# Learning-Centric, Payload and QoS-Aware Function Resource Configuration and Management in Serverless Computing

Siddharth Agarwal

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE, AUSTRALIA

December 2025

ORCID: 0000-0003-0944-8314

# Learning-Centric, Payload and QoS-Aware Function Resource Configuration and Management in Serverless Computing

Siddharth Agarwal
*Principal Supervisor: Prof. Rajkumar Buyya*
*Co-Supervisor: Dr. Maria Read*

---

**Abstract**

In recent years, the ***serverless*** cloud computing paradigm has seen a remarkable adoption for building cloud-native and micro-service-oriented applications. Unlike traditional cloud models, serverless abstracts away the complexities of infrastructure management by offloading these responsibilities to the Cloud Service Provider (CSP). This fundamental paradigm shift empowers developers to dedicate their efforts to core application business logic, significantly enhancing productivity. The primary execution model for serverless applications is Function-as-a-Service (FaaS), an event-driven compute service that executes fine-grained, stateless code segments called "functions."

While FaaS offers a compelling economic advantage with pay-per-use billing, it still faces significant challenges. The issues begin with runtime inefficiencies like cold starts, where a new function instance must be initialised, introducing latency that degrades performance. This is exacerbated by autoscaling, which rapidly provisions new instances during traffic spikes but quickly becomes a performance bottleneck. These performance issues are tied to the static resource configurations developers must provide. Because platforms often link resource allocation solely to a single metric, like memory, optimising for performance directly conflicts with minimising operational cost. This rigidity not only forces developers into suboptimal configuration choices but also prevents the underlying platform from dynamically and efficiently managing its shared resources, including scheduling, placement, and autoscaling, to meet varying request demands.

To address these limitations, this thesis investigates novel architectures, approaches, and techniques to identify and exploit learning-centric function management opportunities to improve operational cost, system throughput and alleviate resource wastage in FaaS environments without compromising application-specific performance Service

Level Objectives (SLO). This thesis advances the state-of-the-art in payload- and Quality-of-Service (QoS)-aware function resource configuration and management by making the following key contributions:

1. A comprehensive taxonomy for dynamic function configuration and management in FaaS, and classifying existing literature based on the identified aspects and factors of function resource management.

2. An on-policy Reinforcement Learning (RL) approach for reducing the function cold start frequency in serverless computing to improve platform throughput and resource utilisation.

3. A proactive autoscaling framework to exploit partially-observable serverless environments to alleviate system load while boosting function performance.

4. A dynamic Machine Learning (ML) technique for predicting optimal function configuration to significantly reduce operational cost, and resource wastage.

5. A comprehensive, fully-integrated serverless system to manage and optimise function execution lifecycle for adaptive resource allocation, and function scheduling decisions.

6. An interactive simulation engine and a visualiser to provide insights into the user-abstracted resource management of FaaS by implementing various request routing and function placement policies.

7. A detailed discussion outlining challenges and the potential future work for the efficient use of serverless computing environments.

# Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Siddharth Agarwal, December 2025

# Preface

## Main Contributions

This thesis research has been carried out in Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2-7 and are based on the following publications:

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "Dynamic Function Configuration and its Management in Serverless Computing: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)* [Under Review, September 2025].

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "On-Demand Cold Start Frequency Reduction with Off-Policy Reinforcement Learning in Serverless Computing", *Proceedings of the 2024 International Conference on Computational Intelligence and Data Analytics (ICCIDA)*, Pages: 1-24, Hyderabad, India, June 28-29, 2024.

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "A Deep Recurrent-Reinforcement Learning Method for Intelligent AutoScaling of Serverless Functions," in *IEEE Transactions on Services Computing*, Volume: 17, Number: 5, Pages: 1899-1910, September-October, 2024.

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "Input-Based Ensemble-Learning Method for Dynamic Memory Configuration of Serverless Computing

Functions", in *Proceedings of the 17th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, Pages: 346-355, Sharjah, UAE, December 16-19, 2024.

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "Saarthi: An End-to-End Intelligent Platform for Optimising Distributed Serverless Workloads", *Proceedings of the 40th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, New Orleans, USA, May 25-29, 2026 [Under Evaluation, September 2025].

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "Serv-Drishti: An Interactive Serverless Function Request Simulation Engine and Visualiser", in *Proceedings of the 35th IEEE International Telecommunications Networks and Applications Conference*, Christchurch, New Zealand, November 26-28, 2025.

# Acknowledgements

My doctoral journey, a marathon of intellectual challenge and personal growth, simply would not have been possible without the unwavering support, belief, and sacrifice of many remarkable individuals.

First and foremost, my heartfelt, deepest gratitude belongs to my supervisors, Professor Rajkumar Buyya and Dr. Maria Read. Over these long, demanding years, your guidance was not just invaluable but was the compass that gave me direction and the anchor that provided stability. Thank you for your patience, your rigorous feedback, and the relentless encouragement that helped me transform ideas into contributions. Further, my gratitude goes to my current PhD advisory committee chair, Professor Udaya Parampalli, and ex-committee chair, Professor George Buchanan, for their invaluable support towards a timely completion of this work.

My heart also goes to all the past and present members of the qCLOUDS Laboratory at the University of Melbourne. We shared countless hours of caffeine-fuelled brainstorming, the frustration of failed experiments, and the joy of breakthrough moments. This unique environment fostered not just colleagues, but true friendships. In particular, I thank Dr. Shashikant Ilager, Dr. Mohammad Goudarzi, Dr. Samodha Pallewatta, Dr. Amanda Jayanetti, Dr. Anupama Mampage, Dr. Tharindu B. Hewage, Hoa Nguyen, Jie Zhao, Ming Chen, Zhiyu Wang, Duneesha Fernando, Thakshila Imiya Mohottige, Qifan Deng, TianYu Qi, Hootan Zhian, Murtaza Rangwala, Yifan Sun, Prabhjot Singh, Haoyu Bai, Abhishek Sawaika, and Avishka Sandeepa, for the support and our friendship fostered over the years.

I would be nowhere without the unconditional love and support of my family. To my Father, who consistently believed in my potential and made every opportunity possible, your faith has been a driving force in my educational journey. To my Mother, whose steadfast support has accompanied me from the earliest years of my learning to this final milestone, your strength has been my constant source of resilience.

To my friend, my confidant, my brother, Anvay Pandey. Thank you for being a permanent fixture in my life for the last 12 years. I owe you a special debt of gratitude for pushing me to start this PhD, for always believing in me, and for supporting me through every challenge and distance. Your support has meant more than words can ever capture. To Duneesha Fernando, thank you for being the truest kind of family here in Australia. You are not just a friend, but the sister I always had, marked by the significance of Rakhi you tie each year. Your love for our baby, Harvin, and your presence during our most demanding moments truly made Australia a home away from

home.

A most special thanks is reserved for my loving wife, Poorvi. You have been my constant companion, my emotional rock, and my safe harbour throughout this entire demanding journey. Thank you for being there through every up and down, and for the resilience and quiet strength shared through all these years. Thank you for being the calm to my chaos and home to my wandering thoughts. And finally, thank you for the most precious gift of my lifetime, our baby, Harvin. Thank you, my little one, for bringing such pure joy, charm, immense love, and laughter into our lives, reminding me daily what truly matters. You are the brightest chapter of my story.

I acknowledge the University of Melbourne for providing me with the scholarship and resources to pursue my doctoral studies. This work is also supported by Australian Research Council grants. I would like to sincerely thank the past and present admin staff of the School of Computing and Information Systems for their support.

This thesis is not just a record of my research; it is a testament to the collective support, sacrifice, and belief of everyone mentioned here, and many more who may not be named but have left an indelible mark on my life. Signing this off by thanking each and every individual from the bottom of my heart, thank you.

*Siddharth Agarwal*
*December 2025, Melbourne, Australia*

# Contents

# List of Figures

# List of Tables

# List of Acronyms

**AI** Artificial Intelligence

**IoT** Internet of Things

**ML** Machine Learning

**LLM** Large Language Models

**SLA** Service Level Agreement

**SLO** Service Level Objectives

**CPU** Central Processing Unit

**GPU** Graphics Processing Unit

**IaaS** Infrastructure-as-a-service

**VM** Virtual Machines

**SC** Serverless Computing

**FaaS** Function-as-a-Service

**PaaS** Platform-as-a-Service

**SaaS** Software-as-a-Service

**CSP** Cloud Service Provider

**OpenFaaS-CE** OpenFaaS (Community Edition)

**LSTM** Long-Short Term Memory

**GRU** Gated Recurrent Unit

**QoS** Quality-of-Service

**RFR** Random Forest Regression

**RL** Reinforcement Learning

**AWS** Amazon Web Services

**CS** Cold Start

**TTL** Time-to-Live

**DAG** Directed Acyclic Graph

# Chapter 1
# Introduction

The cloud computing landscape has recently experienced a fundamental paradigm shift, driven by the migration of applications from monolithic structures to fine-grained, micro-service-oriented architectures [1]. In traditional cloud computing service models, such as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS), the responsibility for provisioning, scaling, and managing the underlying infrastructure is partially or wholly borne by the user. To facilitate the rapid deployment and operational agility required by modern micro-services, an advanced execution model was needed to drastically reduce developer overhead. Serverless computing has emerged as the defining paradigm that meets this demand, shifting the obligation for resource provisioning and management entirely to the cloud service provider (CSP) [2]. According to reports [3][4], the global serverless market was valued at over 24.55 billion US dollars in 2024 and is expected to grow at a compound annual growth rate (CAGR) of 14.15% by the year 2034. However, contrary to its name, serverless still requires computing resources and is fundamentally defined by its extensive infrastructure abstraction and its consumption-based pricing model.

Although precursors to this concept, such as Google App Engine in 2008, offered early forms of PaaS abstraction, the modern serverless paradigm was formally established with the public release of AWS Lambda in late 2014 [5][6]. Lambda introduced the core execution model of FaaS, which offers an event-driven, on-demand compute service. In FaaS, a *function* is the fundamental unit of development and deployment. A function is a stateless code segment holding business logic, wrapped inside an execution environment, that is invoked in response to associated events or *triggers*. When a function is invoked, the platform instantiates a new execution environment, also referred

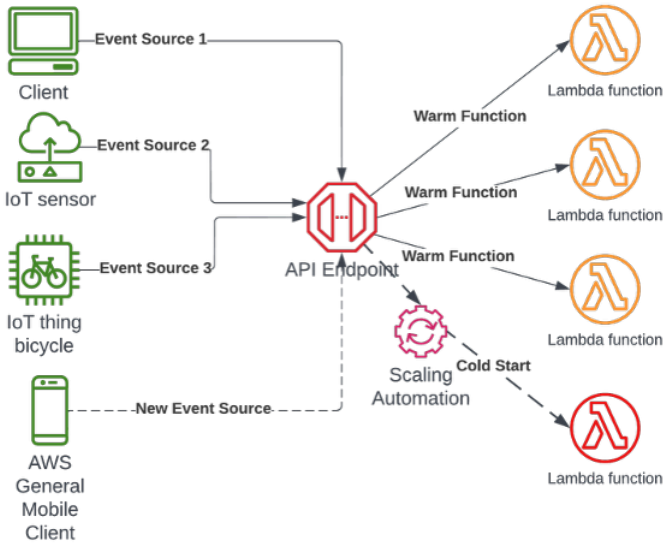



**Figure 1.1:** A Sample FaaS Application Environment

to as a *function instance*, which is typically a microVM or a lightweight container that encapsulates the logic, its dependencies, and the allocated resources. Following this, leading CSPs have also proposed their serverless platforms, such as Azure Functions [7] and Cloud Run Functions [8]. In addition to these proprietary solutions, open-source frameworks such as OpenFaaS [9], Knative [10], and Apache OpenWhisk [11] have also been released. Technically, in FaaS, function instances are provisioned dynamically in response to incoming requests when existing instances are busy executing other requests. Different platforms enforce different concurrency models, which define how many concurrent invocation requests a single function instance can process. This concurrency control helps balance performance and resource utilisation. Moreover, users are charged strictly based on the actual execution duration of the function code and the resources consumed during execution, minimising the expenses associated with idle capacity. A sample FaaS application workflow is demonstrated in Fig. 1.1.

This reactive, on-demand creation of multiple function instances comes with a significant overhead called cold start, where each new function instance must be bootstrapped from scratch [12]. A cold start is the added latency to the actual execution time of a function when the execution environment initialises for the first time, involving critical

stages such as code download, dependency setup, network overlay creation, and language runtime startup. The duration of cold starts can range from milliseconds to a few seconds [13], which is sometimes comparable to the execution time of applications that require ultra-fast responses. Not only does this impact user-perceived latency but also imposes substantial pressure on service providers to respond swiftly to demand fluctuations, allocate resources efficiently, and schedule new functions for immediate execution [14]. The negative consequences of cold starts are further exacerbated in applications with multiple interconnected functions, where each instance's initialisation can have a ripple effect on overall performance and operational costs.

Cold starts are an intrinsic consequence of the dynamic autoscaling mechanisms inherent to FaaS platforms, where the system automatically provisions and adjusts the number of function instances in response to fluctuating incoming traffic. When new requests arrive and all existing instances are either busy or specific scaling thresholds such as request queue length, response latency, or CPU utilisation [6][8] are exceeded, the platform spins up additional instances to maintain service responsiveness. This reactive, event-driven horizontal scaling is foundational to serverless architectures, but each instance creation introduces a cold start which increases application response times, particularly affecting the latency-sensitive workloads. Additionally, proprietary FaaS platforms implement scaling based on distinct primitives, influencing concurrency handling and resource allocation strategies. For example, AWS Lambda [1] [6] enforces strict single-concurrency isolation, dedicating resources to one request per container, which simplifies function scaling logic but can lead to rapid container churn during traffic spikes. In contrast, Azure Functions and Google Cloud Run [7] adopt configurable multi-concurrency models, allowing a single instance to efficiently process several concurrent requests by sharing memory and CPU resources. While these approaches may enhance resource utilisation and reduce operational costs, they introduce new considerations for application design such as managing shared-state and potential resource contention when deploying functions intended for high-throughput or bursty event streams.

FaaS is fundamentally designed to relieve developers from the complexities of man-

[1] referring to the original AWS Lambda service model without the managed instances [15]

aging underlying infrastructure. However, developers are typically responsible for configuring the function instance resources statically, specifying critical settings such as function memory, CPU share, ephemeral storage, and maximum function running time. These configuration settings, particularly the allocated memory, have a profound impact on function execution success and cold start latency [16]. This is primarily because, in most industry platforms, increasing the allocated memory proportionally increases available computing resources [2]. Consequently, allocating higher resources results in a performance benefit as it speeds up the function instance's initialisation cycle (resulting in faster cold starts) and accelerates the function's execution time. Moreover, this significantly influence resource allocation and scheduling decisions from both user and provider perspectives [2]. Under- or over-provisioning of resources directly impact a function's performance and operational cost, as a function is billed for the duration the resources are used for during execution. In addition to this, it also impacts a provider's ability to pack and place function instances on the underlying infrastructure. The relationship between performance and cost is inherently non-linear, meaning that even small changes in allocated resources can result in significant variations in performance and operational expense [17]. This presents a complex trade-off between resource allocation, operational cost and execution time that developers need to address as they strive for optimal function performance while balancing competing priorities. Furthermore, workload features such as payload or the input (its size, type or number of inputs) received by the function, play a significant role in determining the minimum resource requirements for a successful function execution [18]. For instance, functions that process large inputs/payloads may necessitate increased resource allocation, leading to amplified performance and operational costs. This, in turn, highlights the importance of precisely configuring resources to optimise function performance while minimising waste. Furthermore, the interplay between these factors introduces another layer of complexity, as even small changes in resource allocation can have profound effects on overall system behaviour, making resource configuration a crucial yet highly complex decision step that requires meticulous attention.

In serverless, where the underlying infrastructure is abstracted, the prompt alloca-

[2] general trend of industry FaaS platforms where more memory allocates more compute resources

tion and scheduling of function resources pose a significant challenge for CSPs as they strive to maintain a high degree of utilisation while minimising resource waste [19]. This complexity arises from the need to allocate and schedule finer-grained resources quickly, which can lead to increased resource fragmentation if not managed properly [20]. Additionally, FaaS environments are highly multi-tenant, which means that the resource configuration of one function directly impacts the scheduling decisions for co-located functions, potentially leading to poor performance as these functions may contend for shared resources [21]. This scenario is exacerbated by the fact that FaaS platforms typically do not configure or scale functions based on specific workload requirements, thereby missing an opportunity to optimise FaaS for a broader range of application domains. As a result, CSPs must develop sophisticated strategies for function scheduling and resource management to mitigate these challenges and ensure high-performance, low-latency, and cost-effective experiences.

While serverless computing has transformed the way applications are built and deployed, its limitations in terms of resource allocation and management, i.e., inherent cold starts, function configuration, reactive autoscaling and black-box scheduling continue to hinder widespread adoption from both developer and provider perspectives [22][23]. To bridge this gap, this thesis explores novel adaptive and autonomous techniques for function resource configuration and management. The research focuses on optimising the dual objectives of assuring predictable QoS and minimising operational cost for the developer, and maximising resource utilisation and reducing the operational complexity faced by the CSP in resource allocation and provisioning behind the scenes. To achieve this, we conduct an extensive survey of the current state of serverless resource management, examining existing solutions, their limitations, and areas for improvement. We then develop innovative proposals for efficient techniques in resource scaling, allocation, and scheduling, which are designed to optimise the utilisation of this computing environment. The efficacy and performance improvements of these proposals are verified through thorough evaluations. Thus, the objective of our research is to provide a comprehensive understanding of the challenges and opportunities in serverless resource management, and to develop advanced solutions that can satisfy the evolving needs of both developers and providers. By leveraging these techniques and approaches, we aim

to create a truly serverless experience where users are completely freed from resource management tasks, and providers can focus on delivering high-quality services with minimal overhead.

## 1.1 Challenges of Function Resource Management in Serverless Computing

Despite the profound benefits and rapid market adoption of FaaS, the inherent complexity of dynamically managing execution at scale introduces significant technical challenges that undermine its promise of seamless performance and minimal cost. The intricate network of interdependent factors, combining static function configurations by developers and the provider's black-box management, demonstrates the fundamental difficulty in simultaneously optimising performance, latency, and operational cost in the FaaS environment. This thesis focuses on four critical and interrelated challenges within the FaaS ecosystem, as explained in the following subsections.

