performance modeling with LQNs, the fluid approximation technique, and the optimization context for this research.
- **Chapter 2** reviews the state of the art, discussing related work in the areas of autoscaling strategies for microservices and resource provisioning for FaaS platforms.
- **Chapter 3** presents the $\mu$Opt framework. It describes the framework's architecture, the formulation of its optimization problem,

and an extensive evaluation covering both vertical scaling against research baselines and horizontal scaling against the industry-standard Kubernetes HPA, based on the work in (Incerto, Pizziol, and Tribastone, 2023; Incerto, Pizziol, and Tribastone, 2025a).

- **Chapter 4** introduces the WasteLess framework. It presents the framework's architecture for joint resource optimization and provides a comprehensive evaluation on standard benchmarks and large-scale complex topologies, based on the work in (Incerto, Pizziol, Russo Russo, et al., 2025d; Incerto, Pizziol, Russo Russo, et al., 2025a).

- **Chapter 5** concludes the thesis. It summarizes the key contributions, discusses the limitations and threats to validity of the research, and outlines promising directions for future work.

**Note on Terminology.** To ensure consistency across the diverse domains of queueing theory, microservices, and serverless computing discussed in this thesis, we adopt the following terminology:

- While classical queueing theory often refers to units of work as *jobs*, in the context of web-based cloud-native systems, we consistently use the term *requests* (e.g., HTTP requests). We retain the term *job* only when strictly referencing foundational definitions (e.g., *Little's Law*).

- We use *resources* as a general term encompassing all computational capabilities (e.g., CPU, memory). Conversely, when defining the specific decision variables in our optimization models, we use precise terminology (e.g., *CPU cores* in the context of vertical scaling).

- We use *performance* as a broad category of non-functional requirements. When quantifying performance, we specifically refer to *response time* (i.e., *latency*) and *throughput* as the key metrics. The term *QoS* is used to denote the satisfaction of specific constraints on these metrics (e.g., a response time target).

# Chapter 1

# Background

This chapter provides the foundational background for the research presented in this thesis. We begin in Sections 1.1 and 1.2 by discussing modern cloud-native systems—specifically microservices and FaaS—and highlighting the inherent challenges in their performance management and resource allocation. Section 1.3 then reviews fundamental performance laws and introduces LQNs as a powerful technique for modeling these systems. Following this, Section 1.4 details the fluid approximation method used for their analysis. Finally, in Section 1.5, these models are contextualized within optimization and control frameworks, setting the stage for the solutions developed in this work.

## 1.1 Microservice Architectures

The microservice architecture is a design paradigm that structures an application as a suite of small, autonomous, and loosely coupled services organized around specific business capabilities (Dragoni et al., 2017). These services communicate over a network using lightweight protocols and can be developed, deployed, and scaled individually, thereby offering a robust and flexible foundation for building complex distributed systems (Fowler and Lewis, 2014).

This approach stands in contrast to traditional monolithic design,

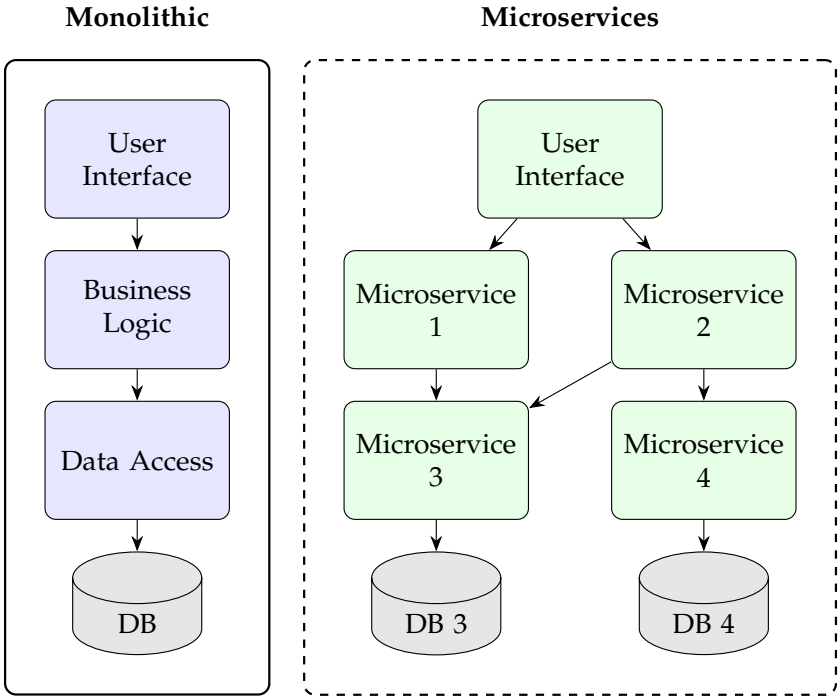



**Figure 3:** A conceptual comparison of monolithic and microservice architectures. Left: The monolithic style tightly couples all application components—user interface, business logic, and data access—within a single deployment unit connected to a unified database. Right: The microservice architecture decomposes the application into a collection of loosely coupled, independently deployable services that communicate over a network, each potentially managing its own dedicated data store.

where all functional components are tightly integrated into a single executable unit.

This architectural shift, illustrated conceptually in Figure 3, transforms the challenge of performance management from optimizing a single entity to managing a complex, distributed ecosystem, where resource allocation becomes a critical factor for both efficiency and cost.

### 1.1.1 Core Principles

The microservice architecture is defined by a set of key principles (Fowler and Lewis, 2014). Each principle offers significant advantages over monolithic design while simultaneously introducing distinct performance and operational challenges that motivate the need for advanced resource management techniques.

- **Compartmentalization:** Each microservice is an autonomous, independently replaceable, and upgradeable component. This modularity enables the independent creation, testing, and deployment of each service, freeing them from the release cycles of other components and thereby accelerating the development process (Blinowski, Ojdowska, and Przybyłek, 2022).

- **Ownership:** Each service assumes exclusive ownership of a distinct business capability, including its logic and data. It acts as the single authoritative source for its specific domain, exercising full control over its own lifecycle and internal implementation. This ensures a clear boundary of responsibility where a service is fully accountable for the end-to-end execution of its designated business function.

- **Decentralization:** A crucial tenet of microservices is the decentralization of technology and data. Developers are free to choose the most appropriate technology stack (programming language, database, etc.) for each service. Each service owns and manages its own database. While this avoids the tight coupling and developmental bottlenecks associated with a single, shared database in a monolith, it introduces challenges in maintaining data consistency across services, often requiring complex solutions (Richardson, 2018).

- **"Smart Endpoints and Dumb Pipes":** Microservices typically communicate using simple, lightweight protocols such as synchronous REST APIs over HTTP or asynchronous messaging. The business logic and intelligence reside within the services themselves (the "smart endpoints"), not in complex middleware (the "dumb

pipes"). This approach simplifies the communication infrastructure but places the burden of handling network latency and ensuring reliable communication on the services themselves (Fowler and Lewis, 2014).

- **Fault-Tolerance:** In a distributed system, the failure of individual components is inevitable. Microservice architectures must therefore be designed for resilience, ensuring that a failure in one non-critical service does not cascade and cause the entire application to fail. This enhances fault isolation but requires the implementation of resilience patterns like circuit breakers, timeouts, and bulkheads to manage partial failures gracefully (Richardson, 2018).

The primary challenge arising from these principles is the significant operational complexity involved in deploying, monitoring, and managing a large ecosystem of distributed services, which stands in stark contrast to the simplicity of managing a single monolithic application (Garriga, 2017).

### 1.1.2 Orchestration with Kubernetes

To manage the operational complexity of a microservice-based system, services are typically packaged as lightweight, portable containers (e.g., Docker[1]), which encapsulate an application and all its dependencies (Rosen, 2013). However, manually managing the lifecycle of hundreds or thousands of such containers is impractical. This necessitates a container orchestration platform, and Kubernetes[2] has emerged as the de facto open-source standard for this task (Jawaddi, Johari, and Ismail, 2022). It provides a declarative framework for automating the deployment, scaling, and lifecycle management of containerized applications. Key Kubernetes concepts relevant to this thesis include:

- **Pod:** The smallest deployable unit in Kubernetes. A Pod encapsulates one or more containers (typically a single container run-

[1]https://docker.com/
[2]https://kubernetes.io/

ning one microservice instance) and is the fundamental unit of scaling (Nguyen et al., 2020).

- **Deployment:** A higher-level object that declaratively manages the state and lifecycle of Pods. It ensures a specified number of identical Pods, referred to as *replicas*, are always running and automates processes like updates and rollbacks (Sun et al., 2023).

Cloud providers offer managed Kubernetes services, such as Google Kubernetes Engine (GKE), which abstract away the complexity of managing the underlying Kubernetes control plane.

### 1.1.3 Autoscaling Microservices

A primary benefit of the microservice architecture is the ability to scale services independently. To leverage this flexibility in dynamic cloud environments, *autoscaling*—the automatic adjustment of computational resources to match demand—is essential for balancing performance and cost (Fetzer, 2016). Autoscaling is typically achieved through two primary dimensions:

- **Horizontal Scaling:** Adjusting the number of running instances (i.e., replicas) of a service.
- **Vertical Scaling:** Modifying the computational resources (e.g., CPU, memory) allocated to each individual service instance.

Effective strategies often combine both approaches to ensure robustness (Incerto, Tribastone, and Trubiani, 2018; Luciano Baresi, Hu, et al., 2021). However, this flexibility transforms resource allocation into a complex, system-wide optimization problem. As the distributed nature and independent scalability of microservices create a vast configuration space, the performance configuration of one service can create cascading effects on others in the call graph (Incerto, Tribastone, and Trubiani, 2017; Razavi et al., 2022). Consequently, the core challenge lies in effectively navigating this complexity to meet global QoS targets (e.g., response time, throughput) without incurring unnecessary expenses (Qu, Calheiros, and Buyya,

2018). This optimization task is non-trivial due to the dynamic and often unpredictable workloads that web applications face, forcing operators to navigate between two undesirable extremes:

**Under-provisioning:** This occurs when an application is allocated insufficient resources to handle the current workload, often in an attempt to minimize operational costs. This leads directly to performance degradation. For instance, consider an online bookstore running a flash sale: if a critical service has too few replicas, it becomes a bottleneck as user traffic spikes. Requests queue up, and the entire application becomes slow or unresponsive, resulting in lost sales and customer dissatisfaction.

**Over-provisioning:** This is the opposite scenario, where an application is allocated far more resources than required. This often occurs as a conservative, static provisioning strategy to ensure availability during peak loads. In the bookstore example, this would involve running enough replicas to handle peak traffic at all times. While this guarantees high performance, it is extremely inefficient; during normal traffic periods, the vast majority of resources sit idle, inflating operational costs without adding value.

Neither of these static extremes is acceptable. An effective resource management strategy must be dynamic, precisely matching resource allocation to workload demand in real-time. This requires a holistic and predictive approach built on a model capable of capturing the intricate performance dependencies between services. Addressing this multifaceted optimization problem is the central motivation for the $\mu$Opt framework presented in Chapter 3.

## 1.2 Function-as-a-Service

Alongside microservices, FaaS has emerged as a core component of the cloud-native paradigm, representing a significant step in the abstraction of server management (Kounev et al., 2023). This evolution is best understood through the shared responsibility model, depicted in Figure 4,

which illustrates the progressive shift of operational burden from the user to the cloud provider.

In the traditional on-premises model, the user maintains full responsibility for the entire technology stack, from physical hardware to the application. The first level of cloud abstraction is Infrastructure-as-a-Service (IaaS), where the provider manages the foundational physical infrastructure while the user remains responsible for the operating system, middleware, and the application itself. Platform-as-a-Service (PaaS) extends this abstraction further, with the provider also managing the operating system and runtime environment. This model has become the predominant deployment target for microservice architectures, which are typically run on managed container orchestration platforms like Kubernetes, as we discussed in Section 1.1.

FaaS represents the highest level of abstraction in this progression. Developers write and deploy individual functions without provisioning or managing any underlying servers, which is why the model is called "serverless". The cloud provider is entirely responsible for executing the code, automatically handling all necessary resource allocation in response to event triggers, such as an HTTP request. This model enables a highly event-driven and scalable architecture where developers are billed only for the actual compute time consumed by their functions, rather than for idle, pre-provisioned servers.

### 1.2.1 Core Principles

The FaaS paradigm can also be defined by a set of core principles that fundamentally distinguish its execution model and operational characteristics from those of long-running microservices.

- **Event-Driven and Stateless:** Functions are executed in response to specific events, and each invocation runs in a separate, stateless container, meaning no data is preserved between executions (Kounev et al., 2023; Y. Li et al., 2022). This ephemeral nature is a key differentiator from microservices, which typically maintain state over their lifecycle.

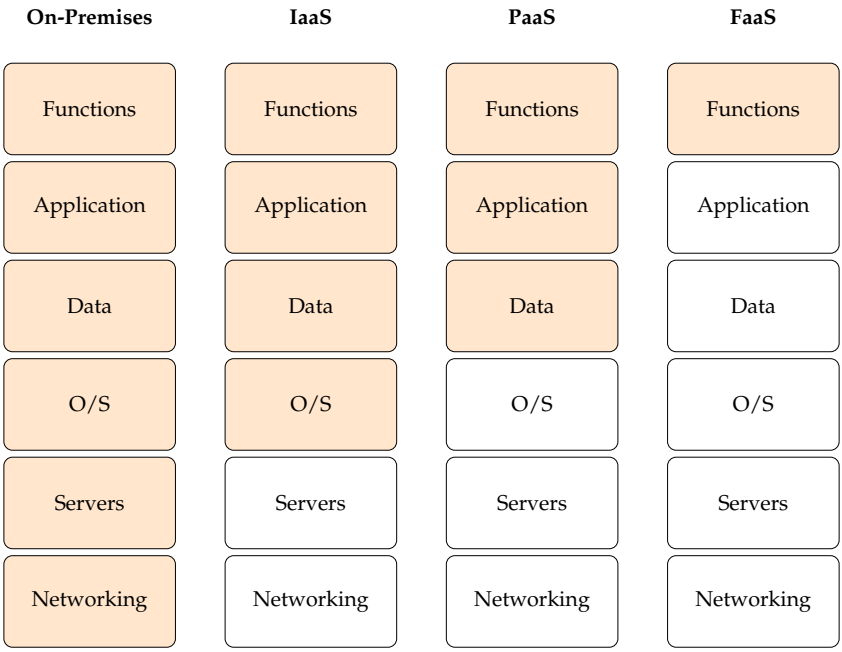



**Figure 4:** The shared responsibility model across different cloud computing paradigms. This diagram illustrates how operational burden progressively shifts from the user (orange blocks) to the cloud provider (white blocks) when moving from On-Premises through IaaS and PaaS, and finally to FaaS. In the FaaS model, the provider manages the entire infrastructure stack, enabling developers to concentrate solely on their function code.

- **Provider-Managed Execution:** The entire execution environment, including scaling, is managed by the cloud provider, who automatically creates new function instances to handle concurrent requests (Kounev et al., 2023). This contrasts with microservice architectures, where scaling logic is typically configured by the developer.

- **Fine-Grained Billing:** Cost is calculated based on the number of invocations and the precise execution duration of a function, often measured in milliseconds (Shahrad et al., 2020; F. Liu and Niu, 2023). This pay-per-use model can be highly cost-effective for applications

with intermittent or unpredictable traffic, unlike the resource-based pricing for the virtual machines that host microservices (Eivy and Weinman, 2017).

### 1.2.2 Compositional Constructs

The behavior of complex FaaS applications is defined by a variety of compositional constructs that determine the application's execution flow and operational logic (Changyuan Lin and Hamzeh Khazaei, 2020).

- **Synchronous Calls:** A synchronous call occurs when a function invokes another function and then pauses its own execution to wait for a response before it can proceed. While this straightforward request-response pattern is simple to structure, it can create performance bottlenecks if the called function is slow, as the caller's resources remain occupied while it waits. This interaction pattern is analogous to synchronous API calls between microservices.
- **Asynchronous Calls:** A function performs an asynchronous call when it invokes another function but does not wait for it to complete before proceeding. Instead, the calling function sends the request and immediately continues its own processing, while the called function executes independently. This non-blocking paradigm is crucial for operations like database updates or image processing, as it allows the caller to remain highly responsive and avoid resource contention.
- **Probabilistic Choice:** This construct models scenarios where a function's execution path is determined dynamically by a probability. After a main function completes, it may invoke one of several potential successor functions with a predefined likelihood. This construct introduces essential variability into the application's flow, which is critical for modeling realistic system behavior.
- **Parallel Calls:** This construct involves a function initiating multiple other functions to execute in parallel. In this thesis, we focus specifically on the fork/join pattern, a common implementation where

an execution path "forks" into several independent paths that run in parallel and later "join" back together. The main execution flow is suspended at the join point until all forked tasks have finished, making this pattern highly effective for workloads that can be split into smaller, independent sub-tasks.

### 1.2.3 Second-Generation FaaS and Concurrency

In the FaaS model, the ephemeral containers that execute the function code are known as instances. While the provider's automatic scaling handles incoming requests by spawning new instances, the key difference between FaaS generations lies in how these instances handle concurrency. In what is known as *First-Generation FaaS*, each instance processes only one request at a time; if multiple requests arrive concurrently, the platform must spawn an equal number of instances. To improve resource efficiency, providers have introduced *Second-Generation FaaS*, which allows multiple requests to be consolidated into a single function instance (Google Cloud, 2025). This capability is controlled by a critical user-configurable parameter called the *concurrency limit*, which defines the maximum number of requests an instance can handle simultaneously.

However, this evolution introduces a complex optimization challenge. The concurrency limit creates a difficult trade-off between resource efficiency and application performance. Setting the limit too low effectively reverts the system to the first-generation model, where concurrent requests trigger new instances, leading to resource underutilization and undermining the cost-saving potential of consolidation. Conversely, setting the limit too high leads to resource contention, as multiple requests compete for the same CPU and memory, causing significant performance degradation (e.g., increased latency).

This trade-off is not always handled effectively by default, self-adaptive mechanisms provided by cloud platforms. Indeed, misconfiguring the concurrency limit can cause adaptive schedulers to prioritize cost savings at the expense of significant performance degradation, as illustrated by the following example.

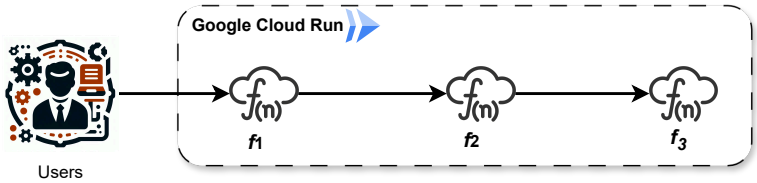



**Figure 5:** Overview of the three-tier FaaS motivating example.

**Table 1:** Average end-to-end response time of the motivating example under two different concurrency limit strategies: no-concurrency and GCR.

| | **no-concurrency** | **GCR** |
|---|---|---|
| Average response time (s) | 2.73 | 15.71 |

**Motivating Example.** Consider a three-tier FaaS application deployed on GCR, consisting of three CPU-bound functions, as depicted in Figure 5. Each function synchronously calls the next, consuming no CPU resources while it waits for a response. Creating additional instances of the caller function results in superfluous resource allocation. Since GCR bills based on the number of active function instances (among other factors like number of invocations, maximum runtime, and storage), users end up facing unnecessarily higher costs. While adjusting the caller function's concurrency limit would suffice, setting it too high can cause resource contention, degrading overall performance.

We evaluated the application under two concurrency limit allocation strategies: no-concurrency and the GCR instance scheduler with its default concurrency limit of 80 [3]. With "no-concurrency" we refer to the *first-generation* approach (concurrency limit of 1). In this simple example, each function was configured with sufficient memory to support the maximum concurrency limit set across all evaluated strategies.

As shown in Figure 6, the GCR instance scheduler achieved a substantial, approximately eight-fold reduction in billable instances compared to the no-concurrency scenario. However, as detailed in Table 1, this

[3] `https://cloud.google.com/run/docs/about-concurrency`

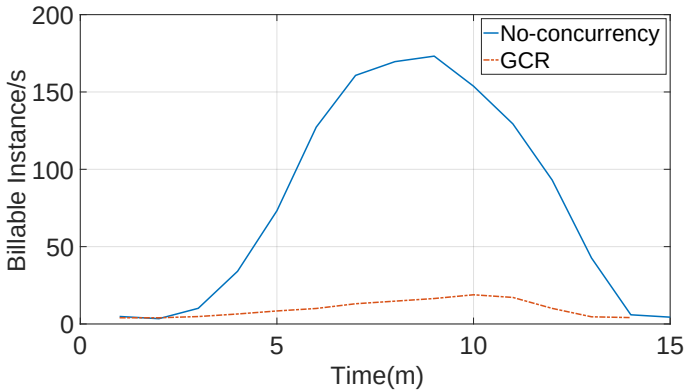



**Figure 6:** Billable instances over time for the motivating example under two strategies: no-concurrency and the default GCR scheduler.

efficiency came at the expense of performance, with the average end-to-end response time increasing by nearly six-fold—a degradation directly attributable to an excessive concurrency limit. This simple example reinforces two core motivations:

- Effective FaaS configuration still relies on user input for selecting the best resource allocation strategy.

- Automated FaaS configurators may result in significant performance degradation if not properly set up.

Finding the optimal configuration for second-generation FaaS applications is thus a multi-dimensional problem that default solutions may fail to solve. Addressing the challenge of jointly optimizing CPU, memory, and the concurrency limit to navigate this fine line is the central goal of the WasteLess framework presented in Chapter 4.

## 1.3 Performance Modeling with Queueing Networks

To analyze the performance of complex, distributed applications, this thesis employs analytical performance models based on *Queueing Theory*. A foundational formalism in this domain is the classical Queueing Network (QN), which models systems as networks of queues and servers processing requests (Bolch et al., 2006). While a comprehensive description of basic QNs goes beyond the scope of this thesis, this section introduces the core modeling principles relevant to our work. We begin by reviewing two fundamental operational laws that govern the behavior of any stable queueing system; these are explicitly presented here as they are frequently employed in our proposed frameworks to derive key performance indices. We then delve into the specifics of LQNs, the powerful modeling formalism (Franks et al., 2008) used in this thesis to capture the intricate interactions found in modern cloud-native systems.

### 1.3.1 Fundamental Performance Laws

Two fundamental operational laws are central to performance analysis in queueing systems: *Little's Law* and the *Utilization Law*. These principles provide simple yet powerful relationships between key performance metrics, allowing for the characterization of system behavior without delving into complex stochastic analysis.

**Little's Law.** Little's Law (Little, 1961; Little and Graves, 2008) establishes a crucial relationship between the average number of jobs in a system, their arrival rate, and their average time spent in the system. For any stable system, the law states that:

$$\mathcal{M} = \mathcal{T}\mathcal{R} \tag{1.1}$$

where $M$ is the average number of jobs in the system (e.g., requests in a queue or being processed), $\mathcal{T}$ is the average arrival rate of jobs (i.e., throughput), and $\mathcal{R}$ is the average time a job spends in the system (i.e.,

response time). The remarkable generality of this law, which holds true independent of the arrival process and service time distribution, renders it an indispensable tool. It is frequently applied within this thesis, for example, to calculate the average response time of a microservice from its measured throughput and the average number of concurrent requests.

**The Utilization Law.** The Utilization Law (Denning and Buzen, 1978) establishes a crucial relationship between the utilization of a resource, its throughput, and the average time required to service a job. For any stable system, the law states that:

$$\mathcal{U} = \mathcal{T}\mathcal{S} \tag{1.2}$$

where $\mathcal{U}$ is the utilization of a resource (i.e., the fraction of time it is busy), $\mathcal{T}$ is once again the average arrival rate of jobs at that resource (i.e., its throughput), and $\mathcal{S}$ is the average time taken to process a single job (i.e., its service time). Resource utilization is a critical metric in cloud-native systems, as it directly relates to both operational cost and potential performance bottlenecks. This law is fundamental to the models developed in this thesis; for example, we use it to connect the throughput of a service to its CPU demand, a relationship that forms a key constraint in our optimization problems.

### 1.3.2 Layered Queueing Networks

Classical QNs are powerful for assessing contention on hardware resources. However, they struggle to represent key aspects of modern software architectures, such as software-level contention (e.g., for finite thread pools) or complex interactions like synchronous blocking calls. In cloud-native systems, a service request often blocks a software thread in the calling service while awaiting a response from a downstream dependency. This "held resource" phenomenon creates a layered contention where software and hardware resources are nested, a defining characteristic that simpler queueing abstractions fail to capture accurately. To address these limitations, we adopt LQNs, which are specifically designed to model

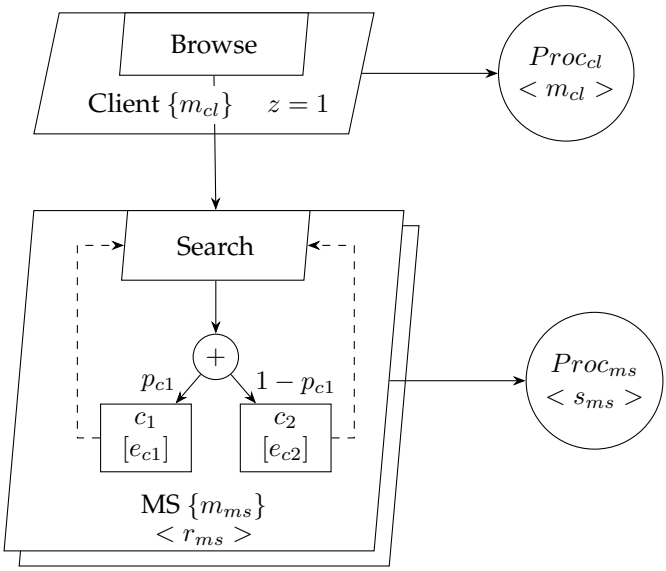



**Figure 7:** A minimal LQN model.

hierarchical dependencies and multi-layered contention. Their ability to capture complex request patterns and client-server interactions makes them uniquely suited for modeling cloud-native architectures (Tribastone, Mayer, and Wirsing, 2010).

An LQN model represents a system using four primary components: *tasks*, *processors*, *entries*, and *activities*, as illustrated in the example in Figure 7.

**Tasks.** A *task*, shown as a parallelogram, represents a logical service provider, such as a microservice container or a FaaS function instance (e.g., the `MS` task in Figure 7). It is defined by a *multiplicity* (in curly brackets, e.g., $m_{ms}$), which represents the thread-pool size of a microservice or the concurrency limit of a function, and a *replication factor* (in angular brackets, equal to one if omitted), which denotes the number of identical instances (e.g., Kubernetes pods). A special *reference task* (e.g., `Client`) is included to model the user population, where its multiplicity ($m_{cl}$)

corresponds to the number of concurrent users. This task models a closed workload, where users issue requests and then pause for an exponentially distributed *think time* (with average $z$) before repeating the cycle.

**Processors.** Each task is mapped to a *processor* (a circle, e.g., $Proc_{ms}$), which models the underlying hardware resources (e.g., allocated CPU cores) with its own multiplicity ($s_{ms}$) specifying the number of available cores. To model the independent operation of users on their own devices, the processor for the reference task (e.g., `Client`) is assigned a multiplicity equal to the number of users ($m_{cl}$).

**Entries.** An *entry* (a small parallelogram nested within a task) represents a specific service or API endpoint, such as the `Search` entry in the `MS` task. Its behavior is defined by a Directed Acyclic Graph (DAG) of *activities* and *control nodes*.

**Activities and Control Nodes.** An activity, depicted as a rectangle, represents a discrete processing step within the service logic (e.g., $c_1$). Each activity is characterized by a service time (in square brackets, e.g., $[e_{c1}]$), representing the average processor demand.

The execution flow between activities is governed by *control nodes*:

- **AND-nodes (&):** Indicate concurrent execution branches.
- **OR-nodes (+):** Represent probabilistic branching. For example, the `Search` entry utilizes an OR-node to select between activities $c_1$ and $c_2$ based on defined probabilities.

Note that for entries consisting of a single step, the activity is implicit, and the service time is annotated directly within the entry.

**Interactions.** Interactions between components, such as HTTP calls, are modeled as *requests* (directed arcs). These can be *synchronous* (closed arrowheads), where the caller blocks until it receives a response, as seen in the request from the `Client` to the `MS` task, or *asynchronous* (open arrowheads), where the caller proceeds immediately.

## 1.4 The Fluid Approximation

The analytical solution of LQN models faces a significant computational barrier known as the *state-space explosion* problem (Franks et al., 2008). Since the underlying stochastic process is a Continuous-Time Markov Chain (CTMC), the state space grows exponentially with the system size. This combinatorial growth makes exact analysis computationally intractable for large-scale cloud applications.

To address this, we employ the *fluid approximation*. This technique reinterprets the discrete-state stochastic process as a continuous deterministic model, described by a system of coupled Ordinary Differential Equations (ODEs) (Kurtz, 1970; Tribastone, 2012). Instead of tracking individual requests, the fluid model tracks their average density over time. This shift offers two decisive advantages for our optimization framework:

- **Scalability:** The size of the ODE system depends only on the number of distinct components, not on the population levels (e.g., users or replicas), making it independent of the scale of the deployment.

- **Efficiency:** The solution of the ODE system provides the time-course evolution of the average system behavior. Steady-state analysis is reduced to finding the equilibrium point where time derivatives are zero ($\dot{x} = 0$), which is computationally inexpensive compared to solving the steady-state probability vector of a massive CTMC.

### 1.4.1 Derivation and Dynamics

The derivation of the fluid model from an LQN is formally achieved by translating the network into a stochastic process algebra, such as PEPA (Hillston, 1996; Tribastone, 2012), from which the system of coupled ODEs is systematically derived. In this thesis, we use DiffLQN (Waizmann and Tribastone, 2016) to automate this generation process. This tool parses the LQN model and efficiently synthesizes the corresponding differential equations, enabling the analysis of complex topologies where manual derivation would be error-prone or infeasible.

Conceptually, the resulting ODEs represent a flow balance for every state in the system:

$$\dot{x}_i(t) = \sum \text{Inflow}_i(x) - \sum \text{Outflow}_i(x) \tag{1.3}$$

where $x_i(t)$ is the continuous variable representing the average number of requests at location $i$ (e.g., queuing at a service or being processed). Crucially for our optimization goals, resource capacities (such as CPU cores) are treated as continuous variables in the flow rates rather than discrete integers. This continuous interpretation allows the model to represent fractional allocations and renders it mathematically suitable for gradient-based optimization solvers.

### 1.4.2 Performance Indices Computation

The optimization frameworks in this thesis rely on the steady-state solution $x^*$, obtained by solving the algebraic system $\dot{x} = 0$. Key performance metrics are derived directly from this equilibrium point. For a generic service entry, the throughput $\mathcal{T}$ corresponds to the steady-state flow rate, while the average response time $\mathcal{R}$ is derived via Little's Law (Equation (1.1)). Similarly, resource utilization is computed as the ratio of utilized capacity to total allocated capacity.

### 1.4.3 Limitations of Fluid Models

While powerful, the fluid approximation involves inherent trade-offs that define the boundaries of its applicability. Specifically, we identify three key limitations:

- By definition, the fluid model captures only the *mean* trajectory of the system. It effectively filters out stochastic noise, meaning it cannot predict variance or tail latencies (e.g., 99th percentile response times), which are often critical for strict SLO definitions.

- The approximation becomes asymptotically exact as the population size tends to infinity (Kurtz, 1970). Consequently, the model's

accuracy degrades in scenarios with small populations (e.g., very few concurrent users). In these regimes, discrete stochastic fluctuations become dominant, and the smooth fluid dynamics may fail to capture the bursty nature of request arrivals.

- The standard fluid formulation does not natively model discrete, non-linear events such as cold starts. As a result, the model overlooks the initial latency spike, causing a deviation from real-world performance during this phase.

In this thesis, we address these limitations through a pragmatic design choice: we trade the probabilistic estimation of risk for the computational speed required by real-time control. While the optimization objectives rely on mean performance metrics, we postulate that improving average system behavior correlates with a reduction in tail latency and an indirect mitigation of transient anomalies. To validate this assumption, our experimental evaluation is conducted on real-world platforms where stochastic fluctuations and discrete events are fully present, ensuring that the model's theoretical simplifications do not compromise the effectiveness of the resulting resource management strategies.

## 1.5 Optimization

The core of this thesis relies on formulating resource management as an optimization problem. The goal is to determine the optimal allocation of resources (e.g., CPU cores, memory, service replicas) that satisfies specific performance objectives—such as minimizing response time or maximizing throughput—while strictly adhering to operational constraints. Due to the complex, non-linear relationships between resource allocation and system performance metrics in cloud-native applications, these problems inherently fall into the domain of Non-Linear Programming (NLP).

### 1.5.1 Non-Linear Programming

An NLP problem seeks to optimize a non-linear objective function subject to a set of non-linear equality and inequality constraints. A general formulation is:

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{1.4}$$

$$\text{s.t.} \quad g_i(x) = 0, \quad i \in \mathcal{E} \tag{1.5}$$

$$h_j(x) \geq 0, \quad j \in \mathcal{I} \tag{1.6}$$

where $x \in \mathbb{R}^n$ is the vector of decision variables, $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $\mathcal{E}$ and $\mathcal{I}$ are the index sets for equality and inequality constraints, and $g_i(x)$ and $h_j(x)$ represent the corresponding constraint functions. Finding a global optimum for such problems is computationally intractable in general (Murty and Kabadi, 1985); therefore, algorithms typically aim to find a locally optimal solution efficiently.

From a computational perspective, solving general NLP problems is known to be NP-hard (Wright, 2005). This complexity is exacerbated in the context of performance models like LQNs, where the mapping from resources to queue lengths results in non-convex constraints. Consequently, effectively managing this complexity requires robust algorithms specifically designed to handle large-scale non-linear constraints.

**The Interior Point Optimizer (IPOPT).** To solve the complex NLP problems that arise from our performance models, this thesis utilizes the Interior Point Optimizer (IPOPT) solver (Wächter and Biegler, 2006). IPOPT is an open-source software package designed for large-scale non-linear optimization. It employs a primal-dual *interior-point* method to handle inequality constraints (Boyd and Vandenberghe, 2004).

The core idea of this technique is to transform the original constrained problem into a series of simpler, unconstrained (or equality-constrained) subproblems. This is achieved by augmenting the objective function with a logarithmic barrier term that penalizes solutions approaching the boundary of the feasible region. For example, the inequality constraints

$h_j(x) \geq 0$ are incorporated as:

$$\min_x \quad \varphi_\mu(x) = f(x) - \mu \sum_{j \in \mathcal{I}} \ln(h_j(x)) \tag{1.7}$$

where $\mu > 0$ is the barrier parameter. As $\mu$ is gradually driven to zero, the solutions of the subproblems converge to the solution of the original constrained problem. Each of these subproblems is then solved iteratively using a variant of Newton's method. Specifically, rather than minimizing $\varphi_\mu(x)$ directly, IPOPT solves the Karush-Kuhn-Tucker (KKT) optimality conditions for the barrier subproblem. At each iteration, the algorithm computes a search direction by solving a large linear system involving the Hessian of the Lagrangian and the Jacobian of the constraints. Crucially, IPOPT exploits the sparse structure of these matrices, ensuring that the solver scales efficiently even as the system size increases. Furthermore, to guarantee global convergence from arbitrary starting points, IPOPT employs a filter line-search method. These features make IPOPT particularly robust and efficient for the types of problems addressed in this work.

**Solver Limitations and Engineering Trade-offs.** While gradient-based solvers like IPOPT offer significant efficiency advantages for large-scale optimization, their application to cloud resource management introduces specific trade-offs. First, the constraints derived from fluid LQN models are generally non-convex. Consequently, gradient-based methods guarantee convergence only to a local optimum rather than a global one. However, in the context of runtime autoscaling, finding a high-quality feasible solution within a strictly bounded time window is often prioritized over the computationally prohibitive search for global optimality. Global search algorithms (e.g., genetic algorithms) typically require numerous function evaluations, rendering them too slow for real-time control loops. In contrast, local, gradient-based methods typically converge in a modest number of iterations in practice, making them the preferred choice for the time-triggered optimization approach adopted in this thesis. Second, cloud resources such as replicas and threads are inherently discrete, whereas NLP solvers operate in a continuous domain.

This necessitates a continuous relaxation of integer decision variables, requiring a subsequent discretization step (e.g., rounding) that may theoretically induce sub-optimality. Finally, the system dynamics involve non-differentiable functions (specifically the `min` operator used to model resource contention), which requires the introduction of smooth approximations (e.g., soft minimums) to ensure the differentiability required by the solver's Jacobian and Hessian computations.

# Chapter 2

# Related Work

This chapter reviews the state of the art in resource management for cloud-native systems, a critical challenge that directly impacts both application performance and operational costs. By surveying the existing literature through the lens of the theoretical foundations established in Chapter 1, we contextualize the contributions of this thesis.

The review is structured around our two primary areas of focus: Section 2.1 examines autoscaling strategies for microservice architectures, while Section 2.2 investigates resource provisioning for serverless platforms, with particular attention to the unique challenges of second-generation FaaS systems. Finally, Section 2.3 provides a systematic synthesis of the resource management design space, establishing the criteria used to contrast existing approaches with the proposed fluid modeling methodology and highlighting gaps in computational efficiency and expressiveness.

## 2.1 Autoscaling Strategies for Microservices

The independent scalability of each service is a core advantage of the microservice architecture. However, as detailed in Section 1.1, this flexibility creates a vast configuration space governed by complex stochastic dynamics. Consequently, developing effective autoscaling techniques

has been a central research challenge since the paradigm's inception. Autoscaling, which is the automatic adjustment of computational resources to match demand, is essential for balancing application performance with operational costs in dynamic cloud environments. While the fluid approximation presented in Section 1.4 offers a scalable deterministic solution to this problem, this section surveys the landscape of alternative autoscaling solutions, which can be broadly categorized as *white-box* (explainable) or *black-box* (data-driven) (Qu, Calheiros, and Buyya, 2018).

**White-Box Approaches.** White-box strategies often rely on performance models, such as QNs, to make scaling decisions. Some solutions utilize simpler QNs combined with heuristics (T. Chen and Bahsoon, 2015), while others employ more complex models (Moreno et al., 2015; Gandhi et al., 2014; Incerto, Tribastone, and Trubiani, 2018; Tong et al., 2021; Barna et al., 2018; Amiri et al., 2022). However, these approaches typically rely on exact analytical solutions or simulations. As discussed in Section 1.4, such techniques suffer from the state-space explosion problem when applied to large-scale systems. Consequently, they are often limited to smaller topologies or incur significant computation times, failing to meet the strict timing constraints required for effective real-time control.

A prominent LQN-based autoscaler, ATOM (Alim Ul Gias, Giuliano Casale, and Woodside, 2019), relies on a genetic algorithm, which can introduce slow convergence times. The $\mu$Opt framework overcomes these limitations by leveraging LQNs with a fluid approximation, enabling efficient analysis and accurate modeling of complex interactions like synchronous requests. For an extensive numerical comparison of $\mu$Opt against ATOM see Section 3.2.1.

Lastly, rule-based strategies constitute a significant category of autoscaling approaches (Gotin et al., 2018; Florio and Di Nitto, 2016; Hamzeh Khazaei et al., 2017; Toffetti et al., 2017; Hossen, Islam, and Ahmed, 2022). These methods typically require the manual definition of scaling logic, whereas $\mu$Opt automatically synthesizes optimal scaling decisions directly from the performance model.

**Black-Box Approaches.** Many other autoscalers utilize data-driven, black-box performance models (Grimaldi et al., 2015; Wajahat et al., 2019; Barrett, Howley, and Duggan, 2013; Iqbal, Dailey, and Carrera, 2015; Qiu et al., 2020; Rossi, Cardellini, and Presti, 2020; Yu, P. Chen, and Zheng, 2019; Khaleq and Ra, 2021; Meng, Tong, et al., 2022; Z. Wang et al., 2022; J. Liu, Zhang, and Q. Wang, 2023; Bai et al., 2024).

Most of these employ Machine Learning (ML) techniques to dynamically adapt the system based on changing workloads. Methods such as regression (Grimaldi et al., 2015; Wajahat et al., 2019), Reinforcement Learning (RL) (Barrett, Howley, and Duggan, 2013; Iqbal, Dailey, and Carrera, 2015; Qiu et al., 2020; Rossi, Cardellini, and Presti, 2020; Khaleq and Ra, 2021; J. Liu, Zhang, and Q. Wang, 2023; Bai et al., 2024), or Deep Learning (DL) (Meng, Tong, et al., 2022; Z. Wang et al., 2022; Meng, Song, et al., 2023) are commonly used. For example, FIRM (Qiu et al., 2020) is a resource management framework that adaptively identifies the microservices responsible for SLO violations and resource underutilization, employing a two-level ML model. A more recent example, µConAdapter (J. Liu, Zhang, and Q. Wang, 2023), also uses RL to specifically address the challenge of fast concurrency adaptation for microservices. More advanced DL models, such as DeepScaler (Meng, Song, et al., 2023), use spatiotemporal graph neural networks to discover latent dependencies for holistic scaling.

Although these methods can be effective, their training may be time-consuming and require a large amount of data (e.g., $\sim 10^3$ GB (Qiu et al., 2020)). Since scaling actions are learned from data, they are often not explainable. In contrast, $\mu$Opt requires minimal data for model calibration and provides explainable scaling decisions based on queuing theory.

## 2.2 Resource Provisioning for FaaS Platforms

As detailed in Section 1.2, the performance and economic benefits of the FaaS model are contingent upon the precise configuration of function resources. This creates a significant provisioning challenge, particularly in second-generation platforms where concurrency plays a pivotal role in

the cost-performance trade-off. This section reviews the state of the art in this domain, distinguishing between approaches that optimize specific subsets of parameters and more holistic strategies.

Very close to our works, Lin et al. consider the problem of modeling and optimizing performance and cost of FaaS applications for the mean (C. Lin and H. Khazaei, 2021) and the distribution (C. Lin, N. Mahmoudi, et al., 2023) of response times using an extension of Stochastic Petri Nets (SPNs), with the objective to optimize memory allocation. In contrast, WasteLess extends this scope by optimizing additional parameters, i.e., CPU cores and concurrency limit, alongside memory.