### 1.1.1 The Cold Start Problem

The FaaS model is fundamentally designed for absolute ephemerality where provisioning a new function instance for each request and de-allocating resources happens immediately after service for individual requests [24]. However, in practice, commercial offerings (e.g., AWS Lambda, Azure Functions, Google Cloud Functions) and open-source frameworks (e.g., OpenFaaS [9] and Knative [10]) deviate from this ideal. They employ resource retention policies to keep instances active for a limited duration, enabling a warm start to serve subsequent requests with minimal latency [12]. However, a surge in demand or a request arriving after a period of idleness necessitates an on-demand instantiation or a cold start, introducing non-negligible delays to the user-perceived latency [25][26]. The frequency of cold starts is governed by the platforms proprietary de-allocation policy (often a dynamic process that is not fixed or transparent to the user) and the function's invocation pattern. For instance, sparsely distributed requests across time are highly susceptible to cold start effects due to the inherent resource recycling

after a period of inactivity. Furthermore, the cold start duration is highly dependent on both application-specific factors (programming language, runtime environment, and code deployment size) and the function's static resource configuration (allocated CPU and memory) [27] [28][29] [30]. According to [12], function cold starts can be addressed in two ways, either by reducing the initialisation time (reactive strategy) or by reducing the frequency or number of potential cold starts (proactive strategy). The reactive approach focuses on optimising the function loading and runtime environment so that when the platform creates new instances in response to requests, the initialisation delay is as short as possible. The proactive approach anticipates incoming workload demand by pre-initialising or pre-warming a set of function instances ready to serve requests, thereby avoiding cold starts altogether [14]. Although simpler solutions like maintaining active instances alive or maintaining idle function containers, can alleviate initial cold starts but impose overhead costs and resource waste if many instances remain idle while waiting for requests. This trade-off between resource utilisation and idle cost demonstrates why cold start mitigation remains a fundamental system-level challenge in FaaS, requiring careful balancing by CSPs and users.

### 1.1.2 Reactive Autoscaling

The critical need to mitigate the cold start problem and dynamically match demand is constrained by the limitations of traditional autoscaling solutions. FaaS platforms primarily employ reactive, rule-based horizontal scaling mechanisms that instantiate new instances only after a performance threshold has been exceeded [31]. CSPs implement this reactiveness via differing policies. For instance, AWS Lambda uses a highly granular, proprietary, request-based scaling policy, while platforms like Google Cloud Run Functions and Azure Functions often utilise resource-based scaling (e.g., CPU utilisation) or target-based concurrency thresholds. However, these reactive scaling strategies inherently introduce a delay between demand surge and resource availability, which in turn exacerbates cold start occurrences during sudden workload spikes. This delay leads to request queuing, throttling, and a sharp increase in user-perceived latency. Critically, this mechanism also significantly raises the risk of failed requests due to the client-side

request Time-to-Live (TTL) expiring before the function can execute. Furthermore, these threshold-based systems demand expert domain knowledge for manual threshold tuning and are susceptible to complexities like hysteresis (temporal dependency on past states), where scaling decisions temporarily depend on previous scaling states that contribute to unpredictable system behaviour. Crucially, the effectiveness of these reactive models is undermined by the partial observability of the cloud environment. The lack of complete, real-time internal metrics of the multi-tenant platform introduces significant delays and unreliability in reflecting the exact system state at any given time [32]. This environmental constraint prevents reactive scaling from adapting and responding to changing demand. Therefore, the unpredictability caused by cold starts combined with the partial runtime visibility complicates building accurate performance models, which inhibit the efforts toward reliable performance-based SLOs for latency-sensitive applications.

### 1.1.3 The Cost-Performance-Configuration Trade-Off

A central conflict of balancing performance and operational cost arises from the developer's responsibility to provide static resource configurations in an intrinsically dynamic FaaS environment. The nature of this configuration varies by platform; for instance, AWS Lambda requires developers to define a function's memory size, which the platform then uses to proportionally allocate other critical resources, such as CPU and network bandwidth. Conversely, other commercial FaaS platforms, such as those offered by Google and Microsoft, require selecting a pre-defined combination of resources (memory, CPU, etc.) that will be applied to all instances during autoscaling. This static, one-size-fits-all allocation is fundamentally sub-optimal because a functions resource requirement varies dynamically based on input payload size and workload complexity [18][33]. This sub-optimality leads to either over-provisioning (wasting resources) or under-provisioning (throttling performance and risking execution failure due to resource exhaustion). Exacerbating the issue, some environments permit concurrent execution on a single instance, leading to fine-grained resource contention [34]. Developers, operating with limited observability into execution, often resort to speculative or ad-hoc

resource configuration decisions, hoping to meet performance SLOs. This challenge is further heightened by the fundamental cost versus latency trade-off [20] where allocating more memory (and thus CPU) may lead to faster execution, but the non-linear execution speedup exhibits diminishing marginal returns, subsequently raising the operational cost. While open-source frameworks like OpenFaaS and Knative may offer fine-grained resource tuning, allowing resources like CPU and memory to be configured independently, this flexibility further complicates the resource optimisation challenge faced by developers. Therefore, the intricate task of determining an optimal resource configuration necessitates moving beyond manual profiling and static allocation. It is critical to explore solutions that enable optimal or near-optimal dynamic function configuration while effectively satisfying the complex, inter-dependent performance-cost constraints [35].

### 1.1.4 Workload-Agnostic Function Management

The collective weight of the aforementioned challenges is amplified by the FaaS platform's black-box management and its workload-agnostic scheduling. The CSP executes a complex, opaque process including resource provisioning, scheduling, and function placement, without providing users clear visibility into these internal operations. This lack of transparency combined with the inherent platform constraints limits the optimisation of critical factors such as data locality, inter-function communication patterns, and the ability to leverage multiple co-existing function configurations efficiently. This systemic failure is further compounded when developers explore adopting workload-aware, i.e., input-aware configurations, leading to version proliferation (the creation of multiple function versions tailored to varying request payloads). This shift introduces a complex multi-objective operational challenge for the CSP's scheduler. On one hand, deploying precise function versions helps users reduce operational cost by avoiding over-provisioned resources. On the other hand, this proliferation increases the risk of performance overheads due to frequent and expensive cold starts or function failures resulting from under-provisioned execution. Consequently, the increased number of distinct instances causes excessive resource fragmentation across the multi-tenant in-

frastructure, severely hindering the CSP's ability to achieve efficient resource pooling and bin-packing [20]. Therefore, the core challenge is to strike a balance, whether to reuse an in-place instance that potentially satisfies the performance and cost SLOs, or to instantiate a new, tailored version at the cost of a cold start overhead. The lack of an integrated, automated framework to navigate this dynamic trade-off and manage these conflicting objectives defines the central challenge.

## 1.2 Research Questions and Objectives

The central premise of serverless computing, the radical shift of infrastructure management responsibility from the developer to the service provider, creates an entirely new set of operational complexities that challenge its foundational value proposition. The efforts of the CSP to provide a seamless, highly automated *server-less* experience puts forward crucial opportunities to investigate and design intelligent mechanisms that can minimise user interference while maximising both resource utilisation and performance predictability. The objective of this thesis is to fundamentally address the constraints of resource management in the FaaS environment by proposing adaptive and workload-aware techniques for managing the entire function lifecycle. We aim to design solutions that simultaneously satisfy the conflicting constraints of performance, operational cost, and resource utilisation from the dual perspective of both the developer and the CSP. In order to meet these objectives and bridge the gaps identified across cold start mitigation, reactive scaling, static configuration, and workload-agnostic scheduling, we formulate and address the following four key research questions:

- Q1. *How can proactive, learning-based techniques reduce the cold start frequency in FaaS platforms while improving resource utilisation as compared to widely adopted solutions?* The cold start problem is an intrinsic challenge of FaaS's reactive, on-demand execution model. Latency is introduced as the platform provisions the runtime environment (code, dependencies, etc.) for new requests. Cold start frequency is governed by the function's workload and the CSP's opaque de-allocation policies, which quickly release idle resources without accounting for intermittent invocation patterns. While workarounds like minimal provisioned concurrency or pro-

prietary keep-alive policies exist, functions still incur significant, unpredictable latencies. This challenge imposes a dual burden of resulting in degraded QoS for the user, and simultaneously reducing resource utilisation for the provider by blocking non-computing, idle resources. Therefore, moving beyond widely adopted reactive systems, a proactive, learning-based approach is warranted to predict and manage cold start frequency while ensuring optimal resource utilisation.

- Q2. *How can RL models be integrated with recurrent policies to manage complex autoscaling in partially-observable FaaS environments to optimise for long-term throughput and better execution latency?* The current CSP approach to function scaling is defined by reactive, threshold-based autoscaling that relies on monitored metrics. However, the effectiveness of this mechanism is fundamentally constrained because (1) multi-tenant execution introduces interference effects, (2) architectural abstractions produce inherent performance variability, and (3) platform visibility is limited or delayed. Collectively, these constraints lead to partial observability. The scaling mechanism's failure to consider dynamic workload patterns, despite industry data [13] suggesting functions often follow recursive invocation traces, creates a significant gap in optimisation. Furthermore, scaling actions are sequentially dependent, meaning current decisions impact future environment states (known as the hysteresis effect). Therefore, a sophisticated approach incorporating adaptive mechanisms is required to effectively manage the complexity and partial observability characteristic of FaaS environments. The goal is to develop an adaptive autoscaling solution that optimises for long-term throughput and guarantees better execution latency, i.e., QoS, by learning from workload patterns and sequential scaling outcomes.

- Q3. *How can the function configuration trade-off be solved by designing a resource allocation model that accurately predicts the optimal resource requirements based on dynamic workload features such as function input or payload?* The function configuration trade-off originates from the fact that a function's resource requirements dynamically change with varying workload features, specifically its input or payload [18][33]. Since function performance and operational cost are intrinsically

linked to resource allocation, the satisfaction of SLO targets and the delivery of acceptable QoS become highly volatile under the current static configuration model. Developers are thus forced to make ad-hoc decisions, leading to both costly over-provisioning and performance-throttling under-provisioning. Therefore, a dynamic resource allocation scheme is essential to designing a function input-aware model that can accurately predict and configure the optimal set of resources (primarily memory) for each execution request. Crucially, the objective is to approach this as a multi-objective optimisation problem, ensuring functions execute within budget constraints while maintaining high QoS.

- Q4. *How can an end-to-end serverless framework integrate input-aware resource predictions and a multi-objective optimisation with adaptive request orchestration to ensure performance continuity for dynamic workloads?* The current operational inefficiency in FaaS platforms is exacerbated by proprietary solutions that fail to integrate the influence of dynamic function inputs into workload management and orchestration. When input-aware resource allocation is pursued, developers risk creating a version proliferation, i.e., a fleet of functions with diverse configurations or tailored versions that may not be reused. This outcome creates a critical trade-off of an increased cold start effect (due to instantiating a new, specific version for seemingly every new input) versus under-utilisation of existing, idle instances that may still have enough resources to satisfy the current execution within SLO constraints. Therefore, an end-to-end serverless framework must be developed that integrates input-aware resource predictions with an adaptive request orchestration mechanism. This orchestration must intelligently solve a constant conflict between exploration (instantiating new configurations to find optimal performance-cost trade-offs) and exploitation (reusing existing, idle instances to mitigate cold starts and save resources). Crucially, the system must use a multi-objective optimisation approach that considers long-term workload patterns and resource allocation decisions to provide QoS assurance and ensure performance continuity, thus bridging the gap toward a self-driving serverless system that guarantees performance-cost efficiency.

## 1.3 Thesis Contributions

In addressing the research problems discussed in Section 1.2, this thesis makes the following contributions:

1. Presents a taxonomy encompassing aspects of function configuration and resource management in FaaS settings along with factors that influence function design, configuration, run-time cost, and performance guarantees from both a developers and service providers perspective, and a classification of existing literature using the proposed taxonomy.

2. Investigates a model-free Q-learning agent to exploit resource utilisation, available function instances, and platform response failure rate to reduce the number of cold starts for CPU-intensive serverless functions within the SLO constraints (addresses the Q1).

    - Analyses function resource metrics such as CPU utilisation, available instances, and the proportion of unsuccessful responses to propose a Q-learning model that dynamically analyses the application request pattern and improves function throughput by reducing frequent cold starts on the platform.
    - Presents a brief overview of explored solutions to address function cold starts and highlight the differences between contrasting approaches to the proposed agent.
    - Performs experiments on a real-world system setup and evaluate the proposed RL-based agent against the default resource-based policy and a baseline keep-alive technique.
    - Produces an open-source release of the implemented environment for application of RL in practical Kubernetes-based FaaS setups, accompanied by detailed documentation, provided for the benefit of the research community.

3. Investigates the application of Recurrent Neural Networks (RNN), specifically Long-Short Term Memory (LSTM) in a model-free Partially Observable Markov Decision Process (POMDP) setting for function autoscaling in FaaS to find a balance between conflicting CSP and user objectives (addresses the Q2).

- Analyses the characteristics of a FaaS environment to model autoscaling decisions as a POMDP. A hypothesis formation that scaling decisions have a sequential dependence on interaction history to propose a POMDP model that captures function metrics.
- Develops an open-source, Gymnasium-compatible, RNN-based RL agent that captures the temporal dependency of scaling actions and workload complexity to predict scaling actions.
- Implements a practical setup using OpenFaaS to perform experiments and evaluate the proposed LSTM-integrated Proximal Policy Optimisation (PPO) approach against the state-of-the-art PPO algorithm, commercially offered threshold-based horizontal scaling, OpenFaaS' request-per-second scaling policy, and a Deep Recurrent Q-Network i.e., DRQN, to demonstrate LSTM-PPO's ability to capture environment uncertainty for efficient scaling of serverless functions.
- Produces an open-source release of the implemented framework, accompanied by detailed documentation, provided for the benefit of the research community.

4. Develops and proposes an Input-Aware Resource Allocation Model using ensemble learning (such as Multi-Output Random Forest Regression) to solve the static configuration challenge (addresses the Q3).

    - Establishes an evidence-based correlation of input-awareness versus the SLO constraint satisfaction and QoS fulfilment for function configuration in FaaS
    - Presents a dynamic resource prediction model that precisely captures the non-linear relationship between dynamic workload features (e.g., input payload features) and optimal function resource requirements (e.g., memory and CPU allocation).
    - Formalises a multi-objective optimisation problem to select a memory configuration that minimises run-time cost and function execution time, while guaranteeing successful execution within specified deadline and budget constraints.

- Develops an agent built with multi-output Random Forest Regression (RFR) model, trained on collected metrics, to infer the optimal input-aware resource predictions while optimising for conflicting objectives.
- Implements a practical setup on AWS Lambda and other commercial serverless service to deploy, evaluate and utilise the proposed agent in production.
- Produces an open-source release of the implemented functions and experimental results, accompanied by detailed documentation, provided for the benefit of the research community.

5. Proposes an integrated, self-driving FaaS optimisation framework (addresses the Q4).

   - Integrates the predictive models from Q3 into a cohesive, end-to-end serverless optimisation framework (Saarthi) capable of orchestrating requests across diverse function versions.
   - Proposes a hybrid optimisation strategy that solves the core trade-off between exploration (cold starting new, optimised versions) and exploitation (reusing idle, in-place instances), ensuring the optimal execution path for every incoming request.
   - Integrates a fault-tolerant request handling with a $G/G/c/K$ buffer to temporarily queue requests and prevent immediate dropping of requests with a retry mechanism to improve system resilience.
   - Proposes an Integer Linear Programming (ILP)-based function optimiser that continuously monitors the cluster and adjusts the number of function instances (version proliferation), mitigating resource fragmentation while adapting to dynamic workloads and assuring high QoS.
   - Proposes a redundancy-based, fault tolerant function scaling mechanism to ensure service continuity and faster service recovery by mitigating mis-predictions and instance failures.
   - Implements a complete framework, built atop OpenFaaS, to automate and integrate multiple aspects of serverless function management, and evaluate its

performance against the baseline OpenFaaS using the established benchmark functions.

- Produces an open-source release of the implemented framework, accompanied by detailed documentation, provided for the benefit of the research community.

6. Proposes a lightweight, interactive simulation engine and a visualiser.

   - Implements a Javascript based simulation engine and visualiser tool that models the end-to-end request flow through a serverless environment by abstracting key operational components into a layered architecture.
   - Integrate different configuration parameters for realisation of various FaaS platforms and frameworks, including the underlying compute infrastructure and the function resource requirement model.
   - Enable different request routing logic and function placement strategies to be tested, compared and visualised against each other.
   - Presents critical design decisions for the request processing in a FaaS environment including the queue management and the cold start impact on the performance.
   - Produces an open-source release of the implemented simulation tool that is accompanied by detailed documentation and guides to plug-in custom routing and placement logic into the visualiser, provided for the benefits of the research and academic community.

## 1.4 Thesis Organization

The structure of this thesis is shown in Figure 1.2. The rest of this thesis is organised as follows:

- Chapter 2 presents a taxonomy and literature review on the function configuration and resource management aspects of serverless computing environments. This chapter is partially derived from:

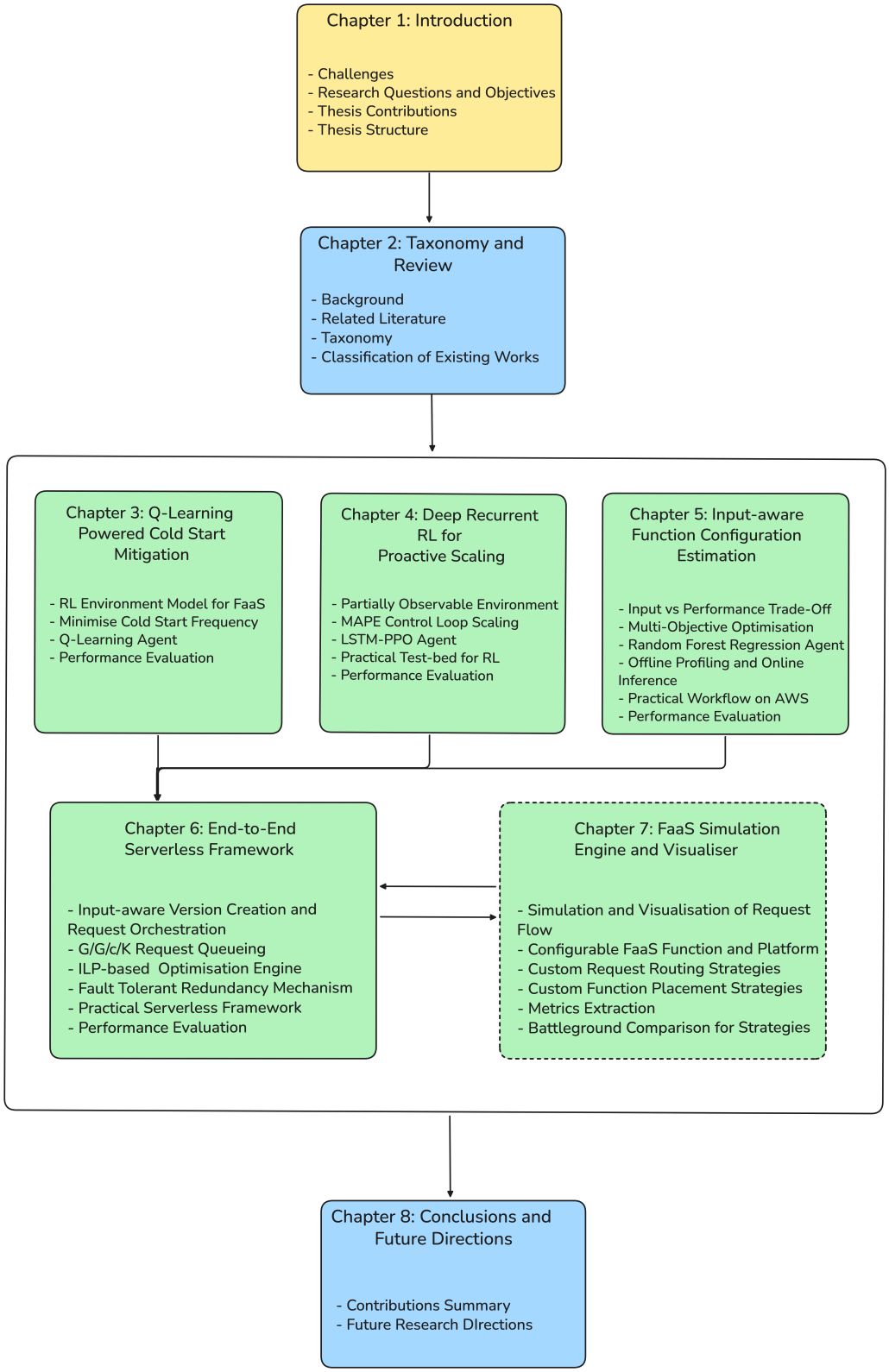



**Figure 1.2:** Thesis Structure

  - **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "Dynamic Function Configuration and its Management in Serverless Computing: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)* [Under Review, September 2025].

- Chapter 3 presents an on-policy RL approach for reducing the function cold start frequency in serverless computing to improve platform throughput and resource utilisation. This chapter is derived from:

  - **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, "On-demand Cold Start Frequency Reduction with Off-Policy Reinforcement Learning in Serverless Computing," *in Proceedings of the 2024 International Conference on Computational Intelligence and Data Analytics (ICCIDA 2024, Springer, Singapore),* Hyderabad, India, June 28-29, 2024.

- Chapter 4 presents a proactive autoscaling framework to exploit partially-observable serverless environments to alleviate system load while boosting function performance. This chapter is derived from:

  - **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, "A Deep Recurrent-Reinforcement Learning Method for Intelligent Autoscaling of Serverless Functions", *IEEE Transactions on Services Computing*, Volume: 17, Number: 5, Pages: 1899-1910, September, 2024.

- Chapter 5 presents a dynamic ML technique for predicting optimal function configuration to significantly reduce operational cost, and resource wastage. This chapter is derived from:

  - **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, "Input-Based Ensemble-Learning Method for Dynamic Memory Configuration of Serverless Computing Functions", *in Proceedings of the 17th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2024),* Sharjah, UAE, December 16-19, 2024.

- Chapter 6 presents a comprehensive, fully integrated serverless system to manage and optimise function execution lifecycle for smarter resource allocation, and function scheduling decisions. This chapter is derived from:
  - **Siddharth Agarwal**, Maria Rodriguez Read, and Rajkumar Buyya, "Saarthi: An End-to-End Intelligent Platform for Optimising Distributed Serverless Workloads", *40th IEEE International Parallel and Distributed Processing Symposium,* May 25-29, 2026, New Orleans, USA.[Under Evaluation, October, 2025]
- Chapter 7 presents a javascript-based simulation engine and visualisation tool for FaaS environments that mimic and model the flow of a request through the FaaS infrastructure, enabling an insight into the black-box resource management. This chapter is derived from:
  - **Siddharth Agarwal**, Maria Rodriguez Read, and Rajkumar Buyya, "ServDrishti: An Interactive Serverless Function Request Simulation Engine and Visualiser", in *Proceedings of the 35th IEEE International Telecommunications Networks and Applications Conference*, Christchurch, New Zealand, November 26-28, 2025.
- Chapter 8 concludes the thesis by summarising its findings and outlining potential future research directions.

# Chapter 2

# A Taxonomy on Function Configuration and Management

*This chapter investigates the fundamental challenges of optimal resource allocation and management within the serverless computing paradigm, focusing specifically on FaaS. We first examine the unique trade-off of FaaS, where the developers configure their functions with minimum operational resources which has a direct impact on runtime performance and resource management decisions such as scaling. This is complicated by a lack of platform transparency, forcing developers to rely on expert knowledge or experience-based ad-hoc decisions for configuring and managing function resources. Consequently, achieving optimal resource management while adhering to performance constraints remains a non-trivial task. We then identify different resource management aspects and propose a taxonomy of factors that systematically influence function design, configuration, operational cost, and performance guarantees from both developer and service provider perspective. Following this, we conduct a comprehensive survey and in-depth literature analysis of existing resource management works in FaaS to classify them under the proposed taxonomy.*

## 2.1 Introduction

The function abstraction in FaaS allows complex applications to be decomposed and developed as a highly decoupled combination of multiple fine-grained functions, directly representing the organisational business logic. These functions significantly enhance developer productivity by eliminating infrastructure management overhead and enabling a rapid time-to-market. While FaaS was initially optimised for highly bursty, com-

This chapter is partially derived from:

- **Siddharth Agarwal**, Maria A. Rodriguez and Rajkumar Buyya, "Dynamic Function Configuration and its Management in Serverless Computing: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)* [Under Review, September 2025].

pute and memory-intensive workloads, its successful application has now expanded into latency-sensitive and compute-intensive domains, event-driven website [36], video streaming platform [37], multi-media processing [38], CI/CD pipeline [39], AI/ML inference task [40], and Large-Language-Model (LLM) query processing [41]. However, this foundational abstraction does not translate to transparent or guaranteed performance. The user-perceived latency, execution time, and operational cost of a FaaS function are crucially dependent on a complex and interacting combination of factors. These critical variables can be broadly categorised into the static, developer-defined configurations and the dynamic, platform-controlled runtime parameters. The developer-defined configurations include the chosen runtime environment (e.g., Python, Node.js), function code dependencies, and the crucial static resource configuration assigned by the developer (e.g., allocated memory, which often dictates CPU share). On the other hand, platform-controlled parameters are governed by the CSP's proprietary scheduling and runtime management strategies, such as the initial cold start latency, the configured resource quota and concurrency limits, and the system's underlying reactive autoscaling policies.

For instance, the static resource configuration assigned by the developer fundamentally drives function behaviour. Allocating more memory to a function generally results in a faster execution time because serverless platforms typically provide a proportionally higher share of CPU and network bandwidth alongside more memory [42][43]. This trade-off, however, means that allocating more resources may result in a higher operational cost. The relationship between cost and resource allocation is non-linear and non-monotonic as the economic benefit does not scale predictably with the amount of resources assigned. This is critical as it relies on the function's actual resource utilisation during execution. Moreover, the initial cold start latency of a function is also significantly influenced by the allocated resources. Increased memory usually reduces cold start latency [44][45] by affording more CPU power for faster instance initialisation, code loading, and runtime setup. Beyond resource allocation, the language runtime choice heavily impacts cold start latency and overall function performance. For example, interpreted languages like Node.js and Python generally exhibit faster initialisation times than compiled languages like Java or .NET [16][46].

Beyond the developer's assigned configurations, the performance and latency of a FaaS function are fundamentally governed by the platform's proprietary and largely opaque control plane. Behind the scenes, the CSP's resource management stack executes a challenging, multi-step process for every request, which is fully abstracted from the user. First, the resources must be allocated and provisioned, and then the function must be scheduled and placed onto the underlying execution infrastructure (Virtual Machines (VM) or micro-container), where eventually the code is executed. This black-box management introduces significant dynamics that directly impact user experience. The constant need for fine-grained resource allocation and placement decisions, especially given the fluctuating demand and ephemeral nature of function instances, can lead to delays in finding appropriate underlying nodes, delays in autoscaling, and subsequent queuing or waiting of incoming requests. This unpredictability in the runtime environment results in variable execution latency. Consequently, CSPs rarely offer Service Level Agreement (SLA) guarantees for individual function performance (e.g., latency), focusing instead on high-level platform availability and up-time.

This chapter investigates the fundamental challenges of resource management in FaaS. The rest of the chapter is organised as follows: Section 2.2 presents the background discussion on serverless computing and FaaS execution model, its key characteristics and popular FaaS platforms. Section 2.3 presents the proposed taxonomy for function resource configuration and management, while Section 2.4 classifies the surveyed literature based on the proposed taxonomy. Section 2.5 concludes the chapter with a summary of our taxonomy.

## 2.2 Background

To better understand the research problems addressed in this thesis, this section presents relevant background on serverless computing, including key characteristics and limitations. Furthermore, we discuss the function-based operational model of FaaS, its configuration and management aspects, and motivation for adaptive resource management in serverless computing environments.

### 2.2.1 Serverless Computing Paradigm

Cloud computing has fundamentally revolutionised how organisations approach and access Information Technology (IT) infrastructure. It provides on-demand resource availability such as compute, storage, and database services delivered via a consumption-based, pay-as-you-go pricing model. Traditionally, cloud services were segmented into three primary models, each defined by a differing level of shared responsibility between the cloud provider and the user. In Infrastructure-as-a-service (IaaS), services like AWS EC2 [47] require developers to rent core compute infrastructure, where they retain the responsibility for managing everything above the hypervisor, including the Operating System (OS) installation, patching, runtime configuration, virtual networking, and security. This adds substantial operational overhead for the developers in addition to building applications. While Platform-as-a-Service (PaaS) services such as AWS Elastic Beanstalk [48] offer a managed runtime environment, abstracting the OS and middleware, they still require developer involvement in application deployment and maintenance, often mandating a minimum number of running instances. Furthermore, Software-as-a-Service (SaaS) delivers a fully managed software application (e.g., Salesforce [49]), representing the highest level of abstraction.

While IaaS and PaaS offer significant advantages to developers in building, deploying and managing applications, they share two major, residual limitations. First, they both require at least a minimum dedicated infrastructure in an always-on state, meaning users are charged for up-time hours regardless of actual utilisation. This commitment to maintaining idle capacity incurs high operational costs for the user as they are being billed regardless of utilisation, and translates into inefficient resource allocation for providers as they must reserve the infrastructure even when not fully utilised. Second, developers still bear the burden of managing and optimising the runtime environment or scaling policies. This crucial gap of needing to eliminate idle resource costs while achieving maximal abstraction warranted a new computing paradigm. This necessity led to the emergence of Serverless computing. Serverless is a computing paradigm that promotes the abstraction of the underlying infrastructure, where the provider assumes the entire burden of resource management and other related activities. However, since its inception in 2012, serverless has evolved into an architectural philosophy or strategic

mindset that enables developers to focus on business value rather than worrying about the underlying infrastructure [50]. It adheres to three core principles - no infrastructure management, seamless autoscaling, and consumption-based pricing. This model ensures that users are only charged for the resources consumed during execution, enabling a cheaper and highly elastic execution model.

The realisation of the serverless paradigm is achieved through two primary service categories: compute and backend services. FaaS is the core compute model. It provides an event-driven, on-demand runtime environment specifically designed for executing stateless, short-lived code segments containing an application's core business logic. Complementary to FaaS is Backend-as-a-Service (BaaS), which represents a suite of provider-managed services, including storage (e.g., AWS S3 [51]), databases (e.g., AWS DynamoDB [52]), and messaging queues that also strictly adhere to the serverless principles of no management and consumption-based billing. Together, FaaS and BaaS constitute an end-to-end serverless paradigm, enabling the seamless and dynamic provisioning, allocation, and scaling of all necessary application resources. While the BaaS components streamline data and storage operations, the dynamic and ephemeral nature of the FaaS compute model introduces unique and critical challenges concerning latency, resource allocation, and cost efficiency. Therefore, this chapter, and the subsequent thesis, will concentrate exclusively on the FaaS paradigm, exploring its execution model, key characteristics, and the critical resource management challenges it entails.

### 2.2.2 Function-as-a-Service Execution Model and Key Characteristics

The FaaS execution model governs how various serverless compute platform work and react to the generated requests. Therefore, to generalise the mechanics of FaaS platforms, including proprietary services as well as open-source frameworks, we formalise the system as an event-triggered, state-driven environment where each incoming request corresponds to an associated ephemeral invocation governed by strict resource, scheduling and concurrency or quota rules. The model consists of three primary entities that provide the means for invocation execution; namely, a Function Registry ($F$), the Worker Pool ($W$) and the Controller. A function registry is a collection of functions $F_i$

registered with the service provider, where each $F_i$ is a static definition that encapsulates the business logic or code package. Each function is associated with a static configuration of resources ($C_i$), generally memory ($mem_i$) and other compute resources like CPU ($cpu_i$) are proportionally allocated. This is packaged as an isolated execution environment - a function instance, implemented via a microVM or container technology. These function instances are explicitly instantiated and functions are invoked in response to a triggering event ($R_i$). A function instance maybe configured with a concurrency or function capacity, $M_i$, that dictates the number of simultaneous invocation that can be handled by a single function instance. In addition to this, providers usually put a threshold on the maximum allowed execution time, known as a timeout, $T_i$. The worker pool, $W$, represents the set of infrastructure hosts or compute nodes that provide the physical resources to host the function instances. A controller is the central unit that acts as the entry point to the system, manages traffic ingestion, scaling decisions and has resource scheduling responsibilities.

When a request or event $R_i$ for function $F_i$ arrives at the controller, the system checks for function instance availability. The controller checks the worker pool for any existing, idle instance associated with $F_i$ and if the found instance has an available concurrency slot. This is known as the warm execution path. If no warm instance is available, the system initiates a cold start, transitioning through three key phases of placement, code download and instance initialisation. The time spent by a request waiting for controller during scheduling decisions is referred to as $T_{wait}$. To provision the new function instance, the controller selects a worker node, $W_k$ such that the condition $Used(W_k) + mem_i \leq Capacity(W_k)$ is satisfied where the newly allocated function resources must not exceed the total capacity of the selected worker node. Thus, the duration taken for this initialisation can be defined as $T_{cold}$. Once a function instance is either found or initialised, the function logic is executed and the response is returned to the caller. Additionally, the platform metering services record the invocation execution time ($T_{exec}$) and memory usage ($mem_{exec}$), typically rounded to nearest millisecond, for billing purposes. Therefore, the total latency $L$ experienced by a request can be defined as $L = T_{wait} + T_{cold} + T_{exec}$. Immediately after the execution, generally, a function instance remains active and idle for non-deterministic period, known as a keep-alive

timeout ($T_{keep}$). Consequently, if no new events are received within this timeout, the instance resources are reclaimed by the provider to free up the worker node resources.

This highly automated execution cycle fundamentally redefines the shared responsibility model for both the developer and the service provider. From the developer's perspective, the primary goal is maximising business value by focusing solely on application logic, reducing wasted effort on infrastructure planning and management. Their workflow is reduced to three core steps of writing the function code, uploading the code to the platform through exposed interfaces, and assigning the initial resource configuration (such as memory or maximum execution time). Developers must specify these resources, as CSPs typically allow configuring function memory while proportionally allocating other compute resources like CPU and network bandwidth [42] or by choosing from pre-defined resource combinations [43]. In contrast, the CSP manages the complete execution lifecycle automatically, from dynamic scheduling and elastic allocation to running the code and eventually de-allocating and recycling resources when the function becomes inactive. This idea of shared responsibility enables service providers to maximise their hardware utilisation through continuous resource pooling while charging customers based only on resource consumption. These combined properties of statelessness, ephemeral execution, fine-grained resource metering, and hyper-elasticity constitute the core design principles of the FaaS model.

Major CSPs have established mature FaaS platforms, including Azure Functions [7], Google Cloud Run Functions [8], and IBM Code Engine, similar to AWS Lambda. Additionally, several open-source serverless frameworks, such as OpenFaaS [9], Knative [10], and Apache OpenWhisk [11], have also been introduced to offer function-based abstraction. These platforms have evolved considerably and are currently transitioning toward the vision of Serverless 2.0, which significantly widens the scope of FaaS capabilities. This evolution includes supporting longer execution times, native state management, providing a rich set of integrated managed services, and enhanced monitoring and observability features. According to an industry report [3], FaaS currently dominates the serverless computing market with over 65% share, evidenced by over 1.5 million users invoking functions on public CSP platforms. Consequently, FaaS has rapidly expanded beyond traditional use cases such as APIs, document processing, and handling high-

traffic HTTP endpoints to encompass modern, compute-intensive applications, including AI/ML inference, video encoding/decoding, and serving Model Context Protocol (MCP) servers. While the core function-based abstraction is similar among serverless platforms, they vary in their flexibility of resource allocation, billing granularity, and support for different function runtimes and BaaS services. Figure 2.1 depicts the FaaS workflow from a developer, user and service provider perspective. A developer uploads the function code along with the function configuration to execute, a user may interact with an event source to trigger these functions, and the service provider manages the platform and controls resource-related decisions. Hence, some of the identified key characteristics [2][53] of the FaaS execution model are discussed below:

**Event-driven** FaaS is an event-driven offering where functions are invoked by various event sources such as HTTP requests, database changes, timers or orchestration events. The platform responds to these events in real-time by creating on-demand function instances.

**Hostless and elastic** The FaaS execution model relieves users of complex resource management activities like server allocation, OS update and patching and reduces operational burden. Additionally, users are unaware of where the function is hosted, i.e., hostless, which allows for more flexible execution and pay-per-use pricing by the service provider. To support elasticity, FaaS offers seamless scalability where resources scale based on demand and are freed if there are no invocations.

**Statelessness** A serverless function executes within an isolated and temporary environment (container or a micro-VM) and does not maintain an in-memory state between invocations. This statelessness makes the function scale-out easier as the platform could instantiate parallel instances for concurrent requests without managing or transferring the function state between them.