Most existing provisioning frameworks focus primarily on memory allocation (Akhtar et al., 2020; Eismann et al., 2021). Examples include COSE (Akhtar et al., 2020), which uses Bayesian optimization, Sizeless (Eismann et al., 2021), which employs a multi-target regression neural network, and Orion (Mahgoub, Yi, Shankar, Elnikety, et al., 2022), which optimizes memory configurations with respect to response time distributions. However, the shift to second-generation FaaS introduces the concurrency limit as a critical performance knob. Ignoring this parameter fails to address the resource contention dynamics that are explicitly captured by our LQN formulation.

Another line of research focuses on *function fusion*, where individual tasks are bundled into a single deployed function to reduce overheads. Elgamal et al. introduce this in Costless (Elgamal, 2018) to minimize cost by fusing, placing, and allocating memory to functions. Similarly, WiseFuse (Mahgoub, Yi, Shankar, Minocha, et al., 2022) uses profiling runs to suggest an optimized DAG of fused functions to the user. Recently, FUSIONIZE++ (Schirmer et al., 2024) proposed a feedback-driven framework that uses live monitoring data instead of profiling runs to automate this process. It dynamically fuses synchronous calls and optimizes the resulting functions' infrastructure configurations. Unlike WiseFuse, it does not require the application to be structured as a DAG and can handle arbitrary call graphs. While these approaches restructure the application's deployment topology, WasteLess focuses on optimally configuring the resources for a given, fixed topology. Roy et al. (Roy et al., 2023) present

ProPack, a solution that "packs" multiple functions instances into a single container and executes them concurrently without an explicit support of the underlying FaaS framework.

To choose the optimal concurrency limit, ProPack constructs an approximate model of expected response time and cost for varying concurrency limit of a single function. However, it is affected by two major limitations: (i) it neglects the topology of the entire application, a crucial aspect considered in our work, (ii) it leaves the CPU and memory sizing of FaaS functions still under designers' responsibility. For an extensive numerical comparison of WasteLess against ProPack see Section 4.2.1.

Another line of work focuses on runtime configuration using learning-based methods. For instance, FaaSConf (Y. Wang et al., 2024) uses Multi-Agent Reinforcement Learning (MARL) to continuously tune resources for serverless workflows at runtime. This approach contrasts with ours: WasteLess computes its configurations only once by leveraging an analytical performance model, deliberately avoiding the overhead associated with continuous online learning and adaptation.

To our knowledge, in the context of FaaS LQNs have been exploited only in COCOA (A. U. Gias and G. Casale, 2020). Differently from our work, the authors consider a capacity planning problem, where they optimize the amount of resources allocated to the whole FaaS system (i.e., number of CPUs and memory) and the "idle time" of each function (i.e., the maximum time an idle container of a function is kept alive before being terminated). In the cloud scenario we target, these parameters are not exposed to us, and we instead focus on (second-generation) function-level resource configuration, including concurrency limit.

Other analytical models have focused on first-generation FaaS platforms. For instance, Mahmoudi et al. (Nima Mahmoudi and Hamzeh Khazaei, 2020) present a performance model for *scale-per-request* systems, such as AWS Lambda, where no request queuing occurs. Their model, based on a Semi-Markov Process, accurately predicts metrics like the cold start probability and the number of idle instances. The authors leverage this model to perform what-if analysis on the *expiration threshold* of idle instances to study its impact on performance and cost. This differs

from WasteLess, which is designed for second-generation platforms and optimizes user-configurable parameters, including the concurrency limit.

While function sizing has been regarded as a key issue in the adoption of cloud-based FaaS, it is worth remarking that other performance issues—which are out of the scope of this work—have received significant attention from researchers, including, e.g., auto-scaling (L. Baresi et al., 2022; Qian et al., 2022; X. Li et al., 2022), scheduling (L. Baresi et al., 2022; Das et al., 2020; Tariq et al., 2020; Russo et al., 2022), and cold start reduction and mitigation (Z. Li et al., 2022; J. Shen et al., 2021; Silva, Fireman, and Pereira, 2020; Daw, Bellur, and Kulkarni, 2020; Oakes et al., 2018). Other works have focused on tackling bottlenecks related to ephemeral storage (Klimovic et al., 2018) and networking to enable high-performance serverless computing (Akkus et al., 2018). Our approach is complementary to these.

Finally, it is worth noting that concurrency tuning for FaaS applications shares conceptual similarities with soft resource allocation for microservices, as discussed in Section 2.1. However, the two problems exhibit fundamental differences. Unlike long-running microservices, FaaS applications are characterized by a unique pricing model and transient dynamics—such as cold starts—that significantly alter the optimization landscape. These specific constraints necessitate the dedicated provisioning approach proposed in WasteLess.

## 2.3 The Resource Management Design Space

The review presented in the preceding sections illustrates that the landscape of resource management for cloud-native systems encompasses a diverse array of modeling and control paradigms. To systematically contrast these methodologies, this section provides a synthesis of the design space based on three key requirements derived from the performance challenges of modern microservice and serverless architectures:

- *Global Coordination (GC)*: The capacity for multi-tier optimization that considers the entire application topology to prevent the inad-

**Table 2:** Systematic comparison of the resource management design space.

| **Methodology** | **GC** | **EX** | **LC** |
|---|---|---|---|
| Heuristic/Rule-Based | × | ✓ | × |
| Control-Theoretic | ✓ | ✓ | × |
| Learning-Based | ✓ | × | ✓ |
| Simple Queueing | ✓ | ✓ | × |
| Analytical (LQNs) | ✓ | ✓ | ✓ |
| **Proposed (Fluid LQN)** | ✓ | ✓ | ✓ |

vertent migration of performance bottlenecks.

- *Explainability (EX)*: The provision of traceable scaling decisions rooted in performance theory, ensuring that resource allocations are transparent, justifiable, and consistent with the application's structural properties.

- *Layered Contention (LC)*: The ability to explicitly model software-level blocking (e.g., thread pool limits) and nested resource sharing, which are primary performance drivers in multi-tier applications.

Table 2 summarizes how the primary classes of approaches address these requirements compared to the fluid LQN methodology proposed in this thesis.

While heuristic and rule-based methods are explainable and industrially common, they fail to capture the complex inter-service dependencies prevalent in microservices. Control-theoretic approaches provide stability but are often error-driven and lack an architectural model of the system, making them less effective at pinpointing the root causes of performance degradation. While learning-based techniques offer significant flexibility, they often lack the explainability required for production troubleshooting, effectively behaving as opaque "black boxes." Furthermore, these approaches are typically characterized by protracted bootstrapping phases, necessitating extensive data collection or exploration before reaching convergence.

Analytical models, such as QNs and LQNs, satisfy the requirements for coordination and explainability by providing a formal mathematical representation of system behavior. However, while simple QNs lack the necessary detail to capture nested software-level contention, more expressive models like LQNs allow for the explicit representation of layered contention. Traditionally, the use of LQNs for real-time control has been limited by the *state-space explosion problem* in large-scale systems. By leveraging the fluid approximation, the methodology proposed in this thesis maintains the architectural depth and transparency of LQNs while achieving the computational tractability necessary for dynamic cloud-native environments. This combination allows for proactive management that is both theoretically sound and practically feasible.

# Chapter 3

# $\mu$Opt

This chapter introduces $\mu$Opt, a novel model-based autoscaler designed to address the complex resource management challenges in microservice architectures.

The chapter begins in Section 3.1 by presenting $\mu$Opt. We detail its architecture, including the design-time and run-time components, and the formulation of the core optimization problem designed to ensure high QoS while minimizing waste. Subsequently, Section 3.2 provides a comprehensive, three-part evaluation. We first benchmark $\mu$Opt against the state-of-the-art research autoscaler ATOM in a vertical scaling context, which also serves to validate the model's accuracy on the complex Acmeair application. Second, we present an extensive validation of the model's accuracy and efficiency in a horizontal scaling context, testing it against a diverse testbed of ten randomly generated applications. Finally, having established the model's robustness, we compare $\mu$Opt against the production-grade Kubernetes HPA. The chapter concludes in Section 3.3 with a summary of the key findings.

## 3.1 The $\mu$Opt Framework Architecture

This section provides a detailed discussion of the $\mu$Opt autoscaler. To illustrate its features independently of any specific technology, we apply it

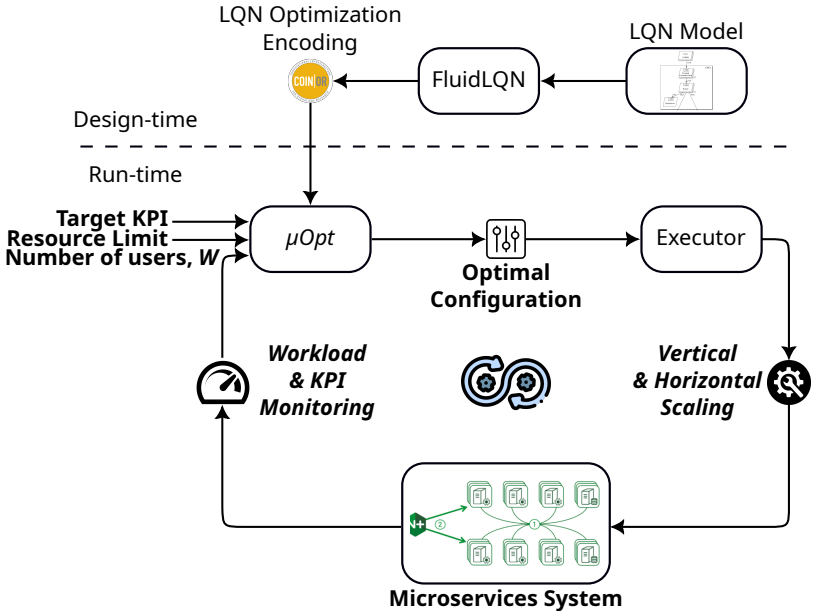



**Figure 8:** Overview of $\mu$Opt.

to an abstract microservices system, which allows us to monitor the total number of users and response time, throughput and CPU usage for each microservice. We also assume the presence of an API for horizontal and vertical scaling, which will be further discussed in Sections 3.2.1 and 3.2.2. Figure 8 presents an overview of $\mu$Opt.

At design time, the process begins with defining the system's LQN model. This can be achieved through model-to-model transformations from UML design artifacts (Petriu and H. Shen, 2002) or by analyzing communication patterns between microservices, which can be discovered using Application Performance Monitoring (APM) tools (Cerny et al., 2022). Average service times can then be estimated using the average response time and throughput data collected by these APM tools, in conjunction with Little's law.

Next, we formulate a nonlinear optimization problem based on the fluid representation of the LQN. This problem incorporates the LQN's

ODEs as constraints, with the decision variables being the processor multiplicities and task thread pool sizes. We automatically generate the LQN's ODEs using a state-of-the-art tool, DiffLQN (Waizmann and Tribastone, 2016).

At runtime, the optimization problem is updated at user-defined intervals with the measured number of users ($W$). It is then solved to adjust the system and achieve the desired Key Performance Indicators (KPIs), such as throughput, response time or CPU usage, while minimizing resource allocation. This approach aligns the autoscaling logic of $\mu$Opt with the well-known time-triggered MPC approach (Garcia, Prett, and Morari, 1989).

Finally, the executor translates the computed processor multiplicities and threadpool sizes into corresponding scaling actions.

In the following, we detail the optimization problem and the autoscaling logic of $\mu$Opt.

### 3.1.1 The $\mu$Opt Optimization Problem

To maximize efficiency, $\mu$Opt formulates optimal autoscaling within the differentiable nonlinear optimization framework (Wright, 2005). Its expressiveness enables encoding optimization problems that adapt to various constraints and requirements.

We now present a generic optimization template that is customizable for different autoscaling scenarios:

$$\max_{s,m} \quad \psi f_k(s,m,x,\theta) - (1-\psi) f_r(s,m) \tag{3.1}$$

$$\text{s.t.} \quad \dot{x} = 0 \tag{3.2}$$

$$\sum_l x_l = W \tag{3.3}$$

$$\mathcal{T}(s,m,x,\theta) \geq \underline{\mathcal{T}} \tag{3.4}$$

$$\mathcal{R}(s,m,x,\theta) \leq \overline{\mathcal{R}} \tag{3.5}$$

$$\mathcal{U}(s,m,x,\theta) \geq \underline{\mathcal{U}} \tag{3.6}$$

$$\mathcal{U}(s,m,x,\theta) \leq \overline{\mathcal{U}} \tag{3.7}$$

$$0 \leq s_p \leq \overline{s}_p \quad \forall p \in Proc \tag{3.8}$$

$$0 \leq m_t \leq \overline{m}_t \quad \forall t \in Task \tag{3.9}$$

Here, $Proc$ and $Task$ are the sets of all processors and tasks of the LQN, respectively; $s = (s_p)_{p \in Proc}$, $m = (m_t)_{t \in Task}$ are the vector of processors' multiplicities and threadpool sizes; $x$ is the vector of queue lengths at any location of the LQN model, with the queue length at generic location $l$ being denoted by $x_l$; $\theta$ is the vector of LQN parameters, i.e., the service times and branch probabilities.

The objective function (3.1) combines two conflicting sub-objectives: maximizing the KPIs $f_k$ and minimizing allocated resources $f_r$. Each sub-objective is governed by a weight $\psi$, where $0 \leq \psi \leq 1$. While tuning this weight has been well-studied in multi-objective optimization (Marler and Arora, 2010), there is no universal solution, as it varies based on the application and user needs.

Equations (3.2)–(3.3) embed the steady-state solution of an LQN model into the optimization problem. Equation (3.2) enforces the steady state of the ODEs, while Equation (3.3) ensures that the total number of users in the model matches the measured value, $W$. Although constraint (3.3) is essential for adequately formulating the ODE system, it does not limit $\mu$Opt's applicability. This information can be gathered through minimally invasive methods, like tracing arrivals and completions or applying Little's law to the throughput and response time of the reference station (Awad and Menasce, 2017).

Equations (3.4)–(3.7) are nonlinear constraints that set upper and lower bounds on the steady-state response time, throughput and CPU usage of the LQN entries. The performance indices $\mathcal{R}$, $\mathcal{T}$, and $\mathcal{U}$ are derived from the steady-state solution by applying the fundamental performance laws described in Section 1.3.1. $\overline{\mathcal{R}}$, $\underline{\mathcal{T}}$, $\overline{\mathcal{U}}$, and $\underline{\mathcal{U}}$ represent user-defined limits for response time, throughput and CPU usage, respectively. Lastly, constraints (3.8)–(3.9) impose bounds on processor multiplicities and threadpool sizes.

From a computational standpoint, solving (3.1)–(3.9) to global optimality is difficult due to the highly nonlinear and non-differentiable relationships expressed in constraints (3.2)–(3.7) between LQN performance indices and their parameters. $\mu$Opt focuses on local optimality by employing the IPOPT solver introduced in Section 1.5, which necessitates a differentiable non-linear programming formulation. Therefore, we transform the constraints in (3.1)–(3.9) by relaxing integer variables to the continuous domain and approximating non-differentiable functions. For instance, the $\min(x, s)$ operator is replaced by the smooth approximation $\frac{x+s-\sqrt{(-x+s)^2+10^{-5}}}{2}$. The resulting continuous solutions are subsequently discretized by the $\mu$Opt executor (see Figure 8) before being applied to the running system.

### 3.1.2 The $\mu$Opt Autoscaling Logic

Upon each invocation, $\mu$Opt executes a three-step control loop:

1. **Update:** The optimization problem is updated with the most recent monitoring data, specifically the current number of users ($W$).
2. **Solve:** The non-linear problem defined in Equations (3.1)–(3.9) is solved to determine the optimal processor multiplicities ($s$) and thread pool sizes ($m$).
3. **Apply:** The computed configuration is enacted on the running microservices through the executor.

The first two steps are primarily computational, involving data aggregation—such as parsing arrival and completion logs—and numerical optimization.

The third step, however, relies on the executor depicted in Figure 8 to enforce the decisions.

This component is designed to be versatile, supporting either vertical or horizontal scaling depending on the operational context. The specific implementation details for these distinct actuation strategies are provided in the experimental sections: vertical scaling is addressed in Section 3.2.1, while horizontal scaling is detailed in Section 3.2.2.

### 3.1.3 Operational Lifecycle

Microservice-based applications are living artifacts that evolve through frequent updates, service refactoring, and architectural shifts. Consequently, the operational deployment of $\mu$Opt is conceived as a continuous process rather than a static configuration.

This lifecycle begins with an initial bootstrapping phase, where the design-time LQN model is parameterized through baseline profiling. Notably, this represents a one-time engineering investment to establish the performance relationships between the various microservices. Although the model is defined upfront, it is not a rigid artifact; it can be updated and recalibrated to reflect the evolving state of the infrastructure or the application logic.

It is essential for the user to consider rerunning this calibration and updating the model whenever the application undergoes significant changes, such as the introduction of new services or major logic updates. While minor code changes may not always necessitate a full model refresh, users should closely monitor telemetry data (e.g., via Prometheus) to detect model drift and determine if the existing parameters still capture the system's performance accurately.

In the current implementation of the framework, this maintenance cycle must be handled manually, though it establishes the necessary foundation for the future automated modeling integration discussed in Section 5.3.

## 3.2 Evaluation and Results

This section presents a comprehensive, two-part evaluation to validate the $\mu$Opt framework, addressing the two primary scaling dimensions: vertical and horizontal.

First, in Section 3.2.1, we conduct a comparative analysis against ATOM (Alim Ul Gias, Giuliano Casale, and Woodside, 2019), a state-of-the-art research autoscaler, in a *vertical scaling* context. This experiment validates $\mu$Opt's computational efficiency and its ability to improve application performance.

Second, in Section 3.2.2, we evaluate $\mu$Opt's practical applicability in a *horizontal scaling* context on a production-grade GKE cluster. This evaluation is extensive: it begins with a validation of the fluid LQN model's accuracy across a diverse testbed of randomly generated applications, and then proceeds to a direct benchmark of $\mu$Opt against the industry-standard Kubernetes HPA.

The whole experimental infrastructure and the associated resources are available at (Incerto, Pizziol, and Tribastone, 2025b).

### 3.2.1 Comparison with ATOM

We start our experimental campaign by benchmarking $\mu$Opt against ATOM. We selected this state-of-the-art research autoscaler as a baseline because, like our framework, it leverages LQNs for decision-making, and has been favorably compared with other methods, making it the most direct competitor for validating our model's efficiency.

#### Experiment Setup

For our experimentation, we selected Acmeair (Tollefson and Spyker, 2021), a widely recognized microservices benchmark (Aderaldo et al., 2017) that simulates an airline web application, allowing users to search for, book, and cancel flights while earning reward points. Acmeair has been widely used in prior publications to study the performance-related

properties of these systems (Blinowski, Ojdowska, and Przybyłek, 2022; Inagaki et al., 2022).

Acmeair provides nine endpoints to its users: *Auth*, *ValidateId*, *ViewProfile*, *UpdateProfile*, *QueryFlights*, *BookFlights*, *UpdateMiles*, *GetRewardMiles*, and *CancelBooking*. Since we aim to validate $\mu$Opt's autoscaling capabilities, we deployed Acmeair with the finest granularity by assigning each endpoint to a dedicated microservice, allowing us to scale them independently when required.

We forked a version of Acmeair based on Java Springboot (McClure, 2023) recently used for detecting layered bottlenecks (Inagaki et al., 2022). In this setting, each microservice maps to a Java web server assigned to a specific Linux Cgroup (Rosen, 2013).

Building on this setup, the executor, shown in Figure 8, manages resource allocation through vertical scaling. This approach is the most natural for evaluating $\mu$Opt, as its core optimization computes the ideal processor multiplicities, which directly translate to CPU shares for each microservice. Specifically, the executor implements these changes by adjusting the corresponding `period` and `quota` directives within the Cgroup system. These directives, commonly employed in modern container orchestrators and Linux-based systems (Rosen, 2013), control resource allocation. For instance, if $s_{ms}$ represents the number of cores allocated to microservice $ms$, the executor sets `quota` = `period` * $s_{ms}$, granting microservice $ms$ up to `quota` microseconds of CPU time within each `period`. In our experiments, we calibrated the workload to align with the machine's capabilities, ensuring that a single machine could handle the aggregate CPU allocation across all microservices.

The service mesh was achieved through *HAProxy* (*HaProxy, The Reliable, High Performance TCP/HTTP Load Balancer* 2023), responsible for directing traffic to the appropriate microservice. We selected HAProxy due to its widespread use, high performance, and availability (Kaushal and Bala, 2011). We also utilized HAProxy rules to allocate endpoints to different microservices, enabling modifications to the deployment of Acmeair without code changes. Each microservice was wrapped with a dedicated HTTP proxy to intercept requests and measure response times

and throughput non-intrusively. A centralized *MongoDB* database handled data storage, chosen over a distributed option as it did not impact system performance and because it was already included in Acmeair's base version.

We deployed the experimental infrastructure on two Ubuntu machines connected to a low-latency network. One machine (with 96 Intel Xeon cores and ∼60GB RAM) hosted the microservices and the configuration executor; the other (with 16 Intel Xeon cores and ∼20GB RAM) executed the controller, the workload generator, and the databases. This configuration was specifically designed to provide a highly controlled environment, minimizing confounding factors such as stochastic network variability and resource fragmentation to ensure a precise evaluation of the optimization logic.