**Short but variable execution time** A typical serverless function lasts for a few milliseconds to seconds [19] and is effectively billed in units of milliseconds for actual running time. However, the cold start time can significantly impact the perceived latency

or performance of the functions that execute for very short durations.

**Memory and CPU Allocation** Functions are often allocated operational resources and many FaaS platforms allow developers to specify the amount of allocated memory. Generally, runtime memory is a critical configuration that is exposed by the providers like AWS, Azure and Google where a function is allocated other resources like CPU and network bandwidth in proportion to memory. For example, AWS allocates compute power proportional to memory where 1 vCPU is allocated at 1769 MB of function memory. A function with more memory typically means more CPU power, leading to faster execution for compute-intensive applications. This also means that the function can process larger amounts of data, perform complex operations at speed and reduce overall latency. However, larger memory allocations also affect the operational cost, as the billing is typically done based on the running time and the allocated memory.

**Execution timeout** FaaS providers typically limit the execution time of a function before it is forcibly terminated by the platform. Developers must configure their function with a long enough function timeout to finish its task. For tasks that execute for a few milliseconds or seconds, setting a short timeout is suitable, whereas a longer running task such as multimedia processing [37], a longer timeout is intended. Providers such as AWS allow timeouts up to 15 minutes, whereas Azure Functions may provide longer timeouts, i.e., more than 30 minutes, based on the subscribed plan. The timeout is also significant as it prevents a function from getting stuck in an infinite loop and saves developers from exorbitant costs. This way, a function is automatically stopped after exceeding the timeout.

**Ephemeral Storage** In addition to the compute resources, a function also has a temporary configurable storage attached to it. This ephemeral storage is only available during the life of a function instance and is useful in applications that require a file system during execution for processing or intermediate data storage, like file download.

**Concurrency** Function concurrency refers to the number of simultaneous incoming requests a function can handle. FaaS platforms typically react to the incoming demand

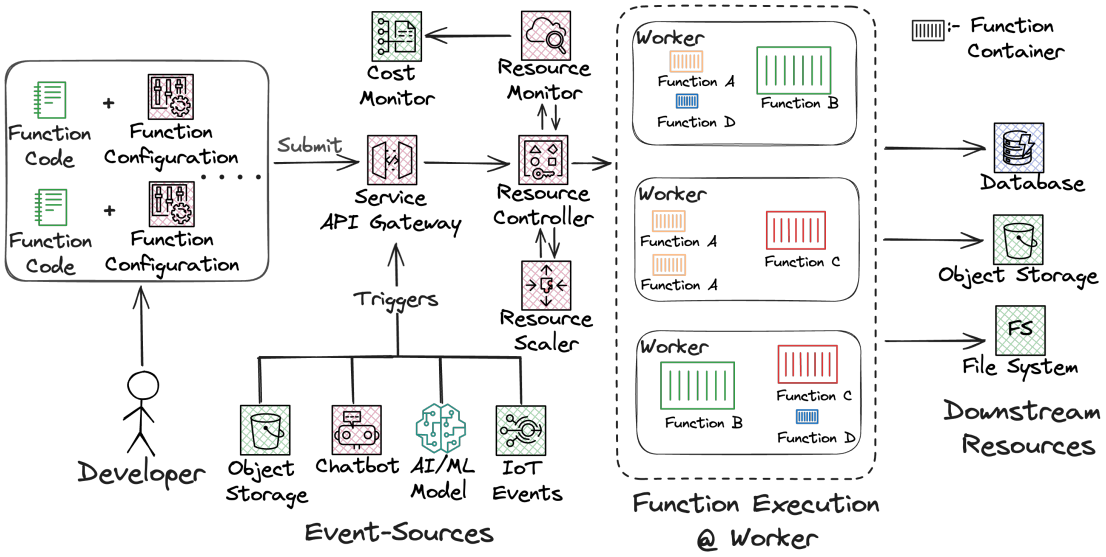



**Figure 2.1:** FaaS Workflow - Developer, User & Service Provider Perspective

and may start multiple function instances concurrently. However, to reduce the effect of cold start and improve function performance, developers can also configure provisioned concurrency [6] or minimum function instances [8] that keep a certain number of function instances warm and ready to execute. This is critical for latency-sensitive applications, but it does incur some extra costs for keeping the resources warm and idle.

### 2.2.3 Function-as-a-Service Platforms and Frameworks

This section introduces the main characteristics and offerings of some of the existing commercial FaaS platforms and open-source serverless frameworks. A summary of the discussed platforms is presented in Table 2.1.

**AWS Lambda** Lambda [6] was first introduced in 2014 as an event-driven compute service that supports various language runtimes such as Python, Node.js and Rust. While it integrates with various AWS and external services and event triggers like Amazon S3 [51] and Apache Kafka [54], it offers an important aspect of configurable resources to developers. Lambda is subject to certain quotas and limits that developers must explore to optimise the operational cost and performance of a function. The most important consideration is a function's memory allocation, which developers can con-

figure between 128 MB and 10,240 MB in 1 MB increments, proportionally allocating other resources like CPU power. For example, 1769 MB of memory allocates an equivalent of 1 vCPU to the function. While increasing the memory gives more CPU share and speeds up a function execution for a compute-intensive task, it may lead to increased operational cost per invocation. A Lambda function is allowed to run for a maximum duration of 15 minutes, and developers must set an appropriate *timeout* to prevent any unexpected costs. Additionally, functions may be configured with ephemeral storage of up to 10,240 MB for tasks that may require intermediate file download or processing. AWS also puts a limit on the size of data passed to a function, i.e., the *payload*, specifically, 6 MB for synchronous request-response, 200 MB for streamed response, and 256 KB for asynchronous invocations [42]. As these functions scale in response to the demand, Lambda functions can scale up to 1000 unique execution environments every 10 seconds to handle the concurrent invocations. Apart from the configurable settings, Lambda does offer generous free-tier usage limits [55], however, understanding the effects of resource configuration is crucial for developing cost-effective and performant functions. Lambda further integrates with services like AWS Step Functions [56] to support workflow application scenarios, where respective solution performance overheads and pricing are applicable in addition to Lambda invocation.

**Microsoft Azure Functions** Microsoft announced the availability of its Azure Functions in November 2016, where an application-first approach is taken for development to represent a group of functions as an application. Azure Functions supports multiple language runtimes, including C#, Java, PowerShell and Python [7] while integrating with other Microsoft and external services event triggers, including Azure Cosmos DB events and message queues. Unlike other FaaS platforms, a defining characteristic of Azure Functions is its flexible hosting and resource configuration schemes [57] that give developers control over the application performance and cost. It offers different hosting options, such as premium, consumption or dedicated, which define how and what type of resources are supported, configured and billed. The most common option is the consumption plan [57] that offers a pay-as-you-go resource consumption and billing scheme where the underlying host will be allocated up to 1.5 GB of memory with a variable CPU share. For more complex workloads, schemes like the *flex consumption plan* are also of-

fered, where instances with memory allocation up to 4 GB and proportional CPU share can be selected. While configuring more memory gives more processing power, Azure recommends using a 2048 MB memory instance out of the available options for most scenarios. Azure functions allow a maximum configurable timeout of up to 10 minutes (default 5 minutes) in standard consumption plan. Contrastingly, in a flex consumption plan, it provides a more flexible timeout setting, enabling support for longer-running functions, although practical limits of 30 minutes may apply based on the environment and usage patterns. Azure further offers both ephemeral and persistent storage as part of its hosting plans, where temporary storage could range from 0.5 GB to 140 GB per instance and persistent storage can be configured between 1 GB and 1000 GB per hosting plan. In addition to this, functions could scale to as many as 1000 instances in a premium plan and as low as 30 instances in a dedicated plan, with a maximum recommended request *payload* size of 100 MB. Azure further allows the development of stateful applications and workflows in addition to simple stateless functions via Durable Functions [58]. These features, combined with various resource configuration options, allow developers to explore settings and fine-tune the application performance.

**Cloud Run Functions** Rebranded in 2024 from *Cloud Functions Gen 2* [8], *Cloud Run Functions* unify previous function deployments (Cloud Functions Gen 1 and Gen 2) under Cloud Run's fully managed container platform [8]. Cloud Run Functions support several programming languages, including Python, Java, Go, Node.js, .NET, PHP, and Ruby, where the runtime base image and container lifecycle are fully managed by Google. They support various *triggers* such as HTTP requests, Pub/Sub messages, Cloud Storage changes, and Firestore database events via Eventarc integration. Memory allocation is configurable up to 32 GiB with flexible CPU allocation proportional to memory, ranging from fractional vCPUs during idle phases to up to 8 vCPUs during active request processing [59]. By default, CPU is only allocated while processing requests and during container startup and shutdown. However, users can configure an always-allocated CPU that enables containers to run background or asynchronous tasks. Request timeouts default to 5 minutes but can be extended up to 60 minutes, accommodating long-running workloads. Cloud Run Functions can scale up to 2000 instances per service and support up to a maximum of 1000 concurrent request per container

instance, improving resource efficiency and reducing cold start latency. These functions seamlessly integrate with the wider Google Cloud ecosystem and support complex workflows through the Workflows service [60], combining the scale and flexibility of container-based deployment with the simplicity of function-as-a-service.

**Table 2.1:** Comparison of FaaS Platforms and Frameworks

| Platform or Framework | Memory (Max) | Timeout | Concurrency or Scaling | Storage | Language Runtimes | Distinctive Features | Ecosystem/Lock-in |
|---|---|---|---|---|---|---|---|
| AWS Lambda | 10,240 MB | 15 min | 1,000 envs/10 sec | 10,240 MB (temp) | Python, Node.js, Rust | Market leader, granular resource config, workflows via Step Functions, free-tier generous | Tight AWS integration; vendor lock-in |
| Azure Functions | 4,096 MB | 10-30 min/no limit (plan) | 100–1,000 instances | 140 GB temp, 1 TB persistent | C#, Java, Python, PowerShell | Application-first, multiple hosting plans, premium scaling, Durable Functions support | Tight Azure integration; vendor lock-in |
| Cloud Run Functions (GCP) | 32 GiB | 5–60 min | 1,000 req/instance | Ephemeral (tmpfs) | Python, Java, Go, Node.js, PHP, Ruby, .NET | Container-native, flexible vCPU/mem, high concurrency, workflow with Workflows, broad triggers | Tight GCP integration; vendor lock-in |
| Knative | Kubernetes config | Configurable | Scale-to-zero, custom | Kubernetes config | Python, Go, Rust, TypeScript | Autoscaling, concurrency config, integrates monitoring/logging, runs on any K8s | Open-source, no vendor lock-in |
| OpenFaaS | Kubernetes config | Configurable | 1–5 (community); scale-to-zero (Pro) | Kubernetes config | Python, Ruby, Go, Java | Simple deployment, flexible triggers, autoscaling policies, OCI image support | Open-source, no vendor lock-in |
| Apache OpenWhisk | 512 MB | 5 min | 120 req/min (default) | N/A | Java, Python, .NET, PHP, Swift | Docker-based, event-rich, open-source, configurable concurrency | Open-source, no vendor lock-in |

**Knative** It is a Google-sponsored Kubernetes-native platform that supports serverless workloads. Knative [10] leverages the underlying Kubernetes [61] primitives of container orchestration, scaling, scheduling and configuration to provide tools that automate the task of continuous integration and continuous delivery (CI/CD).Knative deploys functions as Kubernetes *pods*, supporting language runtimes such as Python, Rust, Go and TypeScript. Functions can be invoked via *triggers* such as HTTP events and events that conform to CloudEvents. Knative offers a scale-to-zero capability that supports concurrency and request-per-second metrics for autoscaling. The platform also

provides fine-grained control to allocate resources like memory and CPU to functions. Kubernetes primitives of resource requests and limits can be leveraged for specifying these values. Another key configuration aspect is concurrency, which defines the number of concurrent requests a single service instance can handle. By default, Knative often sets a concurrency limit of 100 concurrent requests per instance, but this can be adjusted. Setting a higher concurrency can be cost-effective as it reduces the number of running instances, but it requires careful tuning of CPU and memory to ensure each instance can handle the increased load without performance degradation. Additionally, it integrates well with different monitoring and logging solutions like Prometheus [62] and Grafana [63], FluentBit [64] or ElasticSearch [65] to collect several metrics from platform components.

**OpenFaaS** OpenFaaS [9] is an open-source Kubernetes-native serverless platform that simplifies function development, deployment and management. Functions can be deployed as Open Container Initiative (OCI) images with toolkits like Docker and developed in runtimes such as Python, Ruby, Go and Java. OpenFaaS offers subscription-based access like community, pro and enterprise editions with different levels of service offerings. It supports a number of function triggers such as HTTP requests, Cron jobs, AWS events and PostgreSQL [66] database events. Additionally, functions can be configured with three autoscaling types of request-per-second (RPS) scaling, capacity scaling (number of in-flight requests) and CPU utilisation-based scaling, however, within a range of 1 to 5 instances in community edition, and a scale-to-zero provision is available with the Pro edition to release resources after an idle duration of 15 minutes [9]. The CPU and memory resource requirements and limits can be configured for a function as a Kubernetes primitive and limit the concurrent requests executing inside a container or the timeout by setting an environment variable. Furthermore, OpenFaaS can be set up on different Kubernetes-based environments such as managed Kubernetes services by AWS [67], Azure [68] and Google [69].

**Apache OpenWhisk** IBM introduced OpenWhisk [11] as an *open*-source serverless platform in February 2016 with an idea to quickly run users' code and *whisk* or release its resources. It runs functions in response to different events from various sources such as mobile and web applications, databases, scheduled jobs and sensors. Its architec-

ture is powered by multiple open-source technologies such as Docker, Apache Kafka, CouchDB and Nginx engine. The OpenWhisk programming model supports code written in Java, Python, .NET, PHP and Swift while also extending its model to non *out-of-the-box* runtime. Functions are automatically scaled and can be configured with a number of options to limit the system usage [70]. A function can execute up to 5 minutes per invocation with a concurrent invocation rate of 120 requests per minute. Functions can be allocated memory between 128 MB and 512 MB, have a maximum payload size of 1 MB, a maximum code size of 48 MB, and can be configured with a per-action maximum function concurrency.

## 2.3 Taxonomy of Function Resource Management

This section outlines the identified taxonomy of the related literature for function resource management. The high-level categorisation comprises key resource management aspects - function cold start, autoscaling, configuration, and scheduling. Moreover, factors affecting these resource management decisions, such as key performance indicators, optimisation goals, workload characteristics, and deployment environments, are also introduced. The proposed taxonomy is highlighted in Figure 2.2 and a more granular discussion is presented in the following sections.

### 2.3.1 Function Resource Management (FRM) Sub-Aspects

This section presents the taxonomy on function resource management, derived to categorise the existing literature based on their operational domains and the type of resource management policy adopted. Resource management in FaaS encompasses several core problems concerned with the function's operational lifecycle and the persistent technical challenges of the execution model. These domains are used here to classify research into studies that target cold start mitigation, reactive autoscaling, static resource configuration, and function scheduling decisions. Moreover, these aspects are intrinsically interdependent, for example, resource configuration directly impacts cold start duration, which in turn influences the effectiveness of reactive autoscaling policies. Consequently,

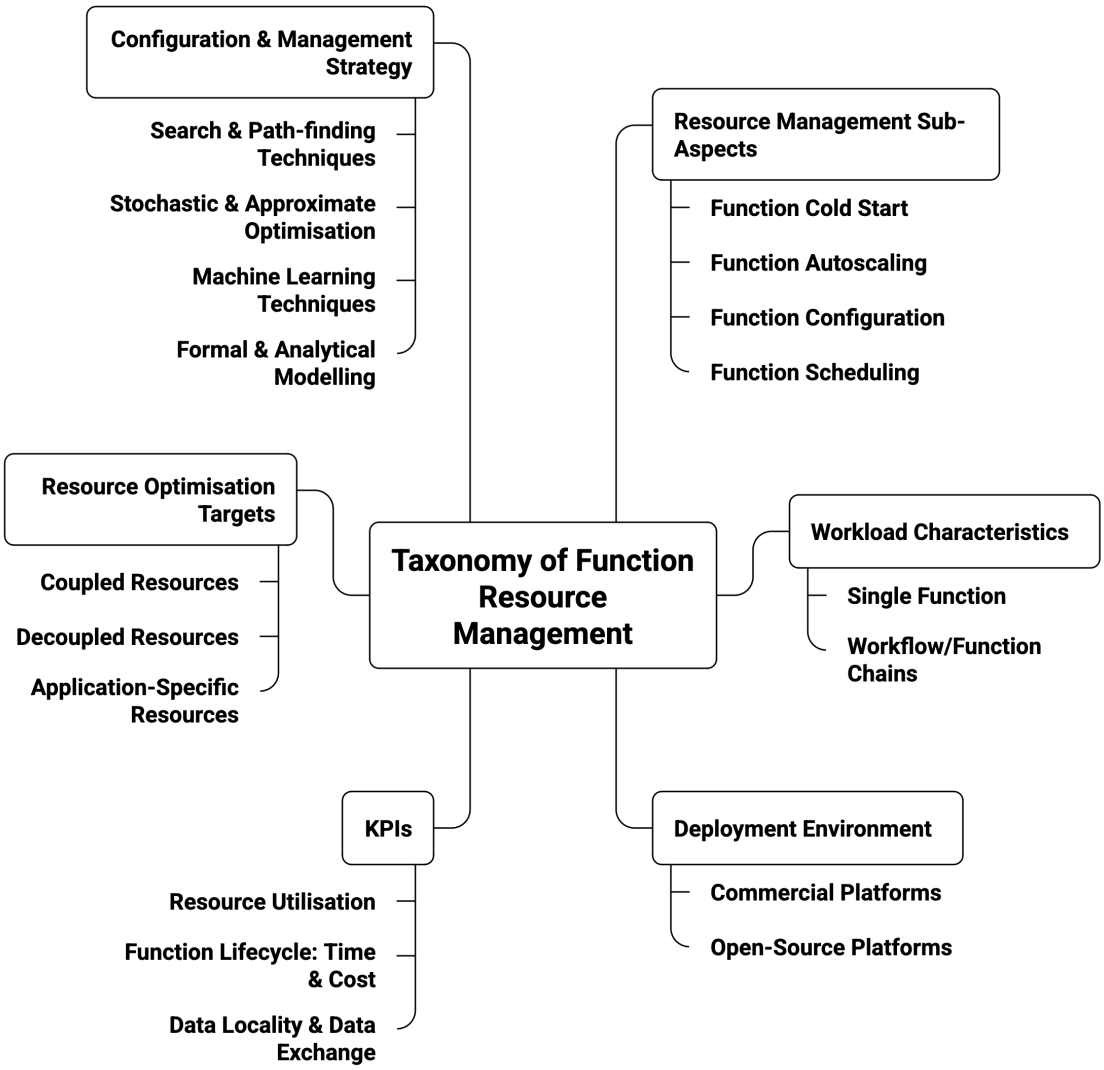



**Figure 2.2:** Taxonomy of Function Resource Management

optimal function management requires a holistic approach, as a limited number of existing works address complexities that span multiple categories, seeking to solve the overall multi-objective problem rather than isolated issues.