The $\mu$Opt's optimization problem, implemented in *Julia*[1] is updated and solved once per second. For the optimization problem described in Section 3.1, we set the weight parameter $\psi = 0.5$ in the objective function (3.1). This value was chosen to establish a balanced trade-off, giving equal importance to maximizing application throughput and minimizing the cost of allocated CPU cores. The executor block, implemented in Python, computes the appropriate actuation in terms of period and quota values (Gao et al., 2020).

**LQN Model Derivation and Validation**

Figure 9 shows the LQN model of Acmeair, derived using the approach described in Section 3.1. Since every microservice in our deployment has a single endpoint, each task in the corresponding LQN has one entry. Communication between microservices is modeled by synchronous calls between LQN entries. The task *Client* models the workload behavior, exercising all Acmeair's endpoints in sequence.

To validate the LQN model, we numerically integrated the corresponding ODEs using Matlab's stiff differential equations solver (MathWorks, 2023) and compared the results with the measured throughput and re-

[1] `https://julialang.org`

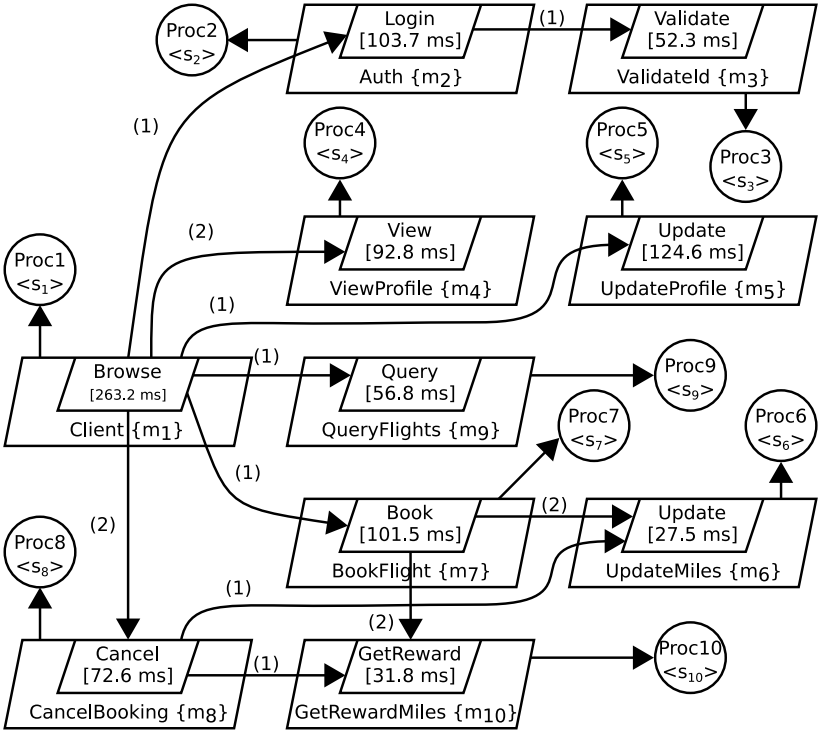



**Figure 9:** The LQN model of Acmeair.

sponse times of a running Acmeair instance under increasing workloads (from 2 to 180 users) and with a different CPU allocation from that used for LQN calibration. Figure 10 presents the error statistics. The model demonstrates high accuracy, with response time errors consistently below 8% (and often under 2%) and throughput errors under 0.5%.

### Evaluation

**Workload Traces.** For our evaluation, we used three workloads, each lasting 2000 s, consisting of two synthetic traces and one real-world workload trace. For the synthetic traces, we used a sinusoidal load with a

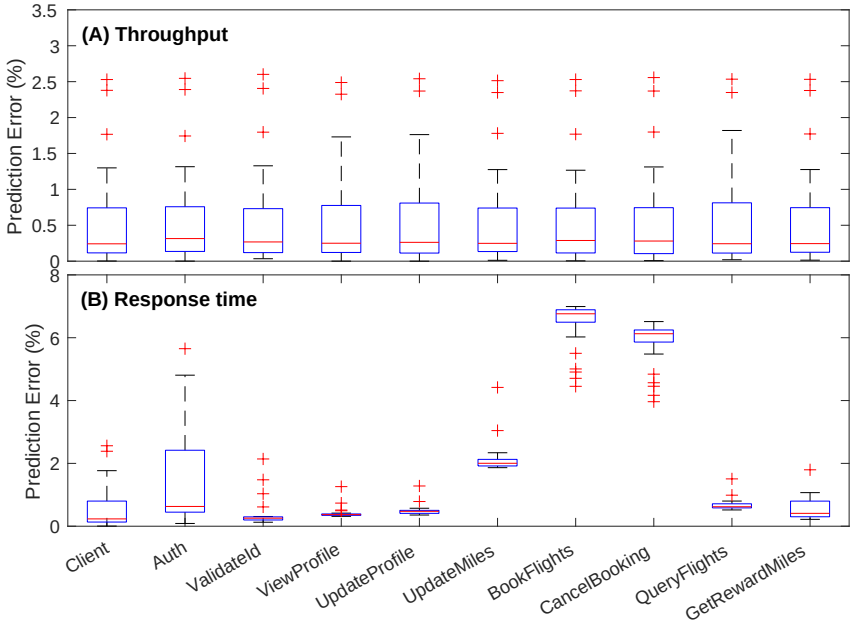



**Figure 10:** The statistics of the percentage error between the measured and the predicted steady-state throughput (A) and response times (B) for each microservice of Figure 9.

period of 200 s (Sin200 Load) and a random step function with high variability, changing every ∼5 s (Step Load)—both commonly used for evaluating autoscaling solutions (Luciano Baresi, Guinea, et al., 2016; Luciano Baresi and Quattrocchi, 2020). The real-world trace was based on Twitter's traffic (Archive Team, 2021) (Twitter Load). To simulate users' behavior, we used *Locust*[2], a widely used open-source load testing tool. We tuned the workloads to keep the number of users between 10 and 80 to account for the computational limits of our infrastructure.

**ATOM Setup.** We adapted ATOM's code (*ATOM: Autoscaler for Microservices* 2023) to work with Acmeair. Since ATOM uses a stochastic method, we ran each autoscaling scenario 30 times to obtain statistically significant

[2] `https://locust.io`

results. The solution found by nonlinear optimization solvers can be sensitive to the initial starting point. To ensure an unbiased comparison between $\mu$Opt and ATOM, we initialized both autoscalers with the same CPU allocation at the beginning of each experimental run. This practice guarantees that any performance difference is attributable to the core logic of the autoscalers rather than their initial state.

**Metrics.** As a performance metric we consider the throughput, measured as the number of completions per unit of time of the user's logic as described by the *Browse* entry. We also track the overall resource allocation, which is measured as the sum of the CPU cores assigned to all microservices. To quantify the difference between $\mu$Opt and ATOM, we considered the following relative metrics:

$$\Delta\mathcal{T} = \frac{\mathcal{T}_\mu - \mathcal{T}_A}{\mathcal{T}_A} \times 100, \quad \Delta S = \frac{S_\mu - S_A}{S_A} \times 100 \tag{3.10}$$

where $\mathcal{T}_\mu$ and $\mathcal{T}_A$ are the average measured throughputs, and $\mathcal{S}_\mu$ and $\mathcal{S}_A$ are the cumulative allocated CPU cores when $\mu$Opt and ATOM were employed, respectively.

**Discussion.** Figure 11 shows the results of the evaluation. For each scenario, we report the workload shape, the cumulative allocated cores, and the throughputs over time. In all the plots except for the workload shape, we report the average values (solid lines) and the corresponding 95% confidence interval (shaded area) computed over the 30 independent runs. These plots confirm that the configurations computed by $\mu$Opt are deterministic across all the runs, whereas ATOM exhibits some variance. However, the number of performed runs was sufficient to support this evaluation statistically, as the confidence intervals' width is narrow (i.e., $\sim$10% of the average value). Finally, Table 3 presents the percentage differences, $\Delta\mathcal{T}$ and $\Delta S$, as defined in Equations (3.10), for each workload, along with the corresponding 95% confidence intervals.

The results presented in this section demonstrate that in the context of vertical scaling, $\mu$Opt is highly effective at achieving its performance

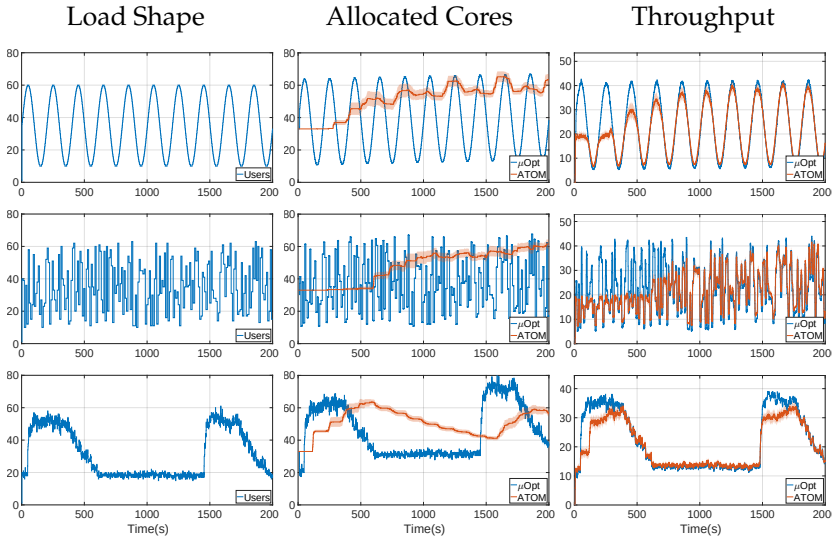



**Figure 11:** Comparison of $\mu$Opt and ATOM applied to Acmeair under three different workloads. Each row displays the workload shape in terms of concurrent users (left), the total allocated cores by each autoscaler (middle), and the measured throughput (right). In all plots except the workload shape, we show the average values (solid lines) and the 95% confidence interval (shaded area), based on 30 independent runs.

targets. Across all workload scenarios, its resource allocation strategy dynamically mirrors the workload shape, allowing it to achieve consistently higher application throughput—by $\sim$8–9%—using significantly fewer resources—by $\sim$12–25%—compared to ATOM. ATOM's slower, genetic-algorithm-based approach leads to prolonged periods of under- (e.g., bottom row of Figure 11 in the first $\sim$400 s) and over-provisioning (e.g., bottom row of Figure 11 in the range $\sim$400–1400 s).

Our findings also highlight $\mu$Opt's minimal computational overhead. Indeed, Figure 12 shows that $\mu$Opt computes each new configuration in a fraction of a second, orders of magnitude faster than ATOM. Specifically, in our experiments $\mu$Opt required negligible CPU time (a fraction of a second on average) and memory usage ($\sim$200MB) thus enabling the fine-grained adjustments that prevent resource waste.

**Table 3:** The percentage difference between $\mu$Opt and ATOM, $\Delta\mathcal{T}$ and $\Delta S$ as described by Equation (3.10), for each workload of Figure 11 with the relative 95% confidence intervals.

| **Workload** | $\Delta\mathcal{T}$ | $\Delta S$ |
|---|---|---|
| Sin200 Load | $+9.34\% \pm 4.16\%$ | $-25.21\% \pm 5.90\%$ |
| Step Load | $+8.35\% \pm 2.80\%$ | $-20.83\% \pm 3.38\%$ |
| Twitter Load | $+7.65\% \pm 1.63\%$ | $-12.38\% \pm 3.54\%$ |

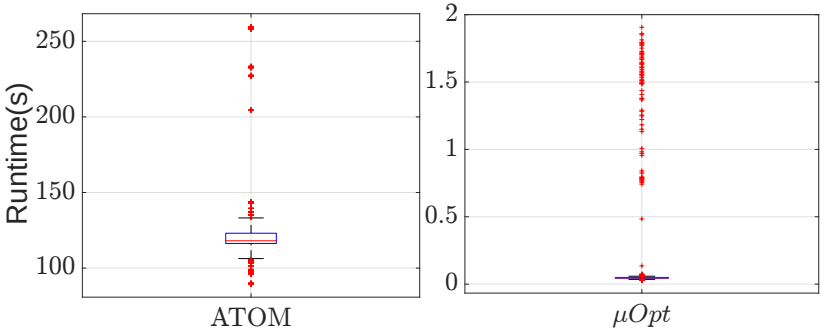



**Figure 12:** Distributions of the runtimes of ATOM and $\mu$Opt for solving each optimization problem during experiments of Figure 11.

While this analysis confirms $\mu$Opt's fundamental effectiveness and efficiency against a research baseline in a vertical scaling scenario, its practical applicability in a modern, horizontal-scaling context remains to be demonstrated.

### 3.2.2 Comparison with HPA

This section provides the comprehensive evaluation of $\mu$Opt in a horizontal scaling context on GKE. To this end, we first present a validation of our fluid LQN model's accuracy and solution time on a testbed of ten randomly generated applications. With the model's credibility established in this context, we then benchmark the horizontal autoscaling capability of $\mu$Opt against the industrial standard, HPA (*Horizontal Pod Autoscaler* 2024), on the Acmeair application, providing a strong assessment of its

**Table 4:** Statistics of the randomly generated systems used in the model validation campaign on GKE. The table reports the mean (± standard deviation) and the maximum number of microservices, call types (synchronous or asynchronous), and probabilistic choices.

| Statistic | Mean (± std. dev.) | Max |
|---|---|---|
| Microservices | $4.50 \pm 2.06$ | 9 |
| Synch Calls | $2.80 \pm 2.27$ | 7 |
| Async Calls | $2.70 \pm 2.15$ | 8 |
| Prob Choices | $1.60 \pm 1.43$ | 5 |

effectiveness and efficiency in an industrial setting.

**Overview of HPA.** In Kubernetes, the most commonly used solution for automatically adjusting the replica count is HPA. It makes scaling decisions by continuously monitoring the CPU usage of pods and comparing it to a user defined target. HPA uses a proportional scaling strategy to increase or decrease the number of replicas of a given microservice $ms$, calculated using the following formula:

$$\mathcal{S}'_{ms} = \left\lceil \mathcal{S}_{ms} \times \frac{\mathcal{U}_{ms}}{\mathcal{U}_{tgt}} \right\rceil \tag{3.11}$$

where $\mathcal{S}'_{ms}$ is the new replica count, $\mathcal{S}_{ms}$ is the current count, $\mathcal{U}_{ms}$ is the measured average CPU usage, and $\mathcal{U}_{tgt}$ is the target utilization.

### Experiment Setup

This section details the setup for two experiments on Kubernetes: a model validation campaign and a benchmark of $\mu$Opt against HPA.

For the model validation, we considered a testbed of ten randomly generated microservice applications using a tool similar to Hydragen (Sedghpour et al., 2023). These applications feature diverse topologies and communication patterns, whose keys characteristics are summarized in Table 4. Each application's microservices is implemented in Python and

uses the Flask library to expose its HTTP endpoint. To conduct the validation under realistic horizontal autoscaling settings, we containerized each microservice using Docker, and deployed as Kubernetes pods with 1.5GB of RAM and 3 CPUs on a single-node cluster (60 vCPUs, ∼240GB RAM ).

For the benchmark against HPA, we used a containerized version of the Acmeair application from Section 3.2.1. Each microservice was deployed as a Kubernetes pod with 1GB of memory and 1 CPU on a multi-node cluster of three nodes (each with 30 vCPUs, ∼120GB RAM). For this benchmark, we adapted $\mu$Opt to target a specific CPU usage level, setting its weight parameter to $\psi = 0.5$ and revising its executor to support horizontal scaling by adjusting the replica count $r_{ms}$ of a microservice using the formula:

$$r_{ms} = \left\lceil \frac{s_{ms}}{s_{pod}} \right\rceil \tag{3.12}$$

where $s_{ms}$ is the number of cores computed by the optimization and $s_{pod}$ is the pod's base CPU configuration (1 CPU in our case). The Acmeair LQN model was recalibrated with service times measured on this platform. For the HPA baseline, we used version 1.24 with default parameters. We avoided manual tuning to ensure the evaluation focused on the intrinsic capabilities of the HPA algorithm rather than a specific, human-optimized configuration. A Debian-based auxiliary machine (2 vCPUs, ∼4GB RAM) hosted the workload generator and monitoring components for both experiments, and additionally hosted the $\mu$Opt controller for the benchmark.

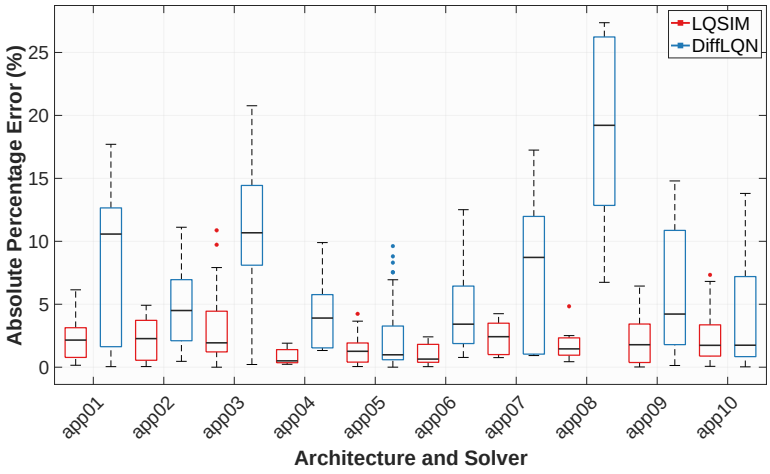
#### Model Validation in a Horizontal Scaling Context

This subsection presents the validation of our fluid LQN model. We test the predictive power of our model on the ten randomly generated applications detailed in the previous subsection. To do so, we evaluate the model's accuracy by comparing real-world measurements against the predictions of our fluid approximation (DiffLQN) and the state-of-the-art discrete-event LQN simulator, LQSIM (Woodside and Franks, 2002).

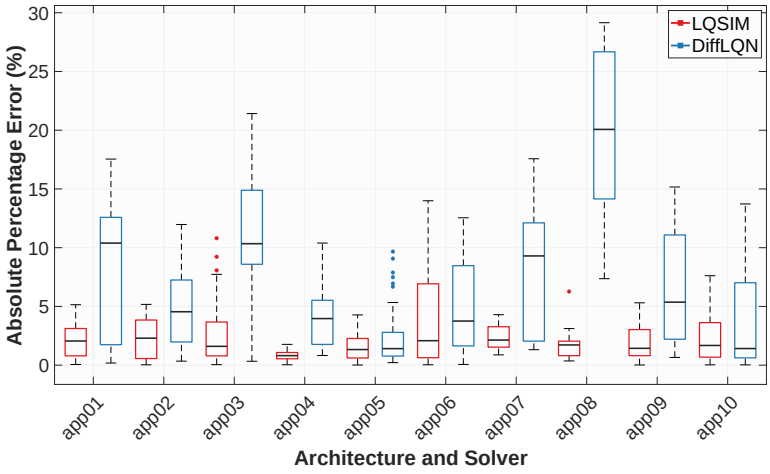
Our validation process consisted of two steps. First, we calibrated each of the ten fluid LQN models by using the Utilization Law and collecting performance metrics (i.e., utilization and throughput) from the corresponding real microservice application. In order to do so we deployed each microservice with a single replica and stressed the application under a single-user workload. We then validated each model by running 10 new experiments with randomly selected numbers of concurrent users (from 1 to 12) and replicas per microservice (from 1 to 8). By doing so we were able to stress the system under various CPU utilization levels not seen during calibration. Then, the measured throughput and CPU utilization were compared against the predicted ones computed with both LQSIM and our fluid approximation.

The results reported in Figure 13 denote a good level of accuracy across the ten diverse architectures, with the median prediction error for throughput and CPU utilization generally below 15%. As expected, LQSIM is more accurate but this precision comes at a high computational cost. Our fluid model, in contrast, reports a slightly higher prediction error in exchange for a dramatic speedup in solution time, as shown in Figure 14. This computational efficiency is precisely what enables real-time control, and the results of the autoscaling experiments conducted in this section prove that the model's accuracy allow $\mu$Opt to undertake effective autoscaling actions.

Finally, in Figure 15, we report the MAE of the model's predictions for both throughput and CPU utilization against the maximum CPU utilization across all microservice for a specific application. The scatter plots reveal no strong correlation between prediction error and the CPU utilization level, even in extreme conditions (i.e., $100\%$) for either metric. This is formally confirmed by low Pearson correlation coefficients ($r = 0.068$ for throughput and $r = 0.048$ for CPU). This consistency under a wide spectrum of CPU utilization level, combined with the overall prediction accuracy, demonstrates that our fluid LQN model is accurate enough for practical use across diverse system architectures and CPU utilization level. This confirms that the LQN fluid model is a reliable foundation for the $\mu$Opt autoscaler.




**(a)**




**(b)**

**Figure 13:** Validation of LQN model prediction accuracy against real-world measurements. The plots compare the distribution of absolute percentage errors for the fluid model and LQSIM. This comparison is performed across ten different LQN architectures for two key metrics: throughput (a) and CPU utilization (b).

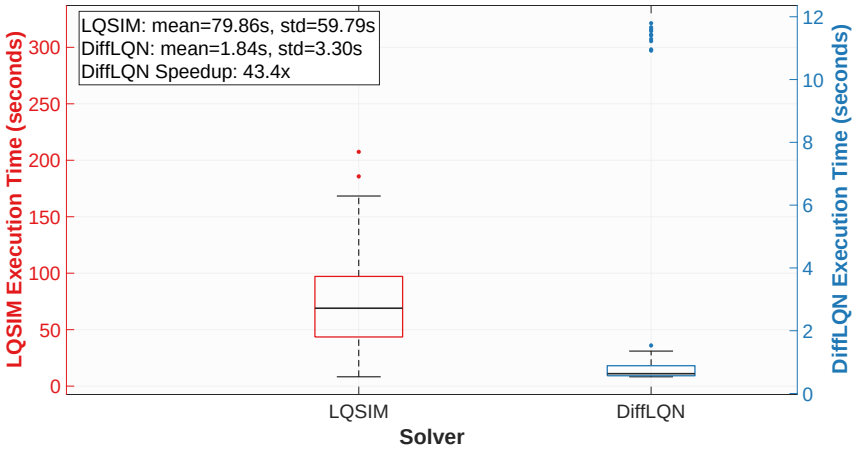
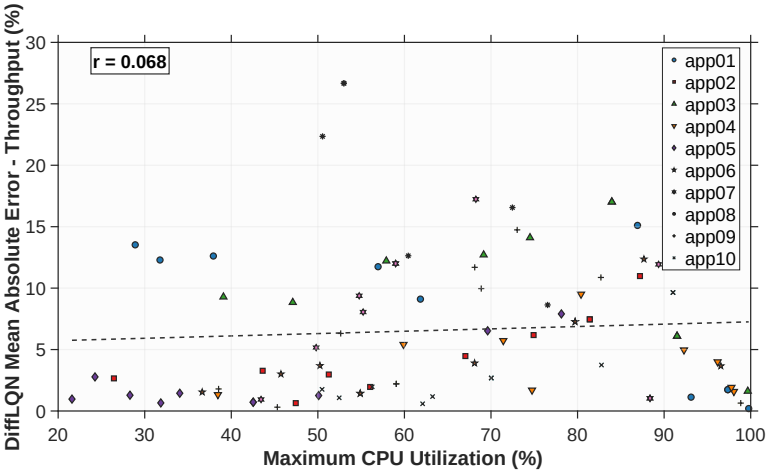


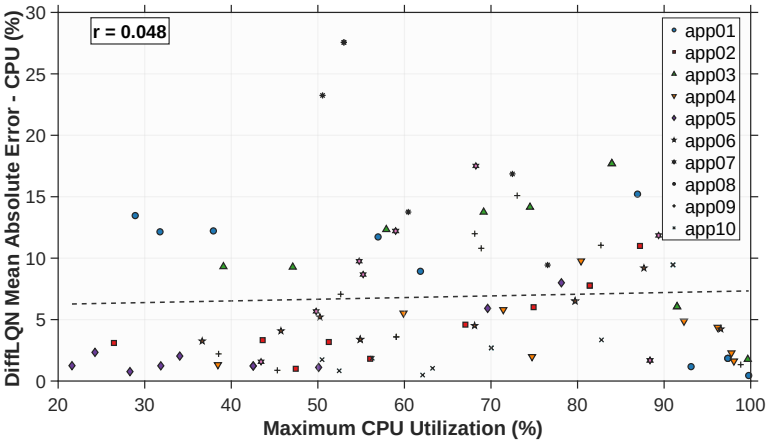

**Figure 14:** Distribution of solution times for the LQN models using LQSIM and the fluid approximation (DiffLQN). For the LQSIM runs, the simulation timeout was set to 250 s, with a precision target of 5% (i.e., the maximum accepted error for results to be considered stable).

## Evaluation

**Workload Traces.** To evaluate $\mu$Opt under realistic conditions, we conducted 60-minute experiments using workload shapes similar to those in Section 3.2.1. Specifically, we slowed down the original Sin200 Load to a period of 400 s (Sin400 Load) and replaced the Step Load with another sinusoidal pattern with a period of 800 s (Sin800 Load). These adjustments were necessary because Google Cloud Platform (GCP) could not keep pace with rapidly fluctuating workloads; otherwise, the frequent changes would result in excessive pod creations and terminations, leading to application instability. Lastly, we used Twitter Load again as a real-world workload scenario. We used Locust to simulate user behavior, adjusting the load to vary between 10 and 55 concurrent users. As both $\mu$Opt and HPA are deterministic, single experimental runs were sufficient for the comparison.




(a)




(b)

**Figure 15:** Generalization power analysis of the DiffLQN model against increasing CPU utilization. The plots show the Mean Absolute Error (MAE) of the model's predictions for throughput (a) and CPU utilization (b) versus maximum CPU utilization. The Pearson correlation coefficient ($r$) is displayed in the top-left corner of each plot.

**Metrics.** To ensure a fair comparison with HPA, we used CPU usage as $\mu$Opt's KPI, setting the target at 0.20. This value was chosen to allow for a larger number of replicas, enabling us to stress the autoscaling behavior. Besides CPU usage, we tracked resource allocation, which was measured as the total number of replicas assigned to all microservices. To quantify the differences between $\mu$Opt and HPA, we defined the following relative metrics:

$$\Delta\mathcal{U} = \frac{\mathcal{U}_{as} - \mathcal{U}_{tgt}}{\mathcal{U}_{tgt}} \times 100, \quad \Delta R = \frac{R_{\mu} - R_H}{R_H} \times 100 \tag{3.13}$$

where $\mathcal{U}_{as}$ represents the average CPU usage obtained by the selected autoscaler (either $\mu$Opt or HPA), and $\mathcal{U}_{tgt}$ is the CPU target. $\mathcal{R}_{\mu}$ and $\mathcal{R}_H$ denote the total number of replicas allocated by $\mu$Opt and HPA, respectively.

**Table 5:** Comparative analysis of HPA and $\mu$Opt on the Acmeair benchmark. The table is organized by **Workload** type (Sin400, Sin800, Twitter). For each autoscaler, we report: **Replicas**, the average number of allocated instances (lower implies higher efficiency); **Avg. Usage**, the mean CPU utilization achieved; and **Tracking Error**, quantifying control precision as the absolute deviation from the $0.20$ target ($|u_{measured} - 0.20|$). Values in parentheses indicate the percentage reduction (improvement) of $\mu$Opt compared to the HPA baseline.

| Workload | Autoscaler | Replicas | Target Tracking Performance | |
|---|---|---|---|---|
| | | | Avg. Usage | Tracking Error |
| Sin400 Load | HPA | 50.43 | 0.12 (-40%) | 0.08 |
| | $\mu$Opt | 35.20 (**-30.20%**) | 0.15 (-25%) | 0.05 (**-37.50%**) |
| Sin800 Load | HPA | 46.30 | 0.13 (-35%) | 0.07 |
| | $\mu$Opt | 33.93 (**-26.72%**) | 0.15 (-25%) | 0.05 (**-28.57%**) |
| Twitter Load | HPA | 34.17 | 0.13 (-35%) | 0.07 |
| | $\mu$Opt | 26.83 (**-21.48%**) | 0.15 (-25%) | 0.05 (**-28.57%**) |

**Discussion.** The results presented in Figure 16 and Table 5, provide a significant validation of $\mu$Opt's capabilities in realistic cloud-native

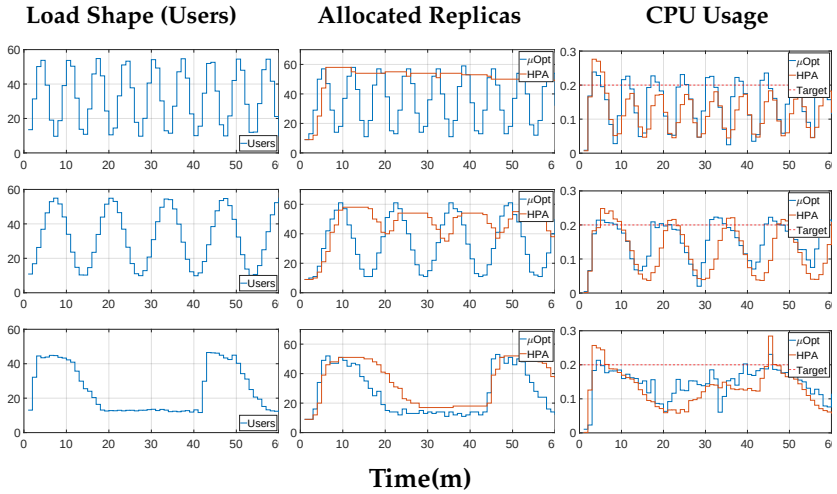



**Figure 16:** Comparison of $\mu$Opt and HPA applied to Acmeair. Each row displays a different workload (from top to bottom: Sin400 Load, Sin800 Load, and Twitter Load). The columns show the workload shape in terms of concurrent users (left), the total allocated cores by each autoscaler (middle), and the resulting average CPU usage against the target value (right).

context.

HPA struggles to adapt to rapid workload changes, especially during downscaling, often taking decisions with a noticeable delay. This is particularly evident in the Sin400 Load scenario (Figure 16, top row), where HPA does not follow the shape of the workload closely, leading to consistent over-provisioning and a CPU usage significantly lower than the target. In contrast, $\mu$Opt's proactive strategy allows it to track the workload far more effectively across all three scenarios. As a result, $\mu$Opt maintains CPU usage much closer to the target of $0.20$, reducing the tracking error by 29–38% across the tested workloads. Crucially, it achieves this superior control precision while simultaneously using $\sim$21–30% fewer replicas compared to HPA.

From a computational perspective, things are different compared to the evaluation in Section 3.2.1. $\mu$Opt's computation time and memory requirements are slightly higher ($\sim 10^1$ s and $\sim 10^2$ MB, respectively) com-

pared to those of HPA ($\sim 10^{-1}$ s and $\sim 10^{1}$ MB, respectively). This increment is attributable to the need for $\mu$Opt to solve a complex optimization problem, which HPA does not require and therefore exhibits maximum efficiency. However, although $\mu$Opt's times are longer, they are compatible with those required by GKE to deploy new replicas (*Horizontal Pod Autoscaler* 2024). Therefore, they do not present a limitation in terms of adaptation speed, even in an industrial context like GKE.

## 3.3 Chapter Summary

This chapter presented $\mu$Opt, a novel, model-based autoscaler designed to address the critical challenge of resource management in microservice architectures. We began by detailing the architecture of the $\mu$Opt framework, which uniquely combines the expressive power of LQNs with the efficiency of non-linear optimization. We explained how the use of a fluid approximation transforms the complex LQN model into a system of ordinary differential equations, making the optimization problem computationally tractable for real-time control. This allows $\mu$Opt to proactively calculate optimal resource configurations, adapting to workload changes in near real-time.

A comprehensive, two-part evaluation demonstrated the practical effectiveness of our approach in both vertical and horizontal scaling contexts:

1. **Comparison with ATOM (Vertical Scaling)**: When benchmarked against ATOM, a state-of-the-art research autoscaler, $\mu$Opt proved to be three orders of magnitude faster in computing scaling decisions. This superior speed enabled it to accurately track highly variable workloads where ATOM could not, resulting in both higher application throughput (up to 9%) and significant resource savings (up to 25%).

2. **Comparison with Kubernetes HPA (Horizontal Scaling)**: This evaluation was conducted on a production-grade GKE cluster. We first validated the fluid LQN model on a diverse testbed of ten

randomly generated applications, confirming its accuracy (median error $< 15\%$) and efficiency (a 43.4x speedup over simulation). Having established the model's robustness, the subsequent benchmark against the HPA demonstrated that $\mu$Opt's proactive nature is superior, reducing allocated replicas by up to 30% while more accurately adhering to the target CPU utilization.

In summary, this chapter successfully demonstrated that the $\mu$Opt framework provides a robust and efficient solution to the microservice autoscaling challenge. By outperforming both a state-of-the-art research tool and a widely adopted industrial solution, we have validated our model-based, predictive methodology as a powerful approach for achieving a superior balance between performance and cost in dynamic cloud-native environments.

# Chapter 4

# WasteLess

This chapter introduces WasteLess, the second framework developed in this thesis, designed to address the resource management challenges of second-generation FaaS platforms. This chapter details the design, architecture, and evaluation of WasteLess, a holistic, offline provisioning framework that leverages the same model-based optimization methodology as $\mu$Opt. We will demonstrate how this approach finds the optimal balance between cost and performance in FaaS applications, ultimately reducing operational costs by an average of 38% compared to baseline deployments and lowering latency by up to 72% against default platform schedulers.

The chapter begins in Section 4.1 by presenting the architecture of the WasteLess framework, detailing its offline optimization process that jointly configures CPU, memory, and concurrency. Section 4.2 provides a comprehensive evaluation of WasteLess, benchmarking it against both the commercial-grade GCR instance scheduler and a state-of-the-art research baseline. Finally, Section 4.3 concludes the chapter with a summary of the key findings related to the WasteLess framework.

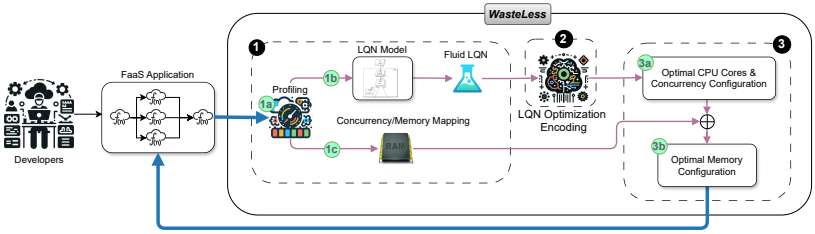



**Figure 17:** Overview of WasteLess.

## 4.1 The WasteLess Framework Architecture

This section provides a comprehensive discussion of WasteLess. Here we present it in the context of a generic abstract FaaS provider that has the following features:

1. a metrics monitor for CPU utilization, memory consumption, concurrency limit and actual concurrency level, and response time, for each function;

2. an API that can configure CPU core count, memory allocation, and concurrency limit per function.

An overview of WasteLess is illustrated in Figure 17. As input, WasteLess takes a FaaS application comprising a set of functions that communicate via HTTP messages.

WasteLess operates in three distinct phases, each depicted by a dashed rounded box in Figure 17. The initial phase serves two primary purposes, both originating from the same profiling campaign: (i) constructing a well-calibrated LQN model of the FaaS application, enabling performance prediction across diverse configurations; (ii) establishing a mapping between the concurrency limit and memory footprint of each function.

In the second phase, utilizing the fluid representation of the LQN (see Section 1.4), a differentiable nonlinear optimization problem is formulated. This problem incorporates the fluid LQN's ODEs as constraints, with decision variables representing the LQN state and the processor

multiplicities of each function. Solving this problem yields the ratio between the optimal concurrency limit and the number of cores, which is preserved regardless of the workload intensity.

In the final phase, WasteLess translates this ratio into concrete configurations for adjusting the deployment of each function within the FaaS system. This phase consists of two steps: firstly, the optimization result directly determines the core count and concurrency limit assigned to each function. Secondly, the optimal number of active users is utilized to compute the memory required by each function to support the concurrent configuration. The subsequent sections elaborate on these phases in detail.

❶ *LQN Model Creation & Concurrency/Memory Mapping*

This phase aims to discover the LQN model of the input FaaS application and establish a link between function concurrency limit and memory usage. Both require initial profiling to gather information. Details on LQN model creation, concurrency/memory mapping, and profiling are reported below.

1a *Profiling*

Here, we gather essential information to construct an LQN model of the FaaS application and a model mapping the concurrency limit of each function to its memory consumption. To achieve this, we assume the ability to exercise the application under study by generating a swarm of users submitting requests. We utilize standard profiling and load testing tools for this purpose. It is important to note that the profiling protocol varies depending on the intended purpose of the collected data, i.e., LQN model creation or mapping the concurrency limit over memory. We will elaborate on these variations as we introduce the procedures for model creation.

1b *LQN Model Creation*

The LQN model is constructed in two distinct phases. Initially, the model's topology is derived by analyzing workflow artifacts, as detailed in (R. Wang, Giuliano Casale, and Filieri, 2022). These artifacts outline

the application's composition in terms of serverless functions and their interconnections. A workflow specification serves as a blueprint, dictating how different functions interact to achieve a desired outcome. Workflows are typically represented as DAGs, where nodes symbolize serverless functions and edges represent dependencies among them, often manifested as HTTP requests. Following the approach in (R. Wang, Giuliano Casale, and Filieri, 2022), the type of connection (synchronous or asynchronous) and the presence of branching behavior can be extracted from these artifacts. This DAG representation readily translates into an LQN structure. However, at this stage, the LQN parameters, i.e., the processing rates ($e_i$) and routing probabilities ($p_i$), remain undetermined.

Parameter calibration is performed via a systematic profiling procedure. First, the application is executed under a controlled single-user load to measure the CPU time of each function. This metric was chosen because it directly maps to the service time in the LQN model, simplifying data collection and reducing profiling duration. Routing probabilities are then derived by annotating the control flow of each function to track the number of executions of different paths, similarly to (Garbi, Incerto, and Tribastone, 2023). Probabilities are then estimated as the ratio of executions of a given path to their total number of executions.

Since at this stage the optimal FaaS configurations is not known, each function is configured to utilize 1 core and a concurrency limit of 1, allowing the serverless platform to automatically scale the number of function instances. Memory configuration primarily requires ensuring that total consumption remains within an acceptable range, such as 80% of total memory, to prevent disruptive events like out-of-memory exceptions. Memory settings can be adjusted through trial and error while monitoring for such exceptions. To accurately capture the system's inherent stochastic behavior, the profiling session continues until a predetermined level of statistical convergence is achieved for the measured quantities. The batch means algorithm (Fishman and Yarberry, 1997) is employed, with a stopping condition where the 95% confidence interval width of the mean CPU time falls below 1% of its mean value. Finally, following a procedure similar to that in Section 1.4, the set of ODEs describing the

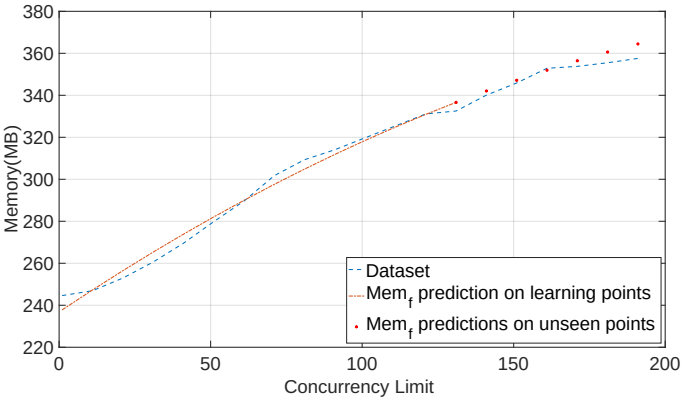



**Figure 18:** Example of WasteLess Concurrency/Memory Mapping. The x-axis denotes the concurrency limit of a function while the y-axis is the required memory.

LQN can be derived and used to define the structure of the WasteLess optimization problem. To simplify this task, we developed a tool included in our replication package (Incerto, Pizziol, Russo Russo, et al., 2025b).

1c *Concurrency/Memory Mapping*

The objective here is to establish the analytic relation $Mem_f(u) : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, which links the concurrent users $u$ on each function to its memory usage. This relationship guides the memory sizing for each function based on the actual concurrency level determined by WasteLess. This step is crucial as WasteLess may optimize concurrency limit, potentially exceeding the memory allocated in a non-concurrent deployment, causing performance issues. To derive $Mem_f$, we conduct a profiling process where the application is run under increasing loads, monitoring memory usage. Each function is configured with a single instance and high concurrency limit to induce memory variation. Collected data is then fitted into a polynomial function correlating the concurrency limit to memory usage for each function.

As a simple example, Figure 18 shows the evaluation of a possible instance of $Mem_f$ synthesized to predict the memory usage of a generic

CPU-bound serverless function. As a model we used a second-degree polynomial trained on a subset of the measured dataset consisting of 14 (concurrency limit, memory) pairs out of 20 available. We performed the fitting using the *polyfit* function natively available in MATLAB. Figure 18 shows three curves: all the measured (concurrency limit, memory) pairs (dashed line), the memory usage predictions computed on the points used for training (dotted line), and the memory usage predictions calculated on concurrency limit not used for training. The results confirm the method's effectiveness.

❷ *WasteLess Optimization Problem*

The goal of this section is to introduce the optimization problem employed by WasteLess to determine optimal configurations for each function. The structure of this optimization problem is outlined below.

$$\min_{c} \quad \psi\mathcal{R}(c,x,\theta) + (1-\psi)\sum_{p\in Proc} c_p \tag{4.1}$$

$$\text{s.t.} \quad \dot{x} = 0 \tag{4.2}$$

$$\sum_{l} x_l = W \tag{4.3}$$

where $Proc$ is the sets of all processors bound to the tasks (i.e., functions) of the LQN and $c = (c_p)_{p\in Proc}$ is the vector of processors' multiplicities; $x$ is the vector of queue lengths at any location of the LQN model, with the queue length at generic location $l$ being denoted by $x_l$; $\theta$ is the vector of LQN parameters, i.e., the service times and branch probabilities.