### Function Cold Start

Cold start latency in FaaS is a well-recognised challenge. Research efforts to mitigate cold starts are broadly divided into two strategic approaches, environmental optimisation and cold start frequency control [12]. Environmental optimisation focuses on minimising the duration of the cold start itself by accelerating container preparation and reducing delays in loading function runtime dependencies. Techniques under this category include checkpoint/restore (C/R) mechanisms, micro-VM technologies such as Firecracker [71] that offer lightweight isolation, and optimised network setup strategies, as well as specialised containers or sandboxes like SOCK and Catalyzer [28]. The second approach targets minimising the frequency of cold starts by proactively preventing function instances from being evicted or maintaining a set of idle function instances. This is achieved through periodic function invocations (pinging), warm-up strategies to keep the function instances in-memory, intelligent caching to store or reuse the data or database connections, and prediction-based autoscaling methods utilising models such as Seasonal Auto-Regressive Integrated Moving Average (SARIMA) or RL techniques to anticipate demand and retain warm function instances [12][72][73].

### Function AutoScaling

This aspect of FRM focuses on managing the elastic capacity of the function fleet, the process of adding or removing instances in response to dynamic workload changes. The central goal is to overcome the limitations of reactive scaling policies and improve resource utilisation and system throughput. These limitations arise from the inherent lag between workload changes and resource response, coupled with the black-box nature of the cloud environment. Additionally, the reactive nature of these scaling policies exacerbates the cold start problem because it necessitates provisioning a new execution environment precisely when demand is peaking and resources are needed most. Research in this domain focuses on stochastic and approximate optimisation strategies, including advanced control theory (e.g., proportional integral derivative (PID) controllers) and, increasingly, ML strategies like recurrent RL. These works frequently analyse the workload to model demand dependencies across the system. The optimisation target is typ-

ically multi-faceted, seeking to maximise system throughput while respecting resource and budget constraints, thereby requiring adaptive solutions that move beyond simple threshold-based heuristics. Traditional rule- and threshold-based autoscaling tend to be insufficient for complex, dynamic environments, leading to a surge in research exploring stochastic optimisation techniques and machine learning, particularly RL, to address the multifaceted autoscaling challenge [32][72]. RL-based autoscalers leverage data-driven models such as Q-learning [73], Deep Q-Learning [74], and recurrent architectures like Long-Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) to predict workload trends and proactively determine optimal scaling actions, outperforming static heuristics by adapting to temporal workload dependencies and multi-resource constraints [75][76]. Empirical investigations [77] have established that autoscaling policies must be sensitive to multi-dimensional metrics such as concurrency, resource utilisation, and QoS to effectively manage the function execution. Such advanced strategies are necessary to dynamically balance trade-offs across horizontal scaling (adding instances), vertical scaling (increasing resources per instance), brownout adaptations (reducing quality under load), and resource allocation decisions. By addressing these complex dimensions, these methods aim to maintain SLA compliance with minimal cost overheads [78][79].

Many of the RL-based and predictive techniques explored in the cold start context [72][73][80] are similarly applicable to autoscaling. For example, predictive models based on LSTM architectures that forecast function invocation patterns to keep instances warm for cold start mitigation naturally enable informed scaling decisions by anticipating workload changes. The use of function resource and performance metrics for RL agents to determine instance counts or concurrency levels applies to both cold start reduction and autoscaling for throughput optimisation. Moreover, strategies like container pool maintenance and proactive resource retention policies, designed to manage function state and reduce cold starts, simultaneously acts as proactive autoscaling [25][28][81]. This overlap reflects the inter-dependency of cold starts and proactive autoscaling [27] in FaaS platforms, where explored solutions often simultaneously address multiple system level objective like throughput and resource utilisation.

### Function Configuration

This aspect addresses the static function configuration challenge, where developers must manually set execution resources and other tunable parameters (such as memory, timeout, etc.), leading to a fundamental trade-off between operational cost and function performance. Research in this area concentrates on automating the decision-making process for granular resource configuration, which has been repeatedly established to have a significant influence not only on the function performance and operational cost, but also the provider's ability to schedule them [2].

The initial approaches in the literature utilised formal and analytical modelling [82] to quantify the non-linear relationship between coupled resource allocation (mostly memory) and execution time. With the growing complexity of serverless workloads and their dynamic resource requirements, the field has progressed toward ML-based profiling techniques [83][84] designed to predict and optimally configure function resources. These studies aim to learn policies that balance multi-objective trade-offs across execution latency and cost efficiency while satisfying SLAs, enabling proactive and fine-grained resource tuning. A complementary research direction explores decoupled function resources where different compute resources can be set independently, unlike proportional to memory allocation. These adaptive frameworks leverage learning-based techniques and probabilistic models to optimise resource utilisation across diverse serverless use-cases [85][86]. Whether focusing on single functions or complex workflows, these methods ultimately seek to automate resource tuning with minimal human intervention, mitigating the risks of over-provisioning and under-provisioning.

### Function Scheduling

The scheduling aspect investigates the challenges of allocating and orchestrating function instances across serverless clusters to optimise resource utilisation, minimise fragmentation, and improve performance indicators such as function latency. Beyond individual function management, this domain addresses strategic placement and routing of computational workloads, deciding not just where a function should run, but when and how requests are orchestrated to existing or new instances [87]. By ensuring a balanced

load distribution and minimising inter-function interference, intelligent scheduling algorithms help avoid resource contention and under-utilisation within shared underlying infrastructure. Some of the key performance objectives that the existing literature addresses include efficient use of warm instances, redirection/reordering/batching of requests to appropriate instances and a careful balance of instantiating new instances with function concurrency.

Moreover, to overcome the inefficiencies of static placement decisions leading to poor resource utilisation, various provider and developer-focused frameworks, including workload-aware, data-locality driven and deadline sensitive scheduling, have also been explored, especially for complex workflows with inter-dependent functions or Directed Acyclic Graph (DAG)s [88][89]. These efforts underpin much of the recent progress in both commercial and open-source deployment environments, supporting scalable serverless computing by harmonising resource utilisation, data access efficiency, and cross-function orchestration [90].

### 2.3.2 Workload Characteristics

This section categorises the existing literature under the scope of workload characteristics that have a direct impact on function resource management strategy and configuration. The identified elements of the workload model form the basis of different management approaches, selection of optimisation goals and key performance indicators recognised in the surveyed literature.

#### Workload Model

Workload model refers to the composition of serverless applications using one or more functions as components with their respective resource requirements. FaaS has been proven suitable for a diverse range of use-cases; however, the application's expected QoS and the number of functions in an application, also known as function cardinality, are mutually dependent. While a single function's performance metrics including the cold start latency, execution time and cost are tied to its allocated resources, the overall application performance is highly susceptible to cascading performance degradation.

This means the individual performance of functions creates a ripple effect in overall application performance, significantly increasing end-to-end latency and execution cost, particularly in complex function chains or workflows. Additionally, a provider's decision to allocate requested resources and optimally place these functions to minimise the communication and data exchange overheads, are also influenced by this cardinality. Critically, as the execution flows through the application, the overall end-to-end performance and cost fluctuate based on the configuration of every connected function. Furthermore, as the function cardinality grows, an application's performance-cost trade-off becomes non-trivial and the influence of overheads (such as cold starts or under-provisioning) has a cascading or ripple effect on downstream function execution and overall latency. Therefore, considering an application workload model is a fundamental factor for tuning and managing function resources.

**Single-function application:** This category exclusively deals with single function applications where it is a fundamental unit of deployment and the lowest level of available customisation interface. A large body of existing research deals with individual functions that perform a dedicated task. The scope of resource management encompasses scheduling, configuration and scaling, all examined specifically at the individual function level. By focusing on standalone function optimisations while targeting different performance trade-offs, this discussion deliberately excludes considerations related to multi-function workflows, dependencies, or orchestration complexities, providing a comprehensive yet targeted analysis of single-function performance and resource management.

**Workflow/Function Chain:** A function chain or a workflow is a method of developing applications with multiple, inter-connected functions that together execute a larger and complex business logic. These functions are executed step by step and are connected by dependencies, forming a DAG. Therefore, the output of one step becomes the input of another, enabling composition of complex, multi-step applications [91][23]. These function chains are critical for scenarios where a single, short-lived function can not handle a complex task such as data processing and ML workflows. Generally, orchestration

tools such as AWS Step Functions [56] and Azure Durable function orchestrations [58] are leveraged to build, manage, and execute workflows. Each function or step in the workflow requires individual settings such as timeout, concurrency and resource configurations, where the overall performance and execution of the workflow are governed by these individual units. Consequently, when functions in a chain are executed, the configuration of downstream services is also vital to avoid overwhelming them or exceeding any platform-specific limits. While individual function executions are billed per execution, the chains are typically charged based on the number of state transitions (transition to different steps of the workflow). Therefore, efficiently designing the workflows with minimal steps and executions can result in reduced overall operational costs.

### 2.3.3 Deployment Environment

This section categorises the research based on the type of serverless environment or platform used for its experimental setup and evaluation. In addition to commercially available FaaS platforms that expose limited configurable parameters and manage resources in a black-box manner, a number of open-source frameworks are also available that provide the necessary transparency required for developers to examine the internal mechanism of a FaaS platform. Therefore, a discussion of these deployment environments is important to recognise and appreciate the feasibility of numerous research works in serverless systems.

#### Commercial Serverless Platforms

Commercial FaaS platforms such as AWS Lambda [6] and Microsoft Azure [7] are widely utilised for conducting research owing to their increased adoption and real-world relevance. These platforms, however, offer limited architectural visibility and configurable function parameters due to their proprietary nature, forcing developers and researchers to infer their system behaviour through experimentation. Consequently, a number of research works aim to address these limitations by focusing on distinct function configuration and optimisation opportunities for performance enhancement and operational cost savings from a developer perspective.

#### Open-Source Serverless Frameworks

In addition to large-scale commercial serverless services, many open-source serverless frameworks have also been introduced, such as OpenFaaS, Apache OpenWhisk, and Knative. Unlike commercial offerings, where function resources such as CPU and network bandwidth are allocated either proportional to memory or are selected from a pre-set list of configurations, open-source frameworks offer greater flexibility in configuring function resources. Additionally, these frameworks allow developers to test and tune different resource allocation and placement strategies that fit specific application domains, especially considering the agnostic or opaque nature of commercial offerings. Therefore, these platforms have also been widely used for recording FaaS experiments and exploring different research opportunities.

### 2.3.4 Key Performance Indicators

In this section, we categorise the existing literature under the key objectives that drive research on function management and configuration in FaaS. These objectives are related to specific performance metrics that define the success level or optimisation constraints targeted by individual studies. A function's configuration settings, such as memory and CPU allocation, are critical as they directly govern resource management decisions and their operation. These decisions intrinsically influence the key indicators that reflect the QoS experienced by the user. Existing studies emphasise optimising performance metrics like throughput, execution time and latency, reduce operational costs, and improve resource utilisation, often alongside data locality considerations to minimise communication overhead.

#### Resource Utilisation

The vast search space of possible resource configurations and their impact on function performance, often makes function configuration an ad-hoc decision-making process. This can lead to resource under-utilisation at both the function level and the underlying infrastructure, primarily as a result of resource over-provisioning or misallocation

of resources [92]. Resource utilisation refers to the actual usage of allocated resources by a function through its execution lifecycle, encompassing dynamic factors like runtime scaling and scheduling decisions. This is critical for developers as the interplay between function performance, cold starts and function runtime cost is tied to how resources are managed and configured. From a CSP perspective, improving resource utilisation is equally important to maximise infrastructure efficiency. This includes optimising function placement, scheduling to avoid contentions, scaling based on demand and keeping the function instances active and available, ultimately reducing resource wastage while maintaining SLO and operational efficiency.

**Function Lifecycle: Time and Cost**

FaaS functions ideally execute for a shorter duration, ranging from milliseconds to a few seconds, while the application expects a near real-time response. Existing research [93] consistently highlights that resource allocation significantly impacts the execution time of a function, eventually affecting the associated runtime costs. In particular, functions are found to speed up their execution with more resources, where serverless platforms usually tie function compute resources with memory allocation [20]. Thus, optimising this non-trivial configuration space involves managing complex trade-offs across cost, performance, and QoS guarantees. In addition to this, scheduling decisions, cold start latencies, and dynamic scaling behaviour, critically affect the function lifecycle time and associated costs. Cold starts can introduce significant latency overheads, particularly for bursty traffic or infrequent invocation patterns, necessitating mitigation strategies. Furthermore, data transfer latencies and inter-function communication costs within workflows occasionally impact individual function performance. Therefore, balancing these factors creates a multi-dimensional optimisation problem central to function resource management. However, the terms response time [88][84], latency [94][95], job completion time [96] and execution time [18][82] have been used interchangeably in the literature and we generalise them under *function lifecycle time* and associated cost as *function lifecycle cost*.

#### Data Locality and Data Exchange

Recently, FaaS has been leveraged for a variety of application domains such as video processing [95][86], big data analytics [96][97]and ML training-inference [98]. These applications generate large quantities of intermediate data, which is then processed in downstream stages. In the FaaS context, this requires a function-to-function exchange of this intermediate data, either wholly or partially. Existing research has identified that direct communication of functions is difficult [91] and usually leverage an external remote storage [82] to promote the data or information exchange amongst functions. Therefore, several studies have identified data locality and exchange as a crucial operational constraint, and few have dedicated their efforts to understanding the impact of function configuration, along with its impact on function placement and application performance.

### 2.3.5 Resource Configuration Model

A serverless function's performance, runtime cost and CSP's efficiency to schedule it, are directly influenced by its resource configuration. This section categorises the existing literature based on the resource configuration model that the studies assume while optimising for different objectives and addressing resource management sub-aspects.

#### Coupled Resource Configuration

In a coupled resource configuration model, function resources such as CPU and network bandwidth are typically allocated in proportion to the amount of configured memory. Therefore, allocation of more memory to the function instance enables allocation of more CPU to it and vice-versa. The current serverless operational model of proprietary solutions such as AWS Lambda [42], Azure Functions [7] and Cloud Run Functions [59] either offers memory as a single configurable knob or provides a fixed set of proportional resources to fine-tune function performance. As discussed earlier in subsection 2.3.4, the amount of configured resources has a direct impact on different function performance indicators. Additionally, this amount of configured resource plays an important role for

the selection of appropriate infrastructure to execute the functions. The provider's black-box resource management tasks such as resource allocation, provisioning and function placement are directly impacted by the tight-coupling between different resources [20]. Thus, existing works leverage this relationship to explore different optimisation techniques.

#### Decoupled Resource Configuration

The result of tightly-coupled and memory-proportional resource allocation leads to over-provisioning and resource wastage in functions requiring additional CPU shares such as a compute-intensive function, in order to execute successfully. This over-provisioning often leads to reduced overall resource utilisation where only one type of resource is heavily used. To address this, open-source serverless frameworks such as OpenFaaS [9], Knative [10] and OpenWhisk [11] provide decoupled resource allocation where the amount of allocated resources, such as memory and CPU, is defined independently which allows flexible and refined function configuration. Moreover, existing research [34][99] have criticised the constraints of coupled function resource allocation and have leveraged the decoupling to optimise function performance and runtime cost in addition to the exploring various function scheduling policies such as dependency-aware function placement.

#### Other Specific Resources

In addition to the compute resources like memory and CPU, the performance of applications such as video processing and analysis, and ML training and inference tasks is also influenced by use-case specific parameters. For instance, the latency and resource efficiency of a video processing pipeline [86][95] are affected by parameters such as sampling rate, number of frames or batch size. On the other hand, the function execution time, resource utilisation and the overall function costs for Artificial Intelligence (AI)/ML training can be negatively impacted by the amount of resources allocated across different training stages of hyperparameter tuning [99]. Moreover, big data workloads [96] maybe optimised for performance by deciding the appropriate number

of workflow stages and its number of functions involved. Therefore, a number of existing works can be categorised based on these alternative resource allocation and function configuration parameters.

### 2.3.6 Configuration and Management Strategy

To address the multifaceted resource management and configuration challenges inherent in FaaS, the existing literature has explored and proposed a wide array of specialised approaches, techniques, and frameworks. This section categories these individual solutions into distinct algorithmic domains based on the underlying strategy used for optimisation. We identify the following broad categories of research methodologies: Search and Path-Finding Techniques, Stochastic and Approximate Optimisation, ML-Based Techniques, and Formal and Analytical Modelling.

#### Search and Path-finding Techniques

In FaaS deployments, service providers usually require developers to specify the minimal function resources required for a successful execution. In practice, this configuration space directly encodes key resource management decisions such as CPU and memory allocation, concurrency limits, and scaling thresholds, which influence both performance and infrastructure efficiency at the platform level. Typically, developers profile their functions at multiple configurations for computational efficiency and decide on the optimal one based on the desired time and cost constraints. However, navigating a vast configuration space and selecting the best setting is challenging due to inherent performance variations, profiling time and the complex interactions between function-level choices and system-wide resource management policies such as scheduling, provisioning, and auto-scaling.

#### Stochastic and Approximate Optimisation

A serverless environment, with its reduced control over low-level infrastructure but different function-level knobs such as memory size and execution timeouts, has been a

choice for bursty, embarrassingly parallel, and highly dynamic workloads. This dynamicity and unpredictability, along with performance and cost variation, present a unique function resource management and configuration challenge where each setting simultaneously affects both application-level QoS and platform-wide resource efficiency. While deterministic search methods are simple and attempt to explore a large search space, they struggle to cope with the workload dynamicity, varying resource requirements, and cost-performance trade-offs that arise when right-sizing individual functions and coordinating them at scale. As a result, stochastic and approximation techniques have emerged as effective alternatives to fine-tune the function resource configuration by leveraging heuristics, meta-heuristics, probabilistic models, and adaptive learning mechanisms. These techniques enable efficient search and optimisation of function configuration while balancing fluctuating performance, cost and utilisation in highly dynamic serverless environments.