The objective function (4.1) expresses the goal of the optimization as a combination of two conflicting sub-objectives: the minimization of the average response time $\mathcal{R}(c,x,\theta)$ and the minimization of the allocated resources $\sum_{p\in Proc} c_p$, where $\mathcal{R}$ is derived by applying Little's Law (see Section 1.3.1) to the steady-state solution of the fluid LQN.

Each sub-objective is scaled according to the weight $0\leq\psi\leq1$. Consistent with the design choice established for $\mu$Opt in Section 3.2.1, we set $\psi = 0.5$ to maintain a balanced trade-off between performance and cost.

Equations (4.2)–(4.3) encode the steady-state solution of the fluid LQN model into the optimization problem: Equation (4.2) enforces the steady state of the ODEs, while Equation (4.3) ensures that the total number of users in the model is equal to $W$. These constraints correspond to the equilibrium conditions of the general fluid dynamics formulation described in Section 1.4.

Solving the optimization problem (4.1)–(4.3) yields two key pieces of information: $c^*$, representing the minimum number of cores to assign to each function to minimize its response time, and $\dot{x}^*$, the optimal steady-state solution for all quantities tracked by the fluid LQN. From these, we can obtain $\mathcal{M}^* = (m_i)_{i \in \text{Task}}$ denoting the number of requests executing or waiting for potential nested calls on each Task of the LQN and consequently function of the FaaS system.

It is important to note that it is not possible to directly use $\mathcal{M}^*$ for determining the concurrency limit for each function, as these values depend on the number of users considered while solving the optimization problem and would therefore need to be recalculated each time. To overcome this issue, WasteLess relies on the ratio

$$b_i^* = \frac{m_i^*}{c_i^*} \cdot U^t \quad \forall i \in Task \tag{4.4}$$

which is instead constant regardless of the number of functions' instances and the number of users interacting with the FaaS system. In this ratio, we also introduce the parameter $U^t \in [0, 1]$. It represents the target CPU utilization, a user-defined value that allows developers to control how aggressively resources are used. By setting this value below $1.0$, a certain level of resource headroom can be enforced, improving the system's stability against unexpected workload spikes.

❸ *WasteLess FaaS Configurator*

3a *CPU & Concurrency Configuration*

As a first step, WasteLess calculates the concurrency limit relying on Equation (4.4) that expresses the optimal balance between the number of actual concurrent requests of a function and the corresponding number

of cores. For doing so, it considers $c_i^*$ as the base core allocation of each function and chooses a concurrency limit equal to $\hat{m}_i = \lceil b_i^* c_i^* \rceil$, as the function concurrency limit can only be set to an integer value.

3b *Memory Configuration*

Central in this phase is the function $Mem$ synthesized in point 1c , and the concurrency limit $\hat{m}_i$ just calculated for each function. By leveraging these two quantities, we allocate to each function a memory level equal to $Mem(\hat{m}_i)$ so that it can support the presence of $\hat{m}_i$ concurrent requests without affecting performance.

### 4.1.1 Operational Lifecycle

The operational lifecycle of WasteLess follows the logic established for $\mu$Opt in Section 3.1.2, but is specifically tailored to the dynamics of FaaS applications.

Promptly updating the model is particularly critical in this context, as FaaS platforms delegate infrastructure management entirely to the cloud provider. Consequently, performance characteristics can shift due to internal provider updates even if the user has not significantly altered the application topology or its underlying functioning. Continuous telemetry monitoring is therefore essential to detect when the existing model parameters no longer accurately reflect the current execution environment.

In its current implementation, this maintenance cycle is handled manually, establishing the necessary groundwork for the automated modeling integration and online calibration strategies discussed in Section 5.3.

## 4.2 Evaluation and Results

In this section, we present a comprehensive, two-part evaluation to validate the WasteLess framework on the GCR platform.

First, in Section 4.2.1, we conduct a comprehensive evaluation on a real-world benchmark, Acmeair, comparing WasteLess against the no-concurrency baseline, the commercial GCR instance scheduler, and the state-of-the-art research baseline, ProPack (Roy et al., 2023).

Second, in Section 4.2.2, we further validate the broad applicability and scalability of our approach on a large-scale testbed of 60 randomly generated applications, incorporating complex topologies with asynchronous and parallel communication patterns.

The experimental infrastructure and the associated resources are available at (Incerto, Pizziol, Russo Russo, et al., 2025c).

### 4.2.1 Comprehensive Evaluation on a Real-World Benchmark

This section presents and analyzes the experimental results on a representative benchmark FaaS application, configured with randomly generated service times to evaluate our approach across a variety of conditions.

We leveraged the standard GCR monitoring tools[1] to collect data from the executing system, while we relied on Google Cloud Logging[2] to track the CPU time consumption of individual functions.

#### Experiment Setup

Consistent with the methodology in Chapter 3, we adopt *Acmeair* (D. Tollefson and A. Spyker, 2015) as our reference benchmark. For this study, we ported the original application to a FaaS architecture; this implementation is available in the replication package (Incerto, Pizziol, Russo Russo, et al., 2025c).

The architecture is depicted in Figure 19. Each microservice is implemented as a serverless function in GCR, invoked via HTTP requests. The application is thus composed of nine functions providing the following endpoints to its users: *Auth*, *ValidateId*, *ViewProfile*, *UpdateProfile*, *QueryFlights*, *BookFlights*, *UpdateMiles*, *GetRewardMiles*, and *CancelBooking*.

We replaced the original business logic of each function with a CPU-intensive load, whose resource demand can be easily configured. Thus, we could generate different Acmeair variants, each characterized by different resource demands. We generated 30 variants of Acmeair by uniformly

[1] `https://cloud.google.com/monitoring`
[2] `https://cloud.google.com/logging`

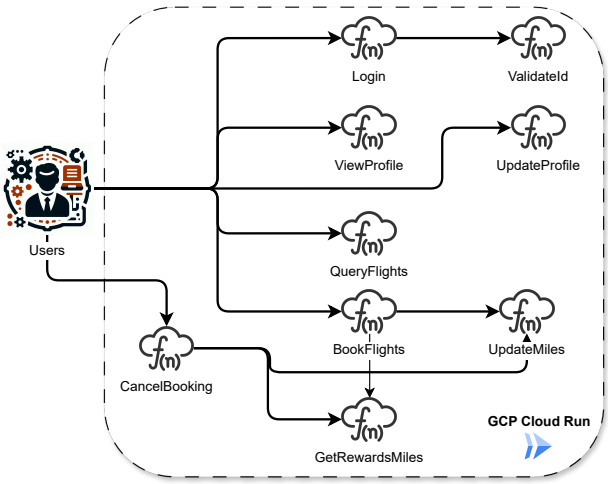



**Figure 19:** Topology of Acmeair's serverless porting.

sampling the CPU demand of each function from the interval $[0.02, 3]$ s. The generation of mixed fast and slow functions served to highlight the unique challenges and potential benefits of concurrent execution.

For each variant, we also generated and calibrated the associated LQN model to be used by WasteLess.

We generated the workload using *Locust*, driven by a real Twitter traffic trace (see, e.g., (Quattrocchi et al., 2024)). Note that this trace differs in duration and intensity from the one used in Chapter 3. This particular trace was selected to thoroughly test the scaling capabilities, as it exhibits both rapid spikes and gradual, slow-varying load changes. The trace was downscaled to a 0–400 req/s range while preserving its original shape. Figure 20 depicts the resulting load profile.

Following the classic implementation of Acmeair (Inagaki et al., 2022), the logic for each user was chosen to solicit all the system functions. We used the Google Cloud command-line interface, namely *gcloud*, to update the number of cores and memory as determined by WasteLess. Furthermore, gcloud is also used at runtime to set the concurrency limit

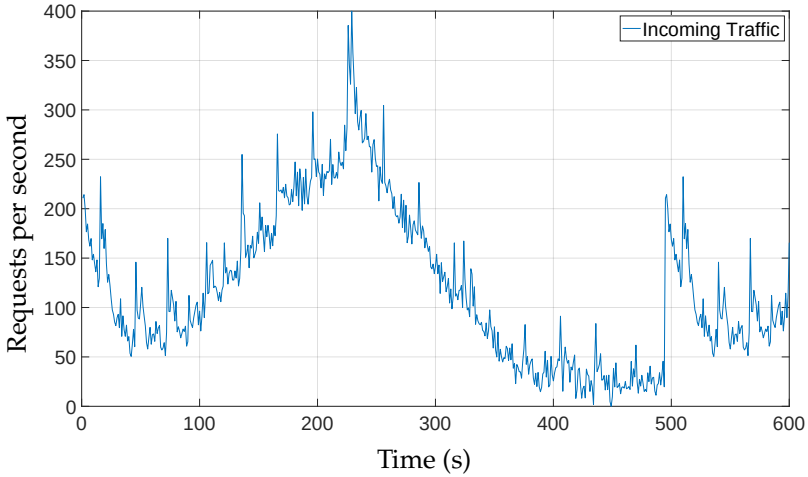



**Figure 20:** Twitter traffic trace used in the experiments; note that this trace differs in duration and intensity from the one used in Chapter 3.

as computed by WasteLess and ProPack when needed.

### LQN Model Derivation

Figure 21 shows the LQN model of Acmeair derived following point 1b in Section 4.1. Each function is assigned to an LQN task with a single entry corresponding to its HTTP endpoint. Function communications are modeled as synchronous calls between LQN entries, and the logic of the task *Client* represents user behavior. Since each Acmeair variant differs in function execution speed, the service rates for each model entry vary accordingly. For brevity, Figure 21 presents the parameterization of a particular instance; detailed information on all variants is available in the replication package (Incerto, Pizziol, Russo Russo, et al., 2025c). To validate each LQN model, we solved the corresponding fluid LQN using MATLAB and compared the results with the average steady-state end-to-end response time and throughput measured in the corresponding Acmeair instance. Specifically, the application was tested under workloads ranging from 2 to 180 users, representing validation conditions

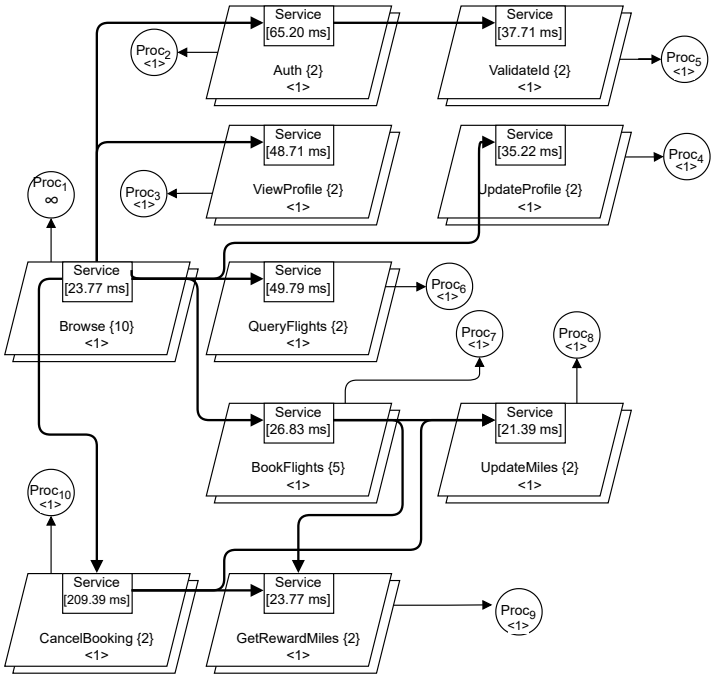



**Figure 21:** LQN model of an analyzed variant of Acmeair.

distinct from the single-user workload used for model calibration (cf. point 1b in Section 4.1). For the LQN reported in Figure 21, model accuracy is high: the error on response time is less than 10% on average, while throughput prediction error is less than 5%. Comparable accuracy scores also apply to all other Acmeair variants.

### Results

We conducted a series of experiments, each involving one of the 30 variants of Acmeair under four different scenarios of concurrency limit configuration: (i) no-concurrency; (ii) GCR instance scheduler with default concurrency limit (i.e., 80); (iii) WasteLess-suggested concurrency limit; (iv) ProPack-suggested concurrency limit.

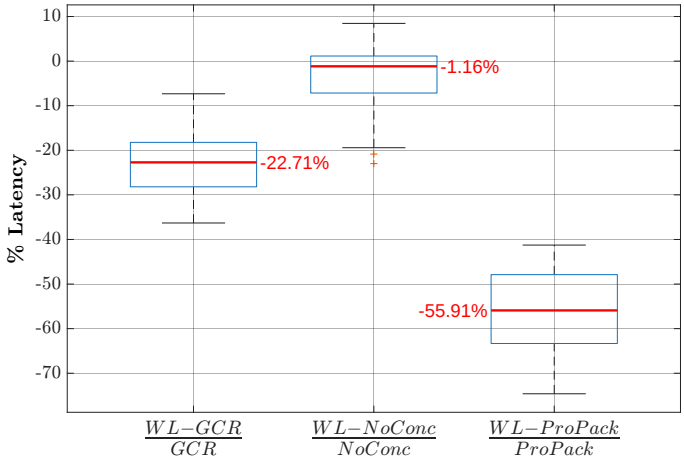



**Figure 22:** Percentage difference in latency between WasteLess (WL) and GCR, no-concurrency (NoConc), and ProPack.

For a fair evaluation, all four scenarios used the same CPU core and memory allocations suggested by WasteLess, since both GCR and ProPack do not offer methods for automatic configurations for these resources.

For the WasteLess-suggested scenario, we set the target utilization parameter $U^t$=1 (see Equation (4.4)). This was chosen to evaluate the core optimization algorithm's performance without user-imposed resource headroom, ensuring a direct and fair comparison against other approaches.

Figure 22 presents three boxplots with the statistics of three crucial metrics:

- **WasteLess vs GCR (left)**: The percentage difference in average end-to-end response time between WasteLess and the GCR instance scheduler with its default configuration.
- **WasteLess vs no-concurrency (middle)**: The percentage difference in average end-to-end response time between WasteLess and no-concurrency.
- **WasteLess vs ProPack (right)**: The percentage difference in average

end-to-end response time between WasteLess and ProPack.

Our analysis shows that WasteLess achieves significantly lower end-to-end response times compared to the GCR instance scheduler and ProPack. Specifically, WasteLess reduces the average response time by 7% to 35% (median 23%) compared to GCR, and by 40% to 74% (median 56%) compared to ProPack. The Kolmogorov-Smirnov test (Berger and Zhou, 2014) confirms the statistical significance of these differences. For GCR configurations versus WasteLess, the null hypothesis of equal response time distributions was rejected with a 95% confidence level and a p-value of 0.0113. Similarly, comparisons between ProPack-suggested configurations and WasteLess yielded a p-value of $5.5870e^{-8}$, demonstrating a high statistical significance.

Comparing the average end-to-end response times of WasteLess with those measured under no-concurrency, we observe percentage differences between approximately -20% and 8%, indicating that no-concurrency is not always the optimal choice performance-wise. The Kolmogorov-Smirnov test does not reject the hypothesis of equality for the average response time distributions obtained with WasteLess and the no-concurrency approach, with a 95% confidence level and a p-value of 0.9360, indicating that the difference between the two distributions is not statistically significant. This is a notable result, as WasteLess is the only approach among those evaluated that enables the use of concurrent functions without significant performance degradation compared to the no-concurrency scenario (i.e., the one with no contention). These results demonstrate that the configurations suggested by WasteLess yield response times statistically indistinguishable from the no-concurrency case.

Figure 23 illustrates the percentage difference in billable instances, and consequently the monetary cost, between configurations suggested by WasteLess and those obtained using the GCR instance scheduler, no-concurrency, and ProPack.

As expected, both ProPack and the GCR instance scheduler aggressively utilize concurrency limit, which results in lower operational costs. Consequently, WasteLess's median billable instances are approximately

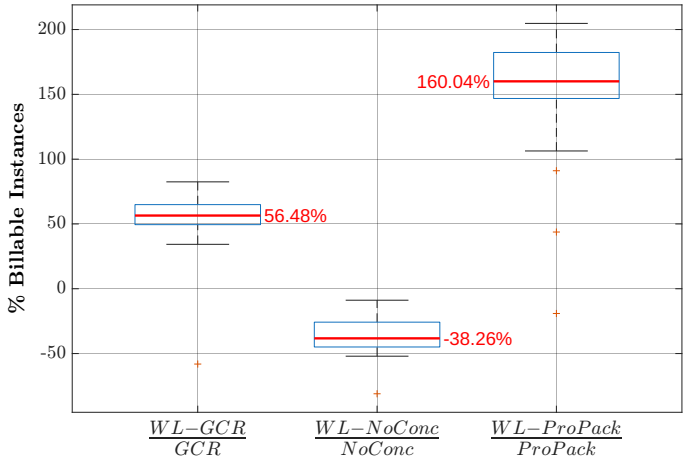



**Figure 23:** Percentage difference in billable instances between WasteLess (WL) and GCR, no-concurrency (NoConc), and ProPack.

$\sim$160% and $\sim$56% higher than those of ProPack and GCR, respectively. In contrast, when compared to the no-concurrency configuration which employs the highest number of function instances, WasteLess achieves significant savings with median billable instances being around $\sim$38% lower, a result that is statistically significant.

These results demonstrate that WasteLess achieves significant cost savings compared to the no-concurrency scenario, without any performance degradation. In contrast, while the GCR instance scheduler and ProPack configurations appear more frugal, they do so at the expense of severe performance penalties. By successfully navigating this trade-off, WasteLess delivers substantial cost benefits without compromising response time.

Table 6 presents statistics on the concurrency limit configured by WasteLess and ProPack in our experiments, compared to the GCR default concurrency limit. The results show that in over 95% of the randomly generated application instances, WasteLess set concurrency limit no higher than 19, while ProPack configured significantly higher values. This highlights the suboptimality of both the GCR default and ProPack, explaining

**Table 6:** Statistics of the concurrency limit computed by WasteLess and ProPack across all Acmeair variants. For GCR with default parameters and no-concurrency these values are set to 80 and 1, respectively.

| | $50^{th}$ | $95^{th}$ | $99^{th}$ |
|---|---|---|---|
| **WasteLess** | 2 | 19 | 145 |
| **ProPack** | 38 | 100 | 100 |

the severe response time degradation caused by excessive resource contention within function instances.

### 4.2.2 Large-Scale Evaluation with Complex Topologies

To further validate the broad applicability of our approach, we conducted experiments on a large and diverse suite of randomly generated FaaS applications. These applications were designed to incorporate more complex compositional constructs (probabilistic choice, asynchronous calls, and parallel fork/join patterns, as detailed in Section 1.2), thereby providing a more comprehensive and robust evaluation. For each experiment, the performance of WasteLess was compared against the self-adaptive strategies employed by GCR and no-concurrency. We excluded ProPack from this analysis due to its inferior performance in our initial tests, allowing us to focus squarely on production-grade algorithms.

#### Experiment Setup

The experimental infrastructure was hosted on a Compute Engine c2-standard-16 VM (16 vCPU, 64 GB RAM) running Ubuntu 24.04. This machine orchestrated the experiments, deployed the FaaS functions, and dynamically updated their configurations. We used Prometheus[3] and a Stackdriver Exporter for comprehensive, non-intrusive metric collection. The VM was located in the same region as the deployed functions to minimize network latency.

[3] `https://prometheus.io/`

**Table 7:** Statistics of the generated systems for each experimental mode. The table reports the mean number (± standard deviation) of functions, probabilistic choices, and call types (synchronous, asynchronous, and parallel).

| | **Synch** | **Async** | **Paral** |
|---|---|---|---|
| **Functions** | $9.60 \pm 4.37$ | $8.30 \pm 4.22$ | $9.70 \pm 4.29$ |
| **Prob. Choices** | $7.60 \pm 4.37$ | $6.30 \pm 4.22$ | $4.85 \pm 2.65$ |
| **Synch. Calls** | $47.90 \pm 40.89$ | $26.00 \pm 28.16$ | $51.40 \pm 40.78$ |
| **Asynch. Calls** | $2.95 \pm 4.24$ | $13.20 \pm 12.69$ | $0.00 \pm 0.00$ |
| **Parallel Calls** | $0.00 \pm 0.00$ | $0.00 \pm 0.00$ | $2.85 \pm 1.80$ |

To evaluate our approach, we generated a diverse set of FaaS systems using a four-step algorithm. First, the total number of functions in a system is randomly selected from a predefined interval. Second, these functions are arranged into a DAG, where the probability of an edge forming between any two nodes ($p_{call}$) was set to its maximum value of $1.0$. This ensures the creation of maximally dense graphs, thereby pushing the structural complexity of our test cases to its upper bound. Third, for each node with multiple outgoing edges, the algorithm decides whether to model it as a parallel fork/join construct or a probabilistic choice based on an input probability, $p_{par}$ ($0$ when omitted). Finally, all remaining edges are classified as either synchronous or asynchronous based on the probability $p_{asy}$ ($0$ when omitted). By tuning these parameters, we generated 60 systems, divided into three distinct experimental modes:

- **Synch**: 20 systems with a strong predominance of synchronous over asynchronous calls ($p_{asy} = 0.05$).
- **Async**: 20 systems with a significant portion of asynchronous calls ($p_{asy} = 0.4$).
- **Paral**: 20 systems where 40% of multi-path invocations occur in parallel and all calls are synchronous ($p_{par} = 0.4$).