#### Machine Learning Techniques

As serverless environments grow in popularity, adoption, and complexity, traditional methods such as search algorithms or optimisation techniques, may lack necessary adaptability to handle the dynamism and unpredictability of workloads and function execution patterns. In comparison to heuristics or approximation methods, ML methods can learn function-specific execution patterns and apply a generalised model to unseen workloads, enabling efficient real-time function resource management decisions. Therefore, ML-based techniques furnish an alternative approach to predict, optimise and dynamically manage and configure function resources by utilising historical and runtime data including RL. As a result, the integration of ML methods allow serverless platforms to advance towards enhanced system autonomy by intelligently allocate and adaptively manage function resources. This further reduces the developer effort and intervention in the configuration process while improving runtime cost, performance and utilisation across dynamic workloads.

#### Formal and Analytical Modelling

While ML approaches, and stochastic and approximation techniques provide a flexible and adaptive way to manage and configure function resources, there are other proposals that require explicit modelling based on application structure or workload characteristics. These methods do not fall under the previously defined categories and generally leverage domain knowledge and analytical or mathematical models to capture the relationship between function resources and their performance.

## 2.4 Classification of Function Resource Management Techniques using Taxonomy

In this section, we review existing key works on function resource management that identify most with the proposed taxonomy. In essence, we present here the works that explore novel techniques for one or more of the primary aspects of resource management that we have identified. However, we find that some of the works do not exclusively lie in a single sub-category and span across multiple to cover and address multi-faceted challenges of the function resource management in FaaS. Table 2.2 summarise the categorised works under the proposed taxonomy.

Lin and Glikson [25] propose a pool-based approach where a number of warm function instances are maintained to proactively serve the incoming workload, thereby mitigating the latency associated with cold starts. This pool-based approach is implemented on the Knative platform and evaluated on single functions like HTTP server, enabling a significant reduction in response latency as compared to baseline Knative offerings. The authors [100] apply the statistical SARIMA forecasting model to predict function invocation requests, with the goal of mitigating cold start latency and reducing resource wastage. It proposes a prediction-based autoscaler to proactively scale the number of function instances and deploy them before incoming traffic. The work demonstrates how cold start and scaling are interdependent and evaluates the proposed autoscaler against the default Kubernetes horizontal pod autoscaler (HPA) with a single function application.

Mohan et al. [28] take a different approach to pre-creating and caching networking

resources, which are then dynamically linked to function containers. They suggest that cold start delays are frequently affected by the network initialisation times and propose a Pause Container Pool Manager (PCPM) strategy, implemented on open-source Apache OpenWhisk framework, that aims to reduce cold start latency and its concurrency impact. The study establishes its claim by experimenting with simple compute-intensive function and demonstrate a significant reduction in function start up time, i.e., a relatively faster execution time, and a negligible memory footprint to maintain the network resources. Exploring another approach, Silva et al. [101], evaluates a 'prebaking' technique to minimise the cold start latency of serverless functions. It leverages process C/R technique to create and restore function process snapshots from previously executed functions. This improves the function lifecycle (startup) and enhances resource utilisation by re-using function instances. The prototype is integrated with OpenFaaS control plane and evaluated on single function applications such as image-resizing and markdown renderer. The works discussed above collectively target the proactive precreation of function execution environments to mitigate cold start latency.

Research by Xu et al. [81] explores an adaptive container pool scaling strategy integrated with an LSTM-based container warm-up strategy to model cold start minimisation as an optimisation problem. The primary objective is to reduce cold start latencies and minimise resource consumption in a function's lifetime. In doing this, the research utilises the function chain model and ML techniques for accurate function invocation time prediction. Simultaneously, it adjusts the number of containers to reduce the resource waste. The authors realise the serverless platform on Kubernetes and perform experiments with a license plate recognition application with multiple functions. Hence, this work is classified under cold start management that proposes a ML-based solution and evaluated on an open-source framework with workflow applications.

Pagurus [102] presents a novel container management system that takes a holistic approach of reducing cold start latency, balance the workload and schedule idle containers amongst functions. It aims to share warm, idle containers between functions without introducing security issues and utilises a zygote image that does not contain any code or data of the previous function to be assigned to new function. By doing so, the study evaluates the proposal on single function applications where it effectively reduces the

instance startup latency for mid- and low- popularity functions while improving the user perceived latency. Similarly, Solaiman and Adnan [30] introduces WLEC, a container management architecture designed to mitigate the cold start problem in serverless computing, particularly within the OpenLambda platform. The architecture uses a modified S2LRU structure, a cache replacement policy, with a warm-up queue, and its performance is evaluated using a real-time image-resizing workflow. Results show reduced cold start latency and lower memory consumption compared to traditional approaches, highlighting its effectiveness for improving function execution performance. In a separate work, Mahajan et al. [103] proposes a new system based on a peer-to-peer network, virtual file system, and content-addressable storage. The goal is to improve compute availability, reduce storage requirements, and prevent network bottlenecks by exploiting data similarities across applications. While the approach focuses on instance deployments rather than application-specific optimisation, it provides an alternative approach to minimising cold start latency and improves resource utilisation and data locality for serverless workloads. The authors perform experiments over AWS platform with applications such as big data processing and simple HTTP server.

Research works [73], [80], [78], [72] introduce the paradigm of RL to the FaaS platforms in different ways. [73] focuses on request-based provisioning of VMs or containers on the Knative platform. The authors demonstrated a correlation between latency and throughput with function concurrency levels and thus propose a Q-Learning model to determine the optimal concurrency level of a function for a single workload. [80] proposed a two-layer adaptive approach, an RL algorithm to predict the best idle-container window, and an LSTM network to predict future invocation times to keep the pre-warmed containers ready. The study demonstrated the advantages of the proposed solution on the OpenWhisk platform using a simple HTTP-based workload and a synthetic demand pattern. Another research [78] focused on resource-based scaling configuration (CPU utilisation) of OpenFaaS and adjusts the HPA settings using an RL-based agent. They assumed a serverless-edge application scenario and a synthetic demand pattern for the experimentation and present their preliminary findings based on latency as SLA. Agarwal et al. [72] introduced the idea of Q-learning to ascertain the appropriate amount of resources to reduce frequent cold starts. The authors shared the preliminary

training results with an attempt to show the applicability of reinforcement learning to the serverless environment. They utilised the platform exposed resource metrics to experiment with a synthetic workload trace, i.e. Fibonacci series calculation, to simulate a compute-intensive application and predict the required resources.

Complementary to research focused purely on cold start mitigation, a significant body of work investigates function autoscaling within the resource management domain. These two aspects are intrinsically interdependent as function scaling decisions directly influence cold start frequency and latency, and conversely, the risk of cold starts constrains the optimal scale-down policy to avoid premature resource de-allocation. Therefore, advanced strategies often span both domains, seeking an overall optimisation that simultaneously reduces cold start delays and improves overall system throughput and resource utilisation. Mahmoudi and Khaziei [104] presents an analytical performance model for serverless computing, focusing on autoscaling and resource utilisation, validated on AWS Lambda. They identify three types of scaling: scale-per-request, concurrency value scaling and metrics-based scaling while proposing a scale-per-request model with other steady state metrics including rate of cold starts based on the system and workload characteristics. The model provides insights to the working of the FaaS platform to improve performance, resource utilisation and reduce operational costs by predicting essential function metrics and allowing for tuning.

Schuler et al. [73] investigates the application of RL, specifically Q-learning, to optimise the autoscaling of serverless applications on the Knative platform. The study focuses on adjusting the concurrency limit (number of requests processed per instance) to improve throughput and reduce latency. The results from the baseline experiment, done with CPU and memory intensive function, demonstrate that different concurrency levels affect performance. The research [78] investigates the use of RL for resource-based autoscaling in OpenFaaS. The focus is on improving resource utilisation, function lifecycle (including cold start), and function performance. It performs analysis on OpenFaaS with single function applications such as HTML based webpage. Similarly, Zhang et al. [76] proposes a RL-based autoscaling technique for delay-sensitive applications. The approach centres around characterising the service's resource profile through Q-learning, exploring performance improvements with different resource allocations, and propos-

ing a hybrid scaling strategy combining both horizontal and vertical scaling. The objective of Kubernetes-based adaptive scaling is to balance QoS with resource efficiency and the experimental analysis with benchmark functions such as face-blur demonstrate that the approach ensures reduction in response latency while reducing operational costs. Following the RL adoption, the study [74] focuses on developing RL environments and agents, particularly with Q-learning, DynaQ+, and Deep Q-learning algorithms, to autonomously manage dynamic workloads and meet QoS requirements while efficiently utilising resources. The performance of these RL-assisted autoscaling mechanisms is evaluated in both real and simulated environments, taking advantage of the Kubeless serverless platform. The aim is to make the management of resources more efficient, including minimising the cost and maximising the benefits of the serverless computing paradigm.

CoScal [77] takes a microservices approach and introduces a multi-faceted autoscaling strategy addressing challenges in resource scaling and QoS. It combines horizontal, vertical scaling, and brownout techniques and leverages RL for decision-making. The framework incorporates a workload prediction algorithm based on GRU to achieve efficient scaling. The effectiveness of CoScal is evaluated through experiments on a containerised microservices prototype system using Alibaba traces and benchmark microservices applications such as Sock Shop and Stans Robot Shop. This multi-faceted approach with RL can reduce the response time and the number of failed requests by improving QoS of the microservices.

Furthermore, Phung and Kim [75] presents a prediction-based auto-scaling approach, specifically targeting Knative to enhance performance, minimise resource utilisation, and optimise response times to meet QoS requirements. It leverages Bi-LSTM, a forecasting model, to improve the calculation of the number of function instances, and adapt to workload changes efficiently. The studys preliminary experiments with resource intensive function suggest that the method achieves better performance in contrast with Knative's default auto-scaling scheme. Li et al. [79] introduces KneeScale, an adaptive auto-scaler designed to optimise the scaling of serverless functions in edge computing environments. It aims to efficiently allocate resources and minimise costs under budget constraints by dynamically adjusting the number of function instances. KneeScale lever-

ages a search algorithm like binary search to detect the knee point of FaaS platforms after which there are diminishing gains of scaling and finds the optimal allocation. This approach distinguishes itself by addressing challenges specific to edge computing, such as resource limitations and dynamic workloads. Experimental results on OpenFaaS show KneeScale's effectiveness in enhancing performance and resource efficiency when compared to existing autoscaling techniques.

The common thread linking these efforts is the treatment of scaling, concurrency, and instance adjustment as an integral part of the resource allocation problem. This highlights that the adjustment of function instances or concurrency acts as a tunable configuration parameter, alongside the actual function-level configuration parameters (e.g., memory, CPU shares, timeout) exposed by CSPs or available on open-source platforms. Both sets of parameters, those directly affecting scaling/concurrency and those defining function-level resource needs are essential for bridging the gap between static resource assignment and dynamic workload demand. Zubko et al. [105] introduces MAFF, a Python-based framework for automatically finding optimal memory configurations for FaaS functions. MAFF focuses on cost-only and balanced optimisation objectives and utilises linear, binary, and gradient descent algorithms. The framework is tested on AWS Lambda, demonstrating the benefits of self-adaptive memory configurations. FireFace [106] presents an adaptive approach for configuring serverless functions on OpenFaaS. It aims to minimise costs and meet SLOs by predicting execution time under different configurations and using Adaptive Particle Swarm Optimisation algorithm and Genetic Algorithm Operator (APSO-GA) to find optimal CPU/memory and edge platform settings. To aid the decision making process, it uses the prediction module that extracts the internal features of all functions within the serverless application and uses this information to predict the execution time of the functions under specific configuration schemes. The experimental analysis is done on Java-based workflows and Deep Neural Network applications. A work by Moghimi et al. [33] introduces Parrotfish, a tool for optimising the rightsizing of serverless functions. It models the relationship between function memory and execution time that utilises an online parametric regression to recommend the near-optimal function configurations. The tool aims to reduce the function operating costs while ensuring lifecycle time expectations compared to existing techniques by

using a cost-aware sampling strategy. The recommendation tool is evaluated on AWS Lambda where the recommendations are made for various single function applications from benchmark serverless suite [107].

SLAM [45] is a tool designed to optimise memory configurations for FaaS functions within serverless applications to meet SLOs and balance performance and cost. The approach uses distributed tracing and performance modelling that learns the relationship between functions of an application to estimate execution times and then utilises a SLO-guided algorithm to determine the best memory settings on AWS Lambda. The tool aims to provide configurations that ensure high performance while optimising the cost, showing high accuracy in function capacity estimation. Similarly, Wen et al. [108] propose StepConf implemented over AWS Lambda and Step Function, that addresses the complex challenge of SLO-aware dynamic resource configuration for serverless function workflows. The approach revolves around three key configuration dimensions that impact performance: memory size, inter-function parallelism (concurrent functions), and intra-function parallelism (multi-threading within a single function instance). StepConf transforms the overall workflow problem into a per-step configuration decision problem, dynamically optimising memory and parallelism in real-time before each function step runs. This heuristic approach utilises critical path algorithm and aims to guarantee end-to-end SLOs while significantly saving costs. In a separate work [109], the researchers estimate the Function Capacity (FC) of serverless functions via a tool called FnCapacity. It uses statistical and ML models, specifically DNNs, to build capacity models and assess the impact of different configurations on performance. The evaluation of tool is demonstrated on Google Cloud Functions and AWS Lambda with benchmark serverless functions [110] like linpack and sentiment analysis.

COSE [93] introduces a framework that uses Bayesian Optimisation (BO), a ML technique, to find the optimal resource configuration and placement for functions. It focuses on minimising cost while meeting performance criteria (SLOs) in the presence of function chains. The evaluation of COSE is performed on AWS Lambda with various real-world applications and also over a wide range of simulated distributed cloud environments, which confirms the efficiency of the proposed approach. Bilal et al. [20] points out the inefficiencies of coupled resource allocation and harness surrogate model

variations of BO technique, considered as a stochastic and approximate optimisation method that is widely used in ML, to evaluate the performance impact of decoupled CPU and memory configurations. The research demonstrates a rich trade-off space between function execution time and cost achievable through flexible resource allocation and VM selection by CSPs. The study was conducted by executing and measuring the performance of six diverse benchmark functions on an open-source Kubernetes distribution atop AWS EC2 [47] instances. The research [111] investigates resource scaling strategies in open-source FaaS platforms, contrasting them with commercial cloud offerings like AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. It centres on the evaluation of resource allocation and scheduling. The paper proposes a methodology to compare the performance of different platforms, focusing on CPU intensive functions. Addressing the noisy neighbour problem and suggesting a pricing scheme based on resource scaling, it aims to provide a QoS-compliant resource allocation. The work is relevant in the context of function configuration and aims at an application-specific resource optimisation via the performance comparison to provide an on-premise hosting. The paper contributes to the understanding of resource efficiency, execution time, and scaling strategies in the serverless domain.

Bhasi et al. [112] consider latency-critical dynamic DAGs to minimise the function resources, particularly the created function instances, by tuning the number of requests served by an instance without violating the SLO. For this, they introduce Kraken, a resource management framework that employ hybrid scaling approach, both proactive and reactive, to avoid the cold start latency impact on end-to-end response times. Kraken is implemented on OpenFaaS and model application DAGs as Variable Order Markov Model (VOMM) to calculate the function invocation probabilities and employ Exponentially Weighted Moving Average (EWMA) model to predict the incoming load leveraged by the proactive scaling module. In a separate work, Bhasi et al. [17] presents Cypress, an input-size-sensitive resource management framework for serverless functions. Cypress uses function profiling, request batching, and reordering techniques in addition to novel proactive autoscalers to optimise resource allocation, reducing function instance usage while maintaining SLOs. The framework is implemented on a Kubernetes cluster using OpenFaaS and evaluated on real-world workflow applications,

demonstrating improved resource utilisation.

Zhu et al. [113] takes a model-driven approach designed for deployment optimisation in FaaS environments to introduce RDOF. It leverages Layered Queuing Networks (LQNs) to predict performance metrics and employs a genetic algorithm (GA) to determine the optimal function configurations that minimise costs while adhering to performance requirements. The research validates RDOF on AWS, demonstrating its effectiveness in achieving significant cost savings compared to baseline methods. The focus is on optimising function memory and concurrency settings within function chains. This approach utilises formal and analytical modelling and stochastic optimisation techniques for addressing the optimisation problem. Lin and Khazaei [88] leverage performance and cost models to transform DAG-based function workflows into a linear structure and propose a heuristic called Probability Refined Critical Path (PRCP) algorithm. The researchers utilise PRCP to optimise the function memory allocation for both budget-constrained performance and performance-constrained runtime cost by introducing different benefit-cost ratio strategies. Building on the previous work, the research in [114] explores and optimises memory configuration in serverless workflow applications, aiming for a balance between execution time and cost. The paper introduces a modified performance model and presents two optimisation algorithms, a heuristic Urgency-based Workflow Configuration (UWC), to obtain a memory configuration under the budget constraints that minimises the execution time, and a meta-heuristic Beetle Antennae Search (BAS) to avoid locally optimal solutions, which are evaluated on AWS Lambda.

Tomaras et al. [35] suggest that functions in a big-data application with similar characteristics, i.e., code-base, will have similar performance and resource requirements. To this end, they utilise Graph Edit Distance, that can be translated as a shortest path finding mechanism, to find similarity between the call graphs of two functions and utilise the prediction model of one, either wholly or partly, to provision a similar amount of resources. Jarachanthan et al. [96] address the resource configuration in data analytics or Map-Reduce style jobs to enhance performance and reduce operational costs. The authors implement these applications on AWS Lambda to efficiently allocate resources across pipeline functions. Furthermore, the study emphasise the difficulty involved in intermediate data exchange of data analytics applications and seeks optimally config-

ured function parameters, such as degree of parallelism, resource allocation and number of stages, to deal with the complexity of ephemeral data management. It develops function performance and cost models based on user requirements and formulate resource optimisation as a Shortest Path problem in Graph theory.

In another work [82], the researchers find an opportunity to co-locate functions and improve data locality. The study suggests that function outputs are immutable, only adding new value to a new location owing to function idempotency. To this end, an explicit, pre-determined data intent declaration approach is proposed, implemented as an Apache OpenWhisk controller, where the developers can specify the input and output location along with the function declaration. This allows easier function co-location, data caching and data locality optimisations while improving performance. Experimenting with a new direction, Mvondo et al. [18] introduces a system that improves function execution time by creating a transparent, in-memory cache using the memory capacity that would otherwise be wasted. Using a ML model, the proposed system estimates the actual memory resources required by each function invocation and utilises the difference between the memory reserved by the tenant and the prediction to aid the cache. It further includes a data-locality aware scheduler targeted to improve execution time and resource utilisation by opportunistically using idle memory. It is implemented on OpenWhisk platform and evaluated on single as well multi-function applications.