As we already did in Section 4.2.1 we relied on *Locust* with the Twitter load traffic trace illustrated in Figure 20.

Table 7 presents the statistics of the generated systems, showing high variance in the number of functions and calls, which ensures a diverse set of evaluation topologies. The systems align with the characteristics of each mode: **Synch** is dominated by synchronous calls (96.67% on average), **Async** features a significant portion of asynchronous calls (34.18%), and **Paral** includes a substantial number of parallel functions invocations (33.80%).

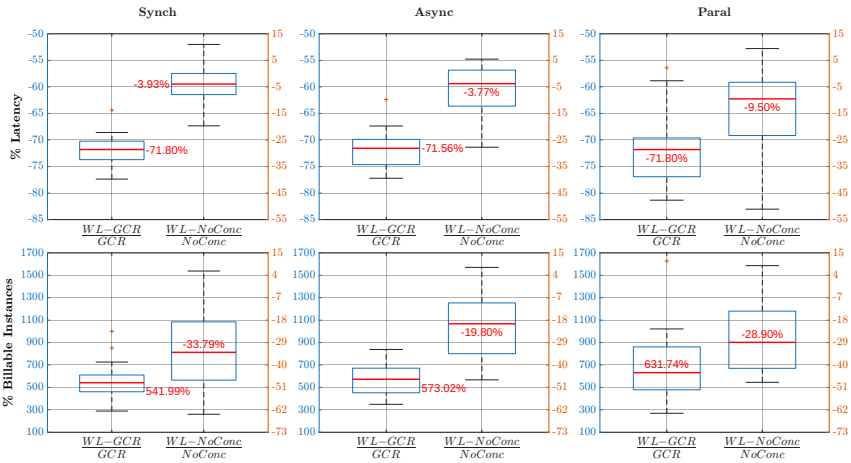



**Figure 24:** Percentage difference in latency (top row) and billable instances (bottom row) across three experimental modes: Synch, Async, and Paral (columns). Each subplot compares the performance of WasteLess (WL) against the GCR instance scheduler and the no-concurrency (NoConc) configuration. Median values for each distribution are highlighted in red.

### Results

The results from our experiments on randomly generated topologies are presented in Figure 24.

The data shows that WasteLess significantly outperforms GCR instance scheduler, reducing median latency by ∼72%. Importantly, WasteLess also improves upon the standard no-concurrency baseline, with latency reductions ranging from 3.8% in asynchronous-heavy systems

to 9.5% in systems featuring parallel calls. The lower improvement in asynchronous-heavy scenarios is an expected outcome, since the non-blocking nature of these calls inherently reduces the idle time that concurrency aims to leverage, thereby narrowing the window for optimization.

This performance advantage over no-concurrency is more pronounced than in Section 4.2.1, largely due to a more rigorous experimental methodology. Previously, requests not met with a "warm" instance could result in failures, which were not captured in the final latency metrics. This artificially lowered the average response time for configurations prone to cold starts. By implementing a request retransmission strategy, a call now persists until it is served by a ready instance. This crucial change transforms hidden failures into measurable high-latency events, accurately reflecting the performance cost of cold starts. This method, combined with a strict policy of beginning each run only after all billable instances returned to zero, ensures a more precise evaluation of how WasteLess's superior instance reuse mitigates such overhead.

The statistical significance of these performance differences was confirmed once again using the Kolmogorov-Smirnov test. In the comparison between WasteLess and GCR, the null hypothesis of the two being drawn from the same response time distribution was rejected with a 95% confidence level, yielding a p-value of 0.0026. Conversely, when comparing WasteLess with the no-concurrency baseline, the test did not reject the null hypothesis (p-value of 0.4973). This indicates that while WasteLess consistently improves median latency over no-concurrency, the overall shape of their response time distributions remains statistically similar, reinforcing that WasteLess achieves cost benefits without degrading the system's performance thus confirming the findings of Section 4.2.1.

In terms of cost, WasteLess provides a tangible decrease in operational expenses when compared to the standard no-concurrency deployment model (see Figure 24). It consistently reduces the number of billable instances, with median cost savings of 33.8% in Synch, 19.8% in Async, and 28.9% in Paral modes. While GCR appears more frugal (with WasteLess using approximately $5-7$ times more instances), this cost efficiency is achieved at the direct expense of the severe 72% performance degrada-

**Table 8:** Statistics of the concurrency limit computed by WasteLess across all experimental modes. The default for GCR is 80 and for no-concurrency is 1.

| | $50^{th}$ | $95^{th}$ | $99^{th}$ |
|---|---|---|---|
| **Synch** | 1 | 4 | 5 |
| **Async** | 1 | 3 | 4 |
| **Paral** | 3 | 10 | 16 |

tion noted earlier. This highlights that WasteLess offers a much more balanced and effective approach to cost management without sacrificing performance. WasteLess achieves significant cost savings over the no-concurrency model. Unlike the GCR scheduler, it provides this benefit without sacrificing application performance, thus delivering a tangible and balanced decrease in operational costs.

Table 8 again reveals the mechanism behind this success. WasteLess recommends conservative, tailored concurrency limit (e.g., a 95th percentile of just 3-4 for synchronous and parallel modes). This prevents the excessive resource contention caused by GCR's aggressive default setting, allowing WasteLess to unlock the benefits of concurrency without the drawbacks. When compared to our earlier results in Table 6, these concurrency limit settings are substantially lower. This reduction is a direct consequence of different service times and call patterns in the FaaS applications, and most importantly, our use of a lower target utilization ($U^t = 0.5$).

These findings demonstrate that precise concurrency management is crucial for balancing performance and cost. WasteLess excels by using conservative configurations to save costs without latency penalties—a balance its competitors fail to attain.

## 4.3 Chapter Summary

This chapter introduced WasteLess, a novel framework designed to tackle the complex resource provisioning challenge inherent in second-generation FaaS platforms. We began by detailing the architecture of the WasteLess

framework, which provides a holistic solution by jointly optimizing CPU, memory, and the concurrency limit. Unlike the online approach of $\mu$Opt, WasteLess employs a single, offline optimization step. By encoding a fluid approximation of the application's LQN model into a non-linear optimization problem, it efficiently computes an optimal and constant ratio between concurrency and CPU cores. This ratio, combined with a novel concurrency-to-memory mapping, yields a complete hardware and software configuration that guides the native FaaS platform scheduler toward a superior performance-cost equilibrium, all without the overhead of continuous runtime adaptation.

A comprehensive, two-part evaluation demonstrated the practical effectiveness of this approach:

1. **Comprehensive Evaluation on a Real-World Benchmark:** When benchmarked on the Acmeair application against commercial (GCR) and research (ProPack) baselines, WasteLess proved uniquely capable of leveraging concurrency without compromising performance. It successfully matched the latency of a standard no-concurrency deployment while reducing operational costs by approximately 38%, and significantly outperformed both the GCR scheduler and ProPack in terms of response time.

2. **Large-Scale Evaluation with Complex Topologies:** A second evaluation on a large suite of randomly generated applications, incorporating complex communication patterns, confirmed and strengthened these findings. WasteLess demonstrated robust and scalable performance, reducing median latency by approximately 72% compared to GCR and achieving cost savings between 20% and 34% relative to the no-concurrency baseline across all scenarios.

In summary, this chapter successfully established that the WasteLess framework provides a powerful and efficient solution to the second-generation FaaS configuration challenge. By outperforming both state-of-the-art research and widely used industrial solutions, we have validated our model-based, offline optimization methodology as a superior approach for achieving a balanced and effective performance-cost trade-off

in modern serverless environments.

# Chapter 5

# Conclusion

This thesis has investigated the critical challenge of efficient resource management in cloud-native systems, a problem that demands a delicate balance between application performance and operational cost (Mooghala, 2023; Alonso et al., 2023). We have demonstrated that the core difficulties in both microservice autoscaling and FaaS provisioning can be effectively addressed through a unified, model-based optimization methodology. By leveraging LQN fluid approximations, we developed computationally efficient frameworks that successfully compute optimal resource configurations.

This final chapter synthesizes the primary outcomes of this research. Section 5.1 summarizes the main contributions embodied by the $\mu$Opt and WasteLess frameworks. Section 5.2 provides a consolidated discussion of the limitations and threats to validity identified throughout our evaluations. Finally, Section 5.3 outlines promising directions for future research that build upon the foundations established in this work.

## 5.1 Summary of Contributions

The contributions of this thesis are embodied in two practical frameworks, $\mu$Opt and WasteLess, which apply a unified model-based optimization methodology to the distinct challenges of microservices and FaaS systems,

**Table 9:** Summary of experimental results for $\mu$Opt. The framework demonstrates superior efficiency in resource usage while maintaining or improving performance compared to research and industrial baselines.

| Scenario | Baseline | Performance ($\sim\Delta\%$) | Cost ($\sim\Delta\%$) |
|---|---|---|---|
| Vertical Scaling | ATOM | $+8$–$9\%$ (Throughput) | $-12$–$25\%$ (Cores) |
| Horizontal Scaling | HPA | $-29$–$38\%$ (Tracking Error) | $-21$–$30\%$ (Replicas) |

respectively.

First, in Chapter 3, we presented $\mu$Opt, a novel model-based autoscaler for microservices systems that efficiently finds optimal configurations based on the current workload. Capable of performing both vertical and horizontal scaling, it combines a fluid LQN model with NLP, enabling the fast computation of CPU shares, thread-pool sizes, and replica counts to maximize system performance while minimizing allocation cost.

Our evaluation demonstrated a speedup of three orders of magnitude for computing optimal configurations compared to the state-of-the-art research method ATOM. As summarized in Table 9, $\mu$Opt improved application throughput by $\sim 8$–$9\%$ while achieving significant cost savings of $\sim 12$–$25\%$ in terms of allocated cores. Furthermore, the comparison with the commercial-ready HPA showed that $\mu$Opt reduced replica usage by $\sim 21$–$30\%$ while achieving superior control precision, reducing the tracking error by $\sim 29$–$38\%$.

Second, in Chapter 4, we presented WasteLess, a framework that optimizes CPU, memory, and concurrency limit configurations for second-generation serverless functions. By leveraging performance models, WasteLess performs an single offline optimization to identify settings that minimize latency at the lowest cost. When integrated with GCR, WasteLess was comprehensively evaluated, proving its effectiveness.

As detailed in Table 10, on the Acmeair benchmark, WasteLess effectively matched the performance of a standard no-concurrency deployment ($\sim 1\%$ median latency reduction) while reducing billable costs by $\sim 38\%$. Furthermore, it significantly outperformed both the default GCR instance scheduler and the ProPack framework in terms of latency ($\sim 23\%$ and

**Table 10:** Summary of experimental results for WasteLess. Reported values represent the median percentage change compared to the respective baselines. Note that comparisons with GCR and ProPack show a cost increase because these baselines achieve lower costs only by violating QoS constraints.

| Scenario | Baseline | Latency ($\sim\Delta\%$) | Billable Cost ($\sim\Delta\%$) |
|---|---|---|---|
| Acmeair Benchmark | No-Concurrency | $-1\%$ | $-38\%$ |
| | GCR | $-23\%$ | $+56\%$[a] |
| | ProPack | $-56\%$ | $+160\%$[a] |
| Large-Scale Study | No-Concurrency | $-4$–$10\%$ | $-20$–$34\%$ |
| | GCR | $-72\%$ | $+542$–$632\%$[a] |

[a] Positive values indicate that WasteLess is more expensive than the baseline. However, the baseline achieves lower costs only by severe under-provisioning, resulting in unacceptable latency violations (QoS breach).

$\sim 56\%$ reduction, respectively).

In a large-scale study on complex, randomly generated applications, WasteLess proved both robust and highly scalable. It reduced median latency by $\sim 72\%$ compared to GCR and consistently improved upon the no-concurrency baseline in all tested scenarios, yielding simultaneous reductions in latency ($\sim 4$–$10\%$) and cost ($\sim 20$–$34\%$) through efficient cold start mitigation. Finally, it must be emphasized that any theoretical cost saving achieved by baselines through under-provisioning (indicated by the values marked with [a] in Table 10) is of no practical value, as it invariably results in severe latency degradation that violates standard QoS requirements.

## 5.2 Limitations and Threats to Validity

Despite the positive results, the methodology presented in this thesis has several limitations and threats to validity, which are consolidated here from our individual framework evaluations.

The primary threat to our methodology's validity is the potential inaccuracy of the LQN model. The model does not currently capture all

involved system dynamics, such as FaaS cold starts or scaling latency. As both $\mu$Opt and WasteLess compute optimal configurations based on this model, we cannot guarantee they will remain optimal when applied to the real system. We mitigated this by validating our methodology in production-ready environments like GKE and GCR, showing that the model accurately predicts performance under various conditions. Specifically for WasteLess, while FaaS cold starts are not explicitly modeled, we deduced the framework's effectiveness in mitigating them: the experimental results in Section 4.2.2 showed that setting the concurrency limit greater than one successfully reduces cold starts by promoting instance reuse.

Furthermore, the effectiveness of the proposed frameworks relies on the accurate creation and calibration of the underlying performance models. A significant challenge lies in the engineering effort required to architect the LQN model and the expertise needed to map complex cloud-native component interactions into a formal stochastic representation. While this dependence on a user-defined model may introduce errors or deployment delays, it can be mitigated by leveraging methodologies that learn LQN structures from execution traces (Israr, Woodside, and Franks, 2007). Similarly, our approach assumes the availability of high-fidelity monitoring data for calibration. In real-world production environments, traces can be noisy, incomplete, or subject to model drift as the application code evolves (Gama et al., 2014). Significant errors in service time estimation or a failure to update parameters after a new deployment could impair predictive accuracy, leading to suboptimal scaling decisions. Currently, the bootstrapping effort for a new application—comprising model design and initial profiling—represents a non-trivial manual overhead compared to model-free approaches like HPA, though it is compensated by the superior resource efficiency achieved once the model is operational.

From an implementation perspective, the current version of $\mu$Opt does not include explicit rollback strategies for severe model misestimation. While its MPC approach inherently self-corrects, a formal safety mechanism (e.g., reverting to a last-known-good configuration) could enhance its operational robustness. For WasteLess, the optimizations incur a com-

putational overhead; however, our experiments showed this to be on the order of milliseconds, making it negligible. While its profiling is the most time-intensive task, WasteLess still requires significantly fewer samples than state-of-the-art alternatives (Roy et al., 2023).

Finally, the scope of our experimental validation presents limitations. The comparison with the HPA in Chapter 3 used default parameters. Fine-tuning the HPA is a complex problem in itself (Al-Haidari, Sqalli, and Salah, 2013; Kang and Lama, 2020), and our use of defaults represents a threat to the comparison's fairness. While a more specialized configuration—specifically regarding the stabilization window and scaling policies—might narrow the performance gap, this comparison highlights the intrinsic value of $\mu$Opt. Unlike reactive controllers such as the HPA, which demand significant engineering effort and human intervention to identify optimal thresholds for specific workloads, $\mu$Opt automates this process through its model-based approach. In parallel, the WasteLess framework was evaluated exclusively on GCR. This was a deliberate choice, as GCR is the only major FaaS offering with native support for second-generation serverless functions. The approach should be portable, but this is unverified. Lastly, our methodology targets applications built entirely on the FaaS model, not hybrid systems combining serverless functions with containers. While our method could model individual functions in such a system, we anticipate some sub-optimality and leave a systematic investigation for future work.

## 5.3 Future Work

Building on the methodological foundations and frameworks established in this thesis, we outline a research roadmap structured around four strategic pillars. Beyond the natural progression of conducting more exhaustive experimental evaluations across a wider variety of applications, benchmark frameworks, and workloads, these future directions are specifically designed to (i) enhance the predictive accuracy of the core model, (ii) automate its creation to facilitate industrial adoption, (iii) bridge the gap between application-level sizing and infrastructure orchestration, and

(iv) expand the applicability of the unified methodology to emerging cloud-native paradigms.

**Theoretical Enhancements.** The primary direction is to extend the expressiveness of the underlying fluid model:

- *General Distributions:* We plan to extend the fluid formulation to support generally distributed service times, broadening applicability to systems where the exponential assumption does not hold.
- *Transient Analysis:* Solving the LQN in its transient regime (time-dependent solution) rather than steady-state would allow $\mu$Opt to explicitly model short-term phenomena, such as reconfiguration delays and cold starts, which are currently abstracted away in the steady-state analysis.

**Operational Autonomy and Industrial Integration.** To enhance practical adoption and minimize the human-in-the-loop requirement, the frameworks must evolve towards full operational autonomy:

- *Online Calibration:* We aim to integrate online estimation techniques (e.g., Kalman Filters) to continuously update model parameters (service times, routing probabilities) from live monitoring data. This would allow the system to adapt to performance changes without manual intervention, effectively addressing the drift caused by software updates or underlying infrastructure fluctuations, while also reducing the reliance on initial profiling.
- *Automated Modeling:* To further reduce the manual effort of model design, future work will explore the integration of static code analysis and service mesh telemetry (e.g., Istio, Linkerd) to automatically extract the LQN topology from source code.
- *Assumptions on Deployment Granularity:* Further investigation is needed to determine the optimal control loop frequency. While

$\mu$Opt solves the optimization problem in sub-second times, the practical sampling rate (typically every 15–60 seconds) must balance the responsiveness to workload bursts with the stability of the system to avoid oscillatory scaling behavior (thrashing).

**Orchestration Awareness and Distributed Environments.** A significant avenue for future research involves bridging the gap between application-level resource sizing and the underlying infrastructure orchestration. While this thesis primarily focuses on determining the optimal quantity of resources, future work could incorporate the specific placement decisions made by the orchestrator to account for node-level variability.

- *Performance-Aware Scheduling*: Integrating $\mu$Opt and WasteLess with the orchestration layer to enable joint optimization of resource quantity and placement. By leveraging model-derived performance sensitivities, the scheduler could make more informed decisions (e.g., co-locating tightly coupled services) to mitigate node fragmentation and hardware resource contention.

- *Geo-Distributed and Multi-Cluster Management*: Extending the framework to geographically distributed settings and edge-to-cloud continuums. By incorporating spatial constraints and inter-cluster WAN latencies into the optimization logic, the system could determine not only how much to scale a component but also the optimal geographic region for its deployment to balance performance with regional availability.

**Domain and Model Expansion.** Finally, the methodology can be transposed to emerging architectural paradigms and more granular cost models. While our approach is currently validated on microservices and second-generation FaaS, its structural assumptions define a specific applicability envelope that we aim to broaden:

- *Expanding the Applicability Envelope:* We aim to extend our model-based formulation to a broader spectrum of application classes. This entails modeling hybrid container/serverless systems, memory-sensitive services, and AI serving pipelines characterized by request batching and highly variable per-request demand.
- *$\mu$Opt beyond Microservices:* Building on the above expansions, we intend to apply the efficient autoscaling logic of $\mu$Opt to other emerging distributed paradigms where LQNs have proven effective, such as serverless computing.
- *Advanced WasteLess Cost Modeling:* We aim to expand the optimization model of WasteLess to consider additional FaaS cost factors beyond CPU and memory—such as invocation counts, maximum function runtime, and storage—to further enhance its optimization capabilities.

# Bibliography


Aderaldo, Carlos M et al. (2017). "Benchmark requirements for microservices architecture research". In: *IEEE/ACM 1st Int. Workshop Establishing Community-Wide Infrastructure Architecture-Based Softw. Eng.* Pp. 8–13.

Akhtar, N. et al. (2020). "COSE: Configuring Serverless Functions using Statistical Learning". In: *Proc. of 39th IEEE Conference on Computer Communications, INFOCOM '20*, pp. 129–138. DOI: 10.1109/INFOCOM41043.2020.9155363.

Akkus, Istemi Ekin et al. (2018). "{SAND}: towards {High-Performance} serverless computing". In: *2018 USENIX annual technical conference (USENIX ATC 18)*, pp. 923–935.

Alonso, Juncal et al. (2023). "Understanding the challenges and novel architectural models of multi-cloud native applications–a systematic literature review". In: *Journal of Cloud Computing* 12.1, p. 6.

Amiri, Amirali et al. (2022). "Cost-aware multidimensional auto-scaling of service-and cloud-based dynamic routing to prevent system overload". In: *IEEE Int. Conf. Web Services*, pp. 379–384.

Archive Team (2021). *Twitter Stream 2021-01*. https://archive.org/details/archiveteam-twitter-stream-2021-01. [Online; 2022-09-01].

*ATOM: Autoscaler for Microservices* (2023). URL: https://github.com/alimulgias/ATOM (visited on 05/16/2023).

Awad, Mahmoud and Daniel A Menasce (2017). "Deriving parameters for open and closed QN models of operational systems through black box optimization". In: *Proc. 8th ACM/SPEC Int. Conf. Perf. Eng.* Pp. 127–138.

Bai, Haoyu et al. (2024). "Drpc: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters". In: *IEEE Transactions on Services Computing*.

Baresi, L. et al. (2022). "NEPTUNE: Network- and GPU-aware Management of Serverless Functions at the Edge". In: *Proc. of Int'l Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '22*. ACM/IEEE, pp. 144–155. DOI: 10.1145/3524844.3528051.

Baresi, Luciano, Sam Guinea, et al. (2016). "A discrete-time feedback controller for containerized cloud applications". In: *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.* Pp. 217–228.

Baresi, Luciano, Davide Yi Xian Hu, et al. (2021). "KOSMOS: Vertical and horizontal resource autoscaling for kubernetes". In: *Service-Oriented Comput.: 19th Int. Conf.* Springer, pp. 821–829.

Baresi, Luciano and Giovanni Quattrocchi (2020). "A simulation-based comparison between industrial autoscaling solutions and cocos for cloud applications". In: *IEEE Int. Conf. Web Services*, pp. 94–101.

Barna, Cornel et al. (2018). "Runtime performance management for cloud applications with adaptive controllers". In: *ACM/SPEC Int. Conf. Perf. Eng.* Pp. 176–183.

Barrett, Enda, Enda Howley, and Jim Duggan (2013). "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud". In: *Concurrency Comput.: Pract. Experience* 25.12, pp. 1656–1674.

Berger, Vance W and YanYan Zhou (2014). "Kolmogorov–smirnov test: Overview". In: *Wiley statsref: Statistics reference online*.

Blinowski, Grzegorz, Anna Ojdowska, and Adam Przybyłek (2022). "Monolithic vs. microservice architecture: A performance and scalability evaluation". In: *IEEE Access* 10, pp. 20357–20374.

Bolch, Gunter et al. (2006). *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons.

Boyd, Stephen and Lieven Vandenberghe (2004). *Convex optimization*. Cambridge university press.

Cerny, Tomas et al. (2022). "Microservice architecture reconstruction and visualization techniques: A review". In: *IEEE Int. Conf. Service-Oriented Sys. Eng.* Pp. 39–48.

Chen, Tao and Rami Bahsoon (2015). "Self-adaptive trade-off decision making for autoscaling cloud-based services". In: *IEEE Trans. Serv. Comput.* 10.4, pp. 618–632.

D. Tollefson and A. Spyker (2015). *AcmeAir: A Sample Cloud Native Application*. https://github.com/AcmeAir/AcmeAir.

Das, A. et al. (2020). "Skedulix: Hybrid Cloud Scheduling for Cost-Efficient Execution of Serverless Applications". In: *Proc. of 13th IEEE Int'l Con-*

*ference on Cloud Computing, CLOUD '20*, pp. 609–618. DOI: 10.1109/CLOUD49709.2020.00090.

Daw, N., U. Bellur, and P. Kulkarni (2020). "Xanadu: Mitigating cascading cold starts in serverless function chain deployments". In: *Proc. of 21st Int'l Middleware Conference, Middleware '20*. ACM, pp. 356–370. DOI: 10.1145/3423211.3425690.

Denning, Peter J and Jeffrey P Buzen (1978). "The operational analysis of queueing network models". In: *ACM Computing Surveys (CSUR)* 10.3, pp. 225–261.

Dragoni, Nicola et al. (2017). "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Springer, Cham, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

Eismann, S. et al. (2021). "Sizeless: predicting the optimal size of serverless functions". In: *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*. Ed. by Kaiwen Zhang et al. ACM, pp. 248–259. DOI: 10.1145/3464298.3493398.

Eivy, Adam and Joe Weinman (2017). "Be wary of the economics of" serverless" cloud computing". In: *IEEE Cloud Computing* 4.2, pp. 6–12.

Elgamal, T. (2018). "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement". In: *Proc. of 2018 IEEE/ACM Symposium on Edge Computing, SEC '18*, pp. 300–312. DOI: 10.1109/SEC.2018.00029.

Fetzer, Christof (2016). "Building critical applications using microservices". In: *IEEE Security & Privacy* 14.6, pp. 86–89.

Fishman, George S and L Stephen Yarberry (1997). "An implementation of the batch means method". In: *INFORMS Journal on Computing* 9.3, pp. 296–310.

Florio, Luca and Elisabetta Di Nitto (2016). "Gru: An approach to introduce decentralized autonomic behavior in microservices architectures". In: *IEEE Int. Conf. Autonomic Comput.* Pp. 357–362.

Fowler, Martin and James Lewis (Mar. 2014). *Microservices: a definition of this new architectural term*. URL: https://martinfowler.com/articles/microservices.html (visited on 06/17/2024).

Franks, Greg et al. (2008). "Enhanced modeling and solution of layered queueing networks". In: *IEEE Trans. Soft. Eng.* 35.2, pp. 148–161.

Gama, João et al. (2014). "A survey on concept drift adaptation". In: *ACM computing surveys (CSUR)* 46.4, pp. 1–37.

Gandhi, Anshul et al. (2014). "Adaptive, model-driven autoscaling for cloud applications". In: *11th Int. Conf. Autonomic Comput.* Pp. 57–64.

Gao, Kaifeng et al. (2020). "Julia language in machine learning: Algorithms, applications, and open issues". In: *Comp. Sci. Rev.* 37, p. 100254.

Garbi, Giulio, Emilio Incerto, and Mirco Tribastone (2023). "$\mu$P: A Development Framework for Predicting Performance of Microservices by Design". In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 178–188.

Garcia, Carlos E, David M Prett, and Manfred Morari (1989). "Model predictive control: Theory and practice—A survey". In: *Automatica* 25.3, pp. 335–348.

Garriga, Hermann (2017). "Microservices in Practice: A Survey Study". In: *IEEE Software* 34.2, pp. 96–102.

Gias, A. U. and G. Casale (2020). "COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms". In: *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*. IEEE, pp. 1–8. DOI: 10.1109/MASCOTS50786.2020.9285966.

Gias, Alim Ul, Giuliano Casale, and Murray Woodside (2019). "ATOM: Model-driven autoscaling for microservices". In: *IEEE 39th Int. Conf. Distrib. Comput. Syst.* Pp. 1994–2004.

Google Cloud (2025). *Cloud Functions (2nd gen) overview*. https://cloud.google.com/run/docs/functions/comparison. Accessed: 2025-07-10.

Gotin, Manuel et al. (2018). "Investigating performance metrics for scaling microservices in cloudiot-environments". In: *ACM/SPEC Int. Conf. Perf. Eng.* Pp. 157–167.

Grimaldi, Domenico et al. (2015). "A feedback-control approach for resource management in public clouds". In: *IEEE GLOBECOM*, pp. 1–7.

Al-Haidari, Fahd, Mohammed Sqalli, and Khaled Salah (2013). "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources". In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 2. IEEE, pp. 256–261.

*HaProxy, The Reliable, High Performance TCP/HTTP Load Balancer* (2023). URL: http://www.haproxy.org/ (visited on 05/16/2023).

Hillston, Jane (1996). *A compositional approach to performance modelling*. Vol. 348. Cambridge University Press Cambridge.

*Horizontal Pod Autoscaler* (2024). Kubernetes.io. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visited on 09/30/2024).

Hossen, Md Rajib, Mohammad A Islam, and Kishwar Ahmed (2022). "Practical efficient microservice autoscaling with QoS assurance". In: *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput.* Pp. 240–252.

Inagaki, Tatsushi et al. (2022). "Detecting Layered Bottlenecks in Microservices". In: *IEEE 15th Int. Conf. Cloud Comput.* Pp. 385–396.

Incerto, Emilio, Roberto Pizziol, Gabriele Russo Russo, et al. (2025a). "Enhancing Performance-Cost Trade-off in Second-Generation FaaS Systems Through WasteLess". In: *IEEE Transactions on Services Computing*. Under review.

— (July 2025b). *Replication Package*. DOI: 10.5281/zenodo.15882810. URL: https://doi.org/10.5281/zenodo.15882809.

— (July 2025c). *The WasteLess Replication Package*. DOI: 10.5281/zenodo.15882810. URL: https://doi.org/10.5281/zenodo.15882809.

— (2025d). "Wasteless: An Optimal Provisioner for Self-Adaptive Second-Generation Serverless Applications". In: *2025 IEEE/ACM 20th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, pp. 61–72.

Incerto, Emilio, Roberto Pizziol, and Mirco Tribastone (2023). "$\mu$Opt: An Efficient Optimal Autoscaler for Microservice Applications". In: *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, pp. 67–76.

— (2025a). "Efficient Microservice Autoscaling through µOpt". In: *IEEE Transactions on Services Computing*. Under review.

— (Oct. 2025b). *The $\mu$Opt Replication Package*. DOI: 10.5281/zenodo.7771408. URL: https://doi.org/10.5281/zenodo.7771408.

Incerto, Emilio, Mirco Tribastone, and Catia Trubiani (2017). "Software performance self-adaptation through efficient model predictive control". In: *32nd IEEE/ACM Int. Conf. Automated Softw. Eng.* Pp. 485–496.

— (2018). "Combined vertical and horizontal autoscaling through model predictive control". In: *Euro-Par Parallel Process.: 24th Int. Conf. Parallel Distrib. Comput.* Springer, pp. 147–159.

Iqbal, Waheed, Mathew N Dailey, and David Carrera (2015). "Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications". In: *IEEE Sys. J.* 10.4, pp. 1435–1446.

Israr, Tauseef, Murray Woodside, and Greg Franks (2007). "Interaction tree algorithms to extract effective architecture and layered performance models from traces". In: *J. Sys. Softw.* 80.4, pp. 474–492.

Jawaddi, Siti Nuraishah Agos, Muhammad Hamizan Johari, and Azlan Ismail (2022). "A review of microservices autoscaling with formal verification perspective". In: *Software: Practice and Experience* 52.11, pp. 2476–2495.

Kang, Peng and Palden Lama (2020). "Robust resource scaling of containerized microservices with probabilistic machine learning". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, pp. 122–131.

Kaushal, Vishonika and Anju Bala (2011). "Autonomic fault tolerance using haproxy in cloud environment". In: *Int. J. Adv. Eng. Sci. Technol.* 7.2, pp. 54–59.

Khaleq, Abeer Abdel and Ilkyeun Ra (2021). "Intelligent autoscaling of microservices in the cloud for real-time applications". In: *IEEE Access* 9, pp. 35464–35476.

Khazaei, Hamzeh et al. (2017). "Elascale: Autoscaling and Monitoring as a Service". In: *Proc. 27th Annual Int. Conf. on Comput. Sci. and Softw. Eng.* CASCON '17. Markham, Ontario, Canada: IBM Corp., pp. 234–240.

Klimovic, Ana et al. (2018). "Pocket: Elastic ephemeral storage for serverless analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 427–444.

Kounev, Samuel et al. (2023). "Serverless computing: What it is, and what it is not?" In: *Communications of the ACM* 66.9, pp. 80–92.

Kurtz, Thomas G (1970). "Solutions of ordinary differential equations as limits of pure jump Markov processes". In: *J. Appl. Probability* 7.1, pp. 49–58.

Li, X. et al. (2022). "KneeScale: Efficient Resource Scaling for Serverless Computing at the Edge". In: *Proc. of 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid '22*, pp. 180–189. DOI: 10.1109/CCGrid54584.2022.00027.

Li, Yongkang et al. (2022). "Serverless computing: state-of-the-art, challenges and opportunities". In: *IEEE Transactions on Services Computing* 16.2, pp. 1522–1539.

Li, Z. et al. (2022). "Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing". In: *Proc. of 2022 USENIX Annual Technical Conference, ATC '22*, pp. 69–84.

Lin, C. and H. Khazaei (2021). "Modeling and Optimization of Performance and Cost of Serverless Applications". In: *IEEE Trans. Parallel Distributed Syst.* 32.3, pp. 615–632. DOI: 10.1109/TPDS.2020.3028841.

Lin, C., N. Mahmoudi, et al. (2023). "Fine-Grained Performance and Cost Modeling and Optimization for FaaS Applications". In: *IEEE Trans. Parallel Distributed Syst.* 34.1, pp. 180–194. DOI: 10.1109/TPDS.2022.3214783.

Lin, Changyuan and Hamzeh Khazaei (2020). "Modeling and optimization of performance and cost of serverless applications". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3, pp. 615–632.

Little, John DC (1961). "A proof for the queuing formula: L= $\lambda$ W". In: *Operations research* 9.3, pp. 383–387.

Little, John DC and Stephen C Graves (2008). "Little's law". In: *Building intuition: insights from basic operations management models and principles*, pp. 81–100.

Liu, Fangming and Yipei Niu (2023). "Demystifying the Cost of Serverless Computing: Towards a Win-Win Deal". In: *IEEE Transactions on Parallel and Distributed Systems*.

Liu, Jianshu, Shungeng Zhang, and Qingyang Wang (2023). "$\mu$ConAdapter: Reinforcement Learning-based Fast Concurrency Adaptation for Microservices in Cloud". In: *Proceedings of the 2023 ACM Symposium on Cloud Computing*. SoCC '23, pp. 427–442. DOI: 10.1145/3620678.3624980.

Mahgoub, A., E. B. Yi, K. Shankar, S. Elnikety, et al. (2022). "ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs". In: *Proc. of 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI '22*, pp. 303–320.

Mahgoub, A., E. B. Yi, K. Shankar, E. Minocha, et al. (2022). "WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows". In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.2, 26:1–26:28. DOI: 10.1145/3530892.

Mahmoudi, Nima and Hamzeh Khazaei (2020). "Performance modeling of serverless computing platforms". In: *IEEE Transactions on Cloud Computing* 10.4, pp. 2834–2847.

Marler, R Timothy and Jasbir S Arora (2010). "The weighted sum method for multi-objective optimization: new insights". In: *Structural and Multidisciplinary Optimization* 41.6, pp. 853–862.

MathWorks (2023). *ode15s*. URL: https://www.mathworks.com/help/matlab/ref/ode15s.html (visited on 05/16/2023).

McClure, Joe (2023). *AcmeAir MicroServices Springboot*. URL: https://github.com/blueperf/acmeair-mainservice-springboot (visited on 05/16/2023).

Meng, Chunyang, Shijie Song, et al. (2023). "Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 53–65.

Meng, Chunyang, Jingwan Tong, et al. (2022). "HRA: An Intelligent Holistic Resource Autoscaling Framework for Multi-service Applications". In: *IEEE Int. Conf. Web Services*, pp. 129–139.

Mooghala, Sridhar (2023). "A Comprehensive Study of the Transition from Monolithic to Micro services-Based Software Architectures". In: *Journal of Technology and Systems* 5.2, pp. 27–40.

Moreno, Gabriel A et al. (2015). "Proactive self-adaptation under uncertainty: A probabilistic model checking approach". In: *Joint Meeting Found. Soft. Eng.*

Murty, Katta G and Santosh N Kabadi (1985). *Some NP-complete problems in quadratic and nonlinear programming*. Tech. rep.

Nguyen, Thanh-Tung et al. (2020). "Horizontal pod autoscaling in kubernetes for elastic container orchestration". In: *Sensors* 20.16, p. 4621.

Oakes, Edward et al. (2018). "{SOCK}: Rapid task provisioning with {Serverless-Optimized} containers". In: *2018 USENIX annual technical conference (USENIX ATC 18)*, pp. 57–70.

Petriu, Dorina C and Hui Shen (2002). "Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications". In: *Comp. Perf. Eval.: Model. Techn. Tools: 12th Int. Conf., Proc. 12*. Springer, pp. 159–177.

Podolskiy, Vladimir, Anshul Jindal, and Michael Gerndt (2018). "IaaS reactive autoscaling performance challenges". In: *IEEE 11th Int. Conf. Cloud Comput.* Pp. 954–957.

Qian, H. et al. (2022). "RobustScaler: QoS-Aware Autoscaling for Complex Workloads". In: *Proc. of 38th IEEE International Conference on Data Engineering, ICDE '22*, pp. 2762–2775. DOI: 10.1109/ICDE53745.2022.00252.

Qiu, Haoran et al. (2020). "{FIRM}: An Intelligent Fine-grained Resource Management Framework for {SLO-Oriented} Microservices". In: *Proc. 14th USENIX Symp. Operating Sys. Des. Implementation*, pp. 805–825.

Qu, Chenhao, Rodrigo N Calheiros, and Rajkumar Buyya (2018). "Autoscaling web applications in clouds: A taxonomy and survey". In: *ACM Comput. Surv.* 51.4, pp. 1–33.

Quattrocchi, Giovanni et al. (2024). "Autoscaling Solutions for Cloud Applications under Dynamic Workloads". In: *IEEE Transactions on Services Computing*, pp. 1–17. DOI: 10.1109/TSC.2024.3354062.

Razavi, Kamran et al. (2022). "FA2: Fast, accurate autoscaling for serving deep learning inference with SLA guarantees". In: *IEEE 28th Real-Time Embedded Technol. Appl. Symp.* Pp. 146–159.

Richardson, Chris (2018). *Microservices Patterns: With examples in Java*. Manning Publications.

Rosen, Rami (2013). "Resource management: Linux kernel namespaces and cgroups". In: *Haifux, May* 186, p. 70.

Rossi, Fabiana, Valeria Cardellini, and Francesco Lo Presti (2020). "Hierarchical scaling of microservices in Kubernetes". In: *IEEE Int. Conf. Autonomic Comput. Self-Organizing Sys.* Pp. 28–37.

Roy, R. B. et al. (2023). "ProPack: Executing Concurrent Serverless Functions Faster and Cheaper". In: *Proc. of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '23*. ACM, pp. 211–224. DOI: 10.1145/3588195.3592988.

Russo, G. Russo et al. (2022). "Towards QoS-aware function composition scheduling in Apache OpenWhisk". In: *Proc. of 1st Workshop on Serverless Computing for Pervasive Cloud-Edge-Device Systems and Services, *LESS '22*. IEEE. DOI: 10.1109/PerComWorkshops53856.2022.9767299.

Schirmer, Trever et al. (2024). "FUSIONIZE++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization". In: *IEEE Transactions on Cloud Computing*.

Sedghpour, Mohammad Reza Saleh et al. (2023). "Hydragen: A microservice benchmark generator". In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pp. 189–200.

Shahrad, Mohammad et al. (2020). "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider". In: *2020 USENIX annual technical conference (USENIX ATC 20)*, pp. 205–218.

Shen, J. et al. (2021). "Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms". In: *Proc. of 41st IEEE International Conference on Distributed Computing Systems, ICDCS '21*, pp. 194–204. DOI: 10.1109/ICDCS51616.2021.00027.

Silva, P., D. Fireman, and T. E. Pereira (2020). "Prebaking Functions to Warm the Serverless Cold Start". In: *Proc. of 21st Int'l Middleware Conference, Middleware '20*. ACM, pp. 1–13. DOI: 10.1145/3423211.3425682.

Sun, Yi et al. (2023). "A Review of Kubernetes Scheduling and Load Balancing Methods". In: *2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*. IEEE, pp. 284–290.

Tariq, A. et al. (2020). "Sequoia: enabling quality-of-service in serverless computing". In: *Proc. of ACM Symposium on Cloud Computing, SoCC '20*, pp. 311–327. DOI: 10.1145/3419111.3421306.

Toffetti, Giovanni et al. (2017). "Self-managing cloud-native applications: Design, implementation, and experience". In: *Future Gener. Comput. Sys.* 72, pp. 165–179.

Tollefson, Doug and Andrew Spyker (2021). *Acme Air Sample and Benchmark*. URL: https://github.com/acmeair/acmeair.

Tong, Jingwan et al. (2021). "A holistic auto-scaling algorithm for multi-service applications based on balanced queuing network". In: *IEEE Int. Conf. Web Services*, pp. 531–540.

Tribastone, Mirco (2012). "A fluid model for layered queueing networks". In: *IEEE Trans. Soft. Eng.* 39.6, pp. 744–756.

Tribastone, Mirco, Philip Mayer, and Martin Wirsing (2010). "Performance prediction of service-oriented systems with layered queueing networks". In: *Leveraging Appl. of Formal Methods, Verification, and Validation: 4th Int. Symp. Proc., Part II 4*. Springer, pp. 51–65.

Wächter, Andreas and Lorenz T Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical programming* 106.1, pp. 25–57.

Waizmann, Tabea and Mirco Tribastone (2016). "DiffLQN: differential equation analysis of layered queuing networks". In: *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*, pp. 63–68.

Wajahat, Muhammad et al. (2019). "Mlscale: A machine learning based application-agnostic autoscaler". In: *Sustain. Comput.: Inform. Sys.* 22, pp. 287–299.

Wang, Runan, Giuliano Casale, and Antonio Filieri (2022). "Enhancing Performance Modeling of Serverless Functions via Static Analysis". In: *Service-Oriented Computing*. Cham: Springer Nature Switzerland, pp. 71–88. ISBN: 978-3-031-20984-0.

Wang, Yilun et al. (2024). "FaaSConf: QoS-aware Hybrid Resources Configuration for Serverless Workflows". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 957–969.

Wang, Ziliang et al. (2022). "DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems". In: *Proc. 13th Symp. Cloud Comput.* Pp. 16–30.

Woodside, Murray and Greg Franks (2002). *Tutorial introduction to layered modeling of software performance*.

Wright, Margaret (2005). “The interior-point revolution in optimization: history, recent developments, and lasting consequences”. In: *Bulletin of the American Mathematical Society* 42.1, pp. 39–56.
Yu, Guangba, Pengfei Chen, and Zibin Zheng (2019). “Microscaler: Automatic scaling for microservices with an online learning approach”. In: *IEEE Int. Conf. Web Services*, pp. 68–75.