Research works [85][86][95] target video analysis and processing pipelines to configure various resources like memory or CPU, including application-specific resources such as the number of functions in a stage or video frames to reduce lifecycle time and associated operating costs. The researchers in [86] exploit the relationship between the memory configuration, input workload and performance for serverless video processing pipelines implemented on AWS Lambda. They suggest a ML approach, a Sequential Bayesian Optimisation-based stage-wise configuration finder that improves relative processing time by up to 408% while satisfying workflow runtime budget. Wang et al. [85] turn towards the configuration of serverless-based video analytics applications, where they investigate joint optimisation of use-case-specific controllable parameters such as frame rate and Deep Neural Network (DNN) selection in addition to computational resource configuration. To this end, the authors formulate the cost-effective configuration

problem as a stochastic process and find near-optimal solutions using Markov Approximation with lower computational overhead. The study [95] posits that users are still required to exhaustively tune and configure the resources in a video analytics workflow. Further, it identifies that large configuration space, input-dependent workflow execution and dynamic adjustment of per invocation resources are the major challenges in the video auto-tuning process. For this purpose, a collaboration of techniques such as dynamic slack calculation, safe delayed batching, early speculation, late commit and priority-based commit is proposed that automatically tunes each invocation for pipeline latency targets while minimising cost.

Kaffes et al. [115] present the design of a centralised, core-granular scheduler for functions, focusing on improving performance predictability, reducing cold start latency, and handling bursty workloads. The approach aims to address the shortcomings of existing coarse-grained scheduling methods of commercial as well as open-source schedulers and leverages CPU core-level granularity and a global view of cluster resources to schedule the function executions to individual CPU cores. The design is inspired by the characteristics of single serverless functions and uses analytical modelling derived from queuing theory to support the fundamental argument for centralised scheduling. Kim et al. [116] propose a fine-grained CPU capacity control solution for serverless functions. The solution leverages control theory and group-aware scheduling to dynamically adjust CPU usage limits to minimise resource contention and improve performance, specifically reduction of skewness in response time distribution and the reduction of average response time. The group-aware scheduling ensures that functions are grouped, dispatched and scaled based on the shared or similar performance requirements. The authors develop an analytical performance model for platforms like Knative and Google Cloud Run and evaluate the solution on open-source Dask framework [117] with CPU intensive functions to improve QoS and utilisation.

Similarly, Ensure [118] classifies the individual functions as edge-triggered or massively parallel based on resource consumption and lifetime patterns to dynamically regulate the CPU shares for functions executing on the same CPU core. Its scheduling component, FnSched [119], schedules requests by concentrating load on an adequate number of underlying hosts. The scaling component, FnScale, proactively scales the

instances based on the square-root staffing policy to maintain warm containers to handle burst of loads. The design of Ensure is grounded in formal queuing theory model for both scheduling and scaling. It is implemented on OpenWhisk platform and targets the cost efficiency of maintaining the hosts by CSPs while providing acceptable latency SLO. Researchers in [120] presents JIAGU, a serverless system aiming to improve resource utilisation, especially focusing on the challenges of dynamic resource allocation. The proposed system introduces pre-decision scheduling, which leverages a ML model, specifically Random Forest Regression for performance prediction, and dual-staged scaling that implements a fast-path finding approach, to balance resource use and cold start overheads. JIAGU is implemented as a prototype based on the open-source OpenFaaS and its evaluation with real-world workload traces from Huawei and benchmark single function applications [107][110] shows a significant improvement in instance density compared to commercial clouds while reducing cold start latency.

Orion [98] takes a holistic approach and introduces a novel technique for performance modelling and optimisation of serverless applications structured as DAGs. It focuses on function right-sizing, instance bundling, and right pre-warming to meet end-to-end latency probabilistic guarantees and minimise cost. The approach uses a performance model to estimate end-to-end latency and is evaluated on AWS Lambda for video analytics, ML, and chatbot applications. The method utilises a combination of Best-First Search and a performance model to determine the appropriate resource allocation and pre-warming strategies in order to meet latency objectives while minimising cost. On the other hand, Yu et al. [121] follows a data-centric approach and proposes a novel serverless computing platform to efficiently orchestrate complex function workflows. It employs a two-tier distributed scheduling hierarchy consisting of local schedulers on worker nodes and shared global coordinators with an objective to minimise the amount of data transmitted across worker nodes. This platform is built atop Cloudburst [126] and employs greedy scheduling algorithm due to the NP-hard nature of scheduling problem, however, achieving negligible data-exchange overhead and low-latency function execution.

Zhuo et al. [122] presents Aquatope, a QoS-aware resource scheduler for serverless workflows. It addresses cold starts and resource allocation by utilising a Bayesian

**Table 2.2:** A classification of resource management techniques in Serverless Computing

| | | WLC | | Env | | KPI | | | Targets | | | Strategy | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Work** | **Asp** | **SF** | **WF** | **Comm** | **OS** | **Res Ut** | **Fn LC** | **DL** | **Coupled** | **Decou** | **App Sp** | **SP** | **Approx** | **ML** | **Formal** |
| [25] | CS | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [100] | CS,Scl | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| [28] | CS | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [101] | CS | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [81] | CS,Scl | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [102] | CS | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [30] | CS | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [103] | CS | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [73] | CS,Scl | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [80] | CS | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [78] | CS,Scl | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [72] | CS | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| [104] | Scl | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| [76] | Scl | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [74] | Scl | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [77] | Scl | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [75] | Scl | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [79] | Scl | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| [105] | Cfg | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| [111] | Cfg,Sch | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| [115] | Sch | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| [120] | Sch | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [121] | Sch | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| [122] | Sch | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ |
| [123] | Cfg | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| [17] | Sch | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| [112] | Sch | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| [20] | Cfg | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |
| [124] | Cfg | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [83] | Cfg | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [96] | Cfg | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| [109] | Cfg | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| [116] | Cfg, Sch | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| [106] | Cfg | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| [114] | Cfg | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| [88] | Cfg | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| [98] | Cfg,Sch | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| [104] | Scl | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| [33] | Cfg | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [18] | Cfg,Sch | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [95] | Cfg | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [45] | Cfg | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| [125] | Scl | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| [34] | Cfg | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| [118] | Sch | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| [82] | Sch | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| [35] | Sch | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| [85] | Cfg,Sch | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |

**Legend and Abbreviations:**

✓ Covered or addressed in detail
✗ Not addressed or focused

**Column Group Abbreviations:**

| | | | |
|---|---|---|---|
| Asp. | Management Aspect | WLC | Workload Characterisation |
| Env | Deployment Environment | KPI | Key Performance Indicator |
| Targets | Resource Optimisation Targets | Strategy | Optimisation Strategy |

**Sub-Column Abbreviations:**

| | | | |
|---|---|---|---|
| CS | Cold Start | Scl | AutoScaling |
| Cfg | Function Configuration | Sch | Function Scheduling |
| SF | Single Function | WF | Workflow/Function Chain |
| Comm | Commercial (Environment) | OS | Open Source (Environment) |
| Res Ut | Resource Utilisation | Fn LC | Function Lifecycle |
| DL | Data Locality & Exchange | Coupled | Coupled (e.g., Memory/CPU) |
| Decou | Decoupled (e.g., CPU/Disk) | App Sp | Application Specific |
| SP | Search & Path Finding | Approx | Stochastic & Approximate Optimisation |
| ML | Machine Learning | Formal | Formal & Analytical Modelling |

approach. The system optimises the pre-warmed container pool and function-level resource allocation with a BO algorithm. It aims at meeting QoS requirements while minimising resource consumption. The scheduler is evaluated on OpenWhisk, demonstrating improvements in function performance and resource utilisation. The work in [124] focuses on automating the configuration of resources, addressing the performance and cost implications of coupled resource settings. It introduces a workload-agnostic method, CPU Time Accounting Memory Selection (CPU-TAMS) that leads to the maximum value configuration i.e., the highest performance at lowest cost. The method is implemented across four commercial FaaS platforms, AWS Lambda, Google Cloud Functions (GCF), IBM Cloud Functions (IBM), and DigitalOcean Functions (DOF) with various individual functions. The core of CPU-TAMS is a platform-specific vCPU-to-memory model used as a regression model to map the required number of utilised CPU shares to the appropriate memory setting.

Lachesis [34] is an online learning-based resource allocation framework for dynamically assigning decoupled resources per-function invocation. The authors show that Lachesis improves execution time and decreases idle CPU cores (resource utilisation) compared to static allocations. The study examined different individual functions and evaluated them on OpenWhisk platform. At the core, it uses a cost-sensitive multi-class classification model to predict the required CPU core count based on input features and function semantics. Eismann et al. [83] presents an approach that uses ML, specifically multi-target regression modelling, to predict execution time based on a single monitoring dataset. The research contributes to automating function memory size optimisation, with the goal of reducing costs and enhancing performance. The training phase uses a large dataset generated from 2,000 synthetic serverless functions created by combining representative function segments and then evaluated on real-world function workflows. The approach is implemented as a custom monitoring solution and evaluated on AWS Lambda. LatEst [125] focuses on minimising the need for horizontal scaling in FaaS and proposes to scale the resources (CPU, memory) of active function instances within a few milliseconds. It is implemented as an extension of vHive [127] based on Knative platform. Furthermore, it implements a feedback control loop using a Proportional Integral Derivative (PID) controller, for each function, that models the relationship between

the incoming load and the required resource allocation. LatEst minimises the execution time variability by improving the cold start latency and aims to achieve high resource utilisation by only adding resource when needed.

## 2.5 Summary

In this chapter, we have presented a comprehensive literature survey on existing function resource management techniques. We propose a taxonomy that identifies various elements associated with the dynamic function configuration and its management within the broader scope of different function resources, available serverless platforms, and key performance indicators. We further discuss the key aspects of function configuration and management to analyse the existing works using the proposed taxonomy. This chapter presents a clear view on the resource management of serverless functions and aids developers to fine-tune their application performance and cost as well as CSPs to offer a true *serverless* experience. Further, this chapter provides a concrete reference point for researchers exploring resource allocation, scheduling, key performance indicators, and function configuration and management schemes in the serverless domain to advance the field. This thesis explores and addresses some of the identified research challenges. Beyond that, we further outline potential new research directions in the last chapter.

# Chapter 3

# A Q-Learning Powered Function Cold Start Frequency Mitigation

*This chapter focuses on a critical performance bottleneck of function cold starts. FaaS provides seamless autoscaling, generally through on-demand function instantiation, that introduces a non-negligible delay, known as the cold start, severely reducing the end-user latency. To address this limitation, in this chapter we focuses on reducing the frequent, on-demand cold starts on the platform by applying RL. We propose a novel approach based on model-free Q-learning that leverages key operational function metrics, including CPU utilisation, existing function instances, and response failure rate, to proactively initialise functions based on the expected demand. The proposed solution is implemented and evaluated on the Kubeless platform using an open-source Azure function invocation trace applied to a matrix multiplication function. Our evaluation demonstrates the superior performance of the RL-based agent when compared to Kubeless' default policy and a function keep-alive policy.*

## 3.1 Introduction

In serverless computing, the FaaS model has attracted a wide range of applications such as Internet of Things (IoT) services, REST APIs, stream processing and prediction services [90], which have strict availability and quality of service requirements in terms of response time. Conceptually, the FaaS model is designed to spin a new function instance for each demand request and shut down the instance after service [24]. However,

This chapter is derived from:

- **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, "On-demand Cold Start Frequency Reduction with Off-Policy Reinforcement Learning in Serverless Computing," *in Proceedings of the 2024 International Conference on Computational Intelligence and Data Analytics (ICCIDA 2024, Springer, Singapore),* Hyderabad, India, June 28-29, 2024.

in practice, commercial FaaS offerings like AWS Lambda, Azure Functions, and Google Cloud Run Function may choose to re-use a function instance or keep the instance running for a limited time to serve subsequent requests [12]. Some open source serverless frameworks like Kubeless [128] and Knative [10], have similar implementations to re-use an instance of a function to serve subsequent requests.

An increase in workload demand leads to an instantiation process involving the creation of new function containers and the initialisation of the function's environment within those containers, after which incoming requests are served. Hence, instantiating a function's container introduces a non-negligible time latency, known as cold start, and gives rise to a challenge for serverless platforms [25], [26], [27], [81]. Some application-specific factors such as programming language, runtime environment and code deployment size as well as function requirements like CPU and memory, affect the cold start of a function [27], [28], [29], [30]. To automate the process of creating new function instances and reusing existing ones, serverless frameworks usually rely on resource-based (CPU or memory) horizontal scaling, known as HPA in Kubernetes-based frameworks like Kubeless [128], to respond to incoming requests. Resource-based scaling policies implement a reactive approach and instantiate new functions only when resource usage rises above a pre-defined threshold, thus leading to cold start latencies and an increase in the number of unsuccessful requests.

Threshold-based scaling decisions fail to consider factors like varying application load and platform throughput and hence, pose an opportunity to explore dynamic techniques that analyse these factors to address cold starts. This chapter presents a model-free Q-learning agent to exploit resource utilisation, available function instances, and platform response failure rate to reduce the number of cold starts for CPU-intensive serverless functions. We define a reward function for the RL agent to dynamically establish the required number of function instances for a given workload demand based on expected average CPU utilisation and response failure rate. The RL-based agent interacts with the serverless environment by performing scaling actions and learns through trial and error during multiple iterations. The agent receives delayed feedback, either positive or negative, based upon the observed state, and consequently learns the appropriate number of function instances to fit the workload demand. This strategy uses no

prior knowledge about the environment, demand pattern or workload, and dynamically adjusts to the changes for preparing required functions in advance to reduce cold starts. The proposed agent scales the number of function instances by proactively estimating the number of functions that are needed to serve incoming workload to reduce the frequent cold starts. It utilises a practical workload of matrix multiplication involved in an image processing task, serving as a sample real-world function request pattern [129], and formally presents the cold start as an optimisation problem. Also, we structure the Q-learning components around the function metrics such as average CPU utilisation and response failure rate and evaluate our approach against the default resource-based policy and commercially accepted function keep-alive technique.

In summary, the key contributions of this chapter are:

1. We analyse function resource metrics such as CPU utilisation, available instances, and the proportion of unsuccessful responses to propose a Q-learning model that dynamically analyses the application request pattern and improves function throughput by reducing frequent cold starts on the platform.

2. We present a brief overview of explored solutions to address function cold starts and highlight the differences between contrasting approaches to the proposed agent.

3. We perform our experiments on a real-world system setup and evaluate the proposed RL-based agent against the default resource-based policy and a baseline keep-alive technique.

The rest of the chapter is organised as follows. Section 3.2 highlights related research studies. In Section 3.3, we present the system model and formulate the problem statement. Section 3.4 outlines the proposed agents workflow and describes the implementation hypothesis and assumptions. In Section 3.5, we evaluate our technique with the baseline approach and highlight training results and discuss about performance in Section 3.6. Section 3.7 summarises and concludes the chapter.

## 3.2 Related Work

In this section, we briefly discuss the works on the current function cold start mitigation techniques and approaches.

### 3.2.1 Function Cold Start and Mitigation

Researchers in [12] described function cold start as the time taken to execute a function. It broadly classified the approaches to deal with function cold start in, environment optimisation, and pinging. The former approach acts either by reducing container preparation time or decreasing the delay in loading function libraries, while the latter technique continuously monitors the functions and periodically pings them to keep the instances warm or running.

An adaptive container warm-up technique to reduce the cold start latency and a container pool strategy to reduce resource wastage is introduced in [81]. The proposed solution leverages a LSTM network to predict function invocation times and non-first functions in a chain to keep a warm queue of function containers ready. Although both the discussed techniques work in synchronisation, the first function in the chain suffered from a cold start.

The research in [25] explained platform-dependent overheads like pod provisioning and application implementation-dependent overheads. It presented a pool-based pre-warmed container technique, marked with selector 'app-label' to deal with the function cold start problem. To tackle the incoming demand, a container pool is checked first for existing pre-warmed containers, or the platform requests new containers as per the demand.

Another study [103] exploited the data similarity to reduce the function cold start. It criticised the current container deployment technique of pulling new container images from the storage bucket and introduced a live container migration over a peer-to-peer network. Similarly, [26] aimed to reduce the number of cold start occurrences by utilising the function composition knowledge. It presented an application-side solution based on lightweight middleware. This middleware enable the developers to control the frequency of cold start by treating the FaaS platform as a black box.

Based on the investigation in [28], network creation and initialisation were found to be the prime contributors to the cold start latency. The study expressed that cold starts are caused due to work and wait times involved in various set-up processes like initialising networking elements. The study explained the stages of the container lifecycle and states that the clean-up stage demands cycles from the underlying containerisation daemon, hindering other processes. Therefore, a paused container pool manager is proposed to pre-create a network for function containers and attach the new function containers to configured IP and network when required.

Some studies [27], [29], [130] have identified significant factors that affect the cold start of a function. These include runtime environment, CPU and memory requirements, code dependency setting, workload concurrency, and container networking requirements. Most works [131], [132], [133], [134], [135] focus on commercial FaaS platforms like AWS Lambda, Azure Functions, Google Cloud Functions and fall short to evaluate open source serverless platforms like OpenFaaS, Knative, Kubeless [128], etc. Very few studies [30], [136], [137] have successfully performed analysis on an open-source serverless platform and provided possible solution by targeting the container level fine-grained control of the platform.

Recent research works [73], [80], [78], [72] introduce the paradigm of RL to the FaaS platforms in different ways. [73] focuses on request-based provisioning of VMs or containers on the Knative platform. The authors demonstrated a correlation between latency and throughput with function concurrency levels and thus propose a Q-Learning model to determine the optimal concurrency level of a function for a single workload. [80] proposed a two-layer adaptive approach,an RL algorithm to predict the best idle-container window, and an LSTM network to predict future invocation times to keep the pre-warmed containers ready.

The study demonstrated the advantages of the proposed solution on the OpenWhisk platform using a simple HTTP-based workload and a synthetic demand pattern. Another research [78] focused on resource-based scaling configuration (CPU utilisation) of OpenFaaS and adjusts the HPA settings using an RL-based agent. They assumed a serverless-edge application scenario and a synthetic demand pattern for the experimentation and present their preliminary findings based on latency as SLA.

**Table 3.1:** Related work summary

| Work | Name | Platform | Solution Focus | Strategy | Application Type |
|---|---|---|---|---|---|
| [12] | - | AWS Lambda | Cold start latency | Optimising environments & function pinging | Concurrent & sequential CPU & I/O intensive |
| [81] | AWU & ACPS | Kubernetes | Cold start latency & resource wastage | Invocation prediction (LSTM) & Container pool | Function chain model |
| [25] | - | Knative | Cold start frequency | Container pool & pod migration | Single function model |
| [26] | Nave, Extended & Global Approach | AWS Lambda, Apache Open Whisk | Cold start frequency | Orchestration middleware | Function chain model |
| [28] | Pause Container Pool Manager | Apache OpenWhisk | Cold start latency | Container Pool | Function chain model |
| [30] | WLEC | OpenLambda | Cold start latency | Container Pool | Single function model |
| [103] | - | AWS | Cold start latency | Container migration & content similarity | Single function model |
| [73] | - | Knative | Cold start frequency | AI-based container concurrency | Emulated CPU & I/O intensive |
| [101] | Prebaking | OpenFaas | Cold start latency | CRIU process snapshot | Single function model |
| [80] | - | OpenWhisk | Cold start frequency | RL-based idle window & LSTM based container pre-warming | Single function model |
| [78] | - | OpenFaas | Function Scaling | RL & SLA-based configuration | Single function model |
| Our work | - | Kubeless | Cold start frequency | AI-based function & throughput metrics | Single function model |

Agarwal et al. [72] introduced the idea of Q-learning to ascertain the appropriate amount of resources to reduce frequent cold starts. The authors shared the preliminary training results with an attempt to show the applicability of reinforcement learning to the serverless environment. They utilised the platform exposed resource metrics to experiment with a synthetic workload trace, i.e. Fibonacci series calculation, to simulate a compute-intensive application and predict the required resources.

Our proposed approach introduces a Q-Learning strategy to reduce frequent cold starts in the FaaS environment. Contrasting existing solutions, we apply the model-free Q-Learning to determine the required number of function instances for the workload

demand that eventually reduces number of on-demand cold starts. Furthermore, the existing solutions takes advantage of either continuous pinging, pool-based approaches, container migration and network building or exploit platform-specific implementations like provisioned concurrency while failing to experiment with CPU-intensive real-world application workloads. Similar to [72], the proposed strategy utilises available resource-based metrics and response failure rate to accomplish the learning, but improves over the discussed approach. Contrasting to their model, we formulate the problem of cold starts as an optimisation approach to proactively spawn the required function instances and minimise frequent, on-demand cold starts.

As part of the Q-learning model in [72], the study used fixed-value constants in the reward modelling and we address this issue by carefully analysing the problem and curate it as a threshold-based reward system. Additionally, we experiment with *matrix multiplication* function, that can be used as part of image processing pipeline, to train and evaluate our agent and utilise the open-sourced function invocation trace [129] by Azure Functions. Further, we describe our design decisions and utilise constants based upon the trial-error analyses. The successful learning of the agent resulted in the preparation of near to optimal function instances in a timeframe to reduce the on-demand function creation or cold starts and improve the platform's throughput. A summary of related works is presented in Table 3.1.

## 3.3 System Model and Problem Formulation

FaaS is an event-driven cloud service model that allows stateless function deployment. It delivers high scalability and scale-to-zero feature being economical to infrequent demand patterns. New functions $n_i$, where $1 \leq n_i \leq N$ and $N$ is the maximum scale, are instantiated on-demand to serve the incoming load (scale up) and removed (scale down) when not in use after a certain time span or below a configured, resource-based threshold metric value for every $i$ iteration window. The preparation time of function containers i.e., cold start $C_t$, adds to the execution time of a request. These frequent on-demand cold starts result in an increased computation pressure on existing resources, neglecting expected average CPU utilisation ($\phi_o$), and expected request failure rate ($\tau_o$).

Therefore, an intelligent, learning-based solution is proposed to address them.

In this proposed approach, we consider Kubeless [128], an open-source Kubernetes-native [61] serverless platform that leverages Kubernetes primitives to provide serverless abstraction. It wraps function code inside a docker container with pre-defined resource requirements i.e. $RR_f = (cpu_f, mem_f, tout_f)$ and schedules them on worker nodes. Similar to commercial FaaS providers, Kubeless has an idle-container window of 5 minutes to re-use functions and scales down to a minimum of one function if the collected metrics (default 15 seconds window) are below the set threshold. We take into account the general illustration of FaaS platform and consider a stochastic incoming request pattern $D = \{d_1, d_2, \ldots, d_i\}$ with $d_i$ requests in $i^{th}$ iteration window. We analyse the request pattern for a timeframe $T$ divided in $i$ iteration windows of duration $t_i$. The system model of the examined scenario is depicted in Fig. 3.1. The workflow of the potential cold start is explained in Fig. 3.2.

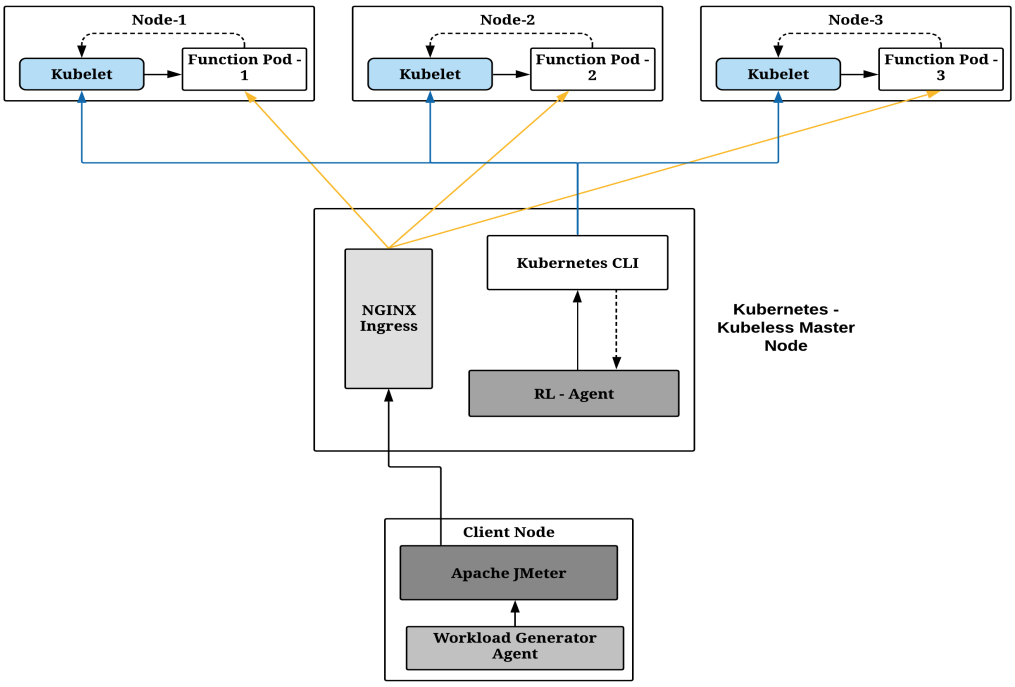



**Figure 3.1:** System Model

### 3.3.1 Problem Formulation

We formulate the function cold start as an optimisation problem aimed at minimising the number of cold starts (Eq. 3.1) by preparing required instances, beforehand and aid the agent in learning a policy to reduce the request failure rate while maintaining average CPU utilisation.

$$\min_{\phi,\tau,d_i} \quad (n_i) \tag{3.1}$$

**such that**

$$\tau_{d_i} < \tau_o;\ \phi_{d_i} < \phi_o \tag{3.2}$$

A cold start happens when there are no function instances available on the platform to deal with the incoming request and a new function instance is requested from the platform. FaaS services scale horizontally as per resource-based thresholds to be agile, usually considering the function's average CPU utilisation. Therefore, the goal of optimisation is to assess the incoming request pattern $d_i$ for an application task in $i^{th}$ iteration window and configure a policy to prepare functions beforehand, considering actual and expected average CPU utilisation ($\phi_{d_i}$&$\phi_o$) and request failure rate ($\tau_{d_i}$&$\tau_o$). Since the preparation time, $C_t$ remains similar for individual function containers, we focus on optimising the frequency of cold start $n_i$ for an individual iteration window.

With easy to implement and economical service model, enterprises are accommodating critical tasks like user verification [38], media processing [86], parallel scientific computations [98], anomaly detection and event-driven video streaming [37] into the serverless paradigm. To assess the necessity of a dynamic solution, we consider matrix multiplication as workload, which is an important task in image processing workflow.

#### Reinforcement Learning model

In a model-free Q-Learning process, the agent learns by exploring the environment and exploiting the acquired information. The core components of the environment are state, action, reward and agent. The environment state represents the current visibility of the agent and is defined as a Markov Decision Process (MDP) [138], [139] where future

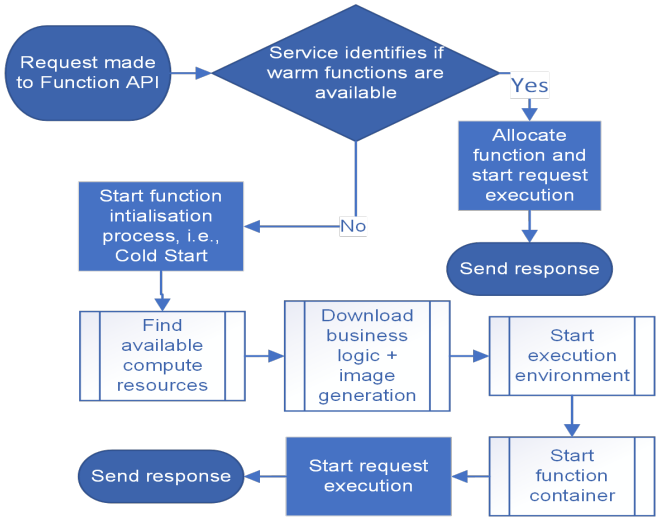



**Figure 3.2:** Function Warm Start & Cold Start workflow

environment state is independent of past states, given the present state information. Actions are the possible set of operations that the agent can perform in a particular state. Additionally, rewards are the guiding signals that lead the agent towards the desired goal by performing actions and transitioning between environment states. The agent maintains a Q-value table to assess the quality of action through obtained reward for the respective state and utilise it for future learning. Therefore, we propose a modelling scheme for the RL environment that is leveraged by a Q-Learning agent to learn a policy for function preparation.

We model the RL environment's state as $s_i = (\hat{n}_i, \phi_{d_i}, \tau_{d_i})$ where $\phi_{d_i}$ represents the average CPU utilisation of the available $\hat{n}_i$ function instances, $\tau_{d_i}$ represents the response failure rate, and $i$ is the iteration window during a timeframe $T$. The agent's task is to prepare the estimated number of function instances in the upcoming iteration window either by exploring or exploiting the suitable actions. These actions of adding or adjusting the number of function instances, compensate for any expected cold starts from the incoming demand and help to improve the throughput of the system. Therefore, we define the agent's action as the number of function instances, $n_i$, to be added or removed from currently available functions $\hat{n}_{i-1}$ and represent it as a set $a_i = \{n_i | 1 \leq (\hat{n}_{i-1} + a_i) \leq N\}$. This heuristic helps the agent to control the degree of explo-

ration by maintaining the number of functions within the threshold $N$, that is adapted based on deployed infrastructure capacity. Hence, we map the function resources and relevant metrics to RL environment primitives.

The motive of the RL-based agent is to learn an optimal policy, and we structure the rewards over resource-based metrics $\phi_{d_i}$, function response failure rate $\tau_{d_i}$, and expected threshold values ($\phi_o$ and $\tau_o$). It evaluates the quality of action $a_i$ in state $s_i$ by keeping a value-based table, i.e., Q-table, that captures this information for every $(s_i, a_i)$ pair. After executing the action, the agent waits for the duration of the iteration window and receives a delayed reward $r_i$, expressed based on the difference between the expected and actual utilisation and failure rate values, as shown in Eq. 3.3.

$$r_i = \frac{(\phi_o - \phi_{d_i}) + (\tau_o - \tau_{d_i})}{\hat{n}_i} \tag{3.3}$$

and the Q-table is represented as a matrix (Eq. 3.4) of dimension $S \times A$.

$$Q_{(S_n \times A_m)} = \begin{bmatrix} s_1, a_1 & \dots & s_1, a_m \\ \vdots & \ddots & \vdots \\ s_n, a_1 & \dots & s_n, a_m \end{bmatrix} \tag{3.4}$$

## 3.4 Q-Learning for Cold Start Reduction

Here, we apply model-free Q-learning algorithm in FaaS paradigm to reduce frequent on-demand function cold starts. We select this algorithm due to its simple and easy implementation, model interpretability, strong theoretical convergence guarantees, ability to process the perceived information quickly using the Bellman equation and its adaptability to other advanced algorithms like Deep Q-Learning (DQN). As discussed in Section **??**, we model the process of creating required function instances as an MDP and map the serverless computing primitives to RL agent's environment, state and actions. We explore and exploit an off-policy RL algorithm to reduce the on-demand function cold starts and determine the required function instances with the intuition of it being

easy, simple to implement, less complex with stable convergence in a discrete action space. The proposed approach has two phases: an agent training phase and a testing phase. Algorithm 1 demonstrates the agent training workflow. The environment setup process precedes the agent training, where the agent interacts with the environment and obtains information. After initial setup, the agent is trained for multiple epochs or time-frames where it assesses the function demand $d_i$ over individual iteration windows $i$ and ascertains appropriate function instances. During an iteration window $i$, the agent observes the environment state $s_i$, selects an action $a_i$ according to $\epsilon$-greedy policy. This greedy policy helps the agent to control its exploration and selects a random action with $\epsilon$ probability, otherwise exploiting the obtained information. This exploration rate is a dynamic value and decays with ongoing learning to prioritise the acquired information.

**Algorithm 1** Q-Learning for Cold Start Reduction

**Require:** Initialise Environment variables
**Ensure:** Initialise Q-Table, $decayRate$, $\epsilon$, $epoch$
  $\epsilon = 0.01 + 0.99\mathrm{e}^{(-decayRate \times epoch)}$
  Repeat for each $TrainingEpoch$
  $epoch \leftarrow epoch + 1$
  **while** $t_{elapsed} < T$ **do**
    $s_i \leftarrow currentState(\hat{n}_i, \phi_{d_i}, \tau_{d_i}, i)$
    $a_i \leftarrow$ choose using $\epsilon$-greedy policy from Q-Table
    Scale & wait for $i^{th}$ iteration window
    $r_i \leftarrow calculateReward(\phi_{d_i}, \tau_{d_i})$
    $s_{i+1} \leftarrow getNewState(\hat{n}_{i+1}, \phi_{d_{i+1}}, \tau_{d_{i+1}}, i+1)$
    $Q(s_i, a_i) \leftarrow (1-\alpha)Q(s_i, a_i + \alpha(r_i + \gamma \max_a Q(s_{i+1}, a_i))$
    $t_{elapsed} = t_{elapsed} + t_i$

After performing the selected action, the agent waits for duration $t_i$ of an iteration window to obtain the delayed reward $r_i$, calculated using the relevant resource-based metrics $\phi_{d_i}$ and function failure rate $\tau_{d_i}$. This reward helps the agent in action quality assessment, and it combines the acquired knowledge over previous iterations using the Bellman Equation (Eq. 3.5). It is the core component in learning as it aids Q-value or Q-table updates and improves the agent's value-based decision-making capability. The equation uses two hyper-parameters learning rate, $\alpha$ and discount factor, $\gamma$. The learning rate signifies the speed of learning and accumulating new information, and the discount factor balances the importance of immediate and future rewards.

$$Q(s_i, a_i) = (1 - \alpha)Q(s_i, a_i) + \alpha(r_i + \gamma \max_a Q(s_{i+1}, a_i)) \tag{3.5}$$

The agent then evaluates and adjusts the Q-value in Q-Table based upon the delayed reward for the corresponding $(s_i, a_i)$ pair. The agent continues to analyse the demand over multiple iteration windows, selecting and performing actions, evaluating delayed rewards, assessing the quality of action and accumulating the information in Q-table, and repeating this process over multiple epochs for learning. Once the agent is trained for sufficient epochs and the exploration rate has decayed significantly, we can exploit obtained knowledge in the testing phase.

In the testing phase, the agent is evaluated using a demand pattern for the matrix multiplication function and the Q-table values guide the agent in taking informed actions. The agent determines the current environment state and obtains the best possible action i.e. action with the highest Q-value for the corresponding state, and adjusts the required number of functions based on its understanding of the demand. We hypothesise that there exists a relationship between throughput and function availability to serve incoming requests. Therefore, we evaluate the agent's performance by considering metrics such as system throughput, function resource utilisation and available function instances. We further hypothesise that the RL-based agent learns to prepare and adjust required number of functions beforehand and improve the throughput while keeping the function's resource utilisation below the expected threshold.

## 3.5 Performance Evaluation

In this section, we provide the experimental setup and parameters, and perform an analysis of our agent compared to other complementary solutions.

### 3.5.1 System Setup

We set up our experimental test-bed as discussed in Section **??**, using NeCTAR (Australian National Research Cloud Infrastructure) services on the Melbourne Research Cloud. We configure Kubernetes (v1.18.6) and Kubeless (v1.0.6) on a service cluster

**Table 3.2:** System Setup Parameter values

| **Parameter Name** | ***Value*** |
|---|---|
| Kubernetes version | v1.18.6 |
| Kubeless version | v1.0.6 |
| Nodes | 4 |
| OS | Ubuntu 18.04 LTS |
| vCPU | 4 |
| RAM | 16 GB |
| Workload | Matrix Multiplication ($m \times m$) |
| m | 1024 |

of 4 nodes, each with Ubuntu (18.04 LTS) OS image, 4 vCPUs, 16 GB RAM, and 30 GB of disk storage to perform the relevant experiments. Typical serverless applications expect high scalability for their changing demands and can be compute-intensive, demanding a considerable amount of resources such as CPU, memory, or time to execute. These factors add to frequent cold starts on the platform by keeping the available functions or resources busy while requesting new functions for the subsequent workload demand. We use Python-based matrix multiplication (1024 pixels x 1024 pixels) to mimic the image processing task as our latency-critical application to deploy serverless functions.

The experimental setup mimics real-time application demand experienced in commercial FaaS platforms [140], [129]. We consider a single function invocation trace from the open-source Azure function data [129] and downsize it according to our resource capacity. We deploy the Apache JMeter [141] load testing tool to generate the HTTP-based requests and randomise its request ramp-up period to guarantee the changing demand pattern for our workload. Also, we collect the relevant resource-based metrics and throughput information via Kubernetes APIs. Table 3.2 summarises the parameters used for system setup.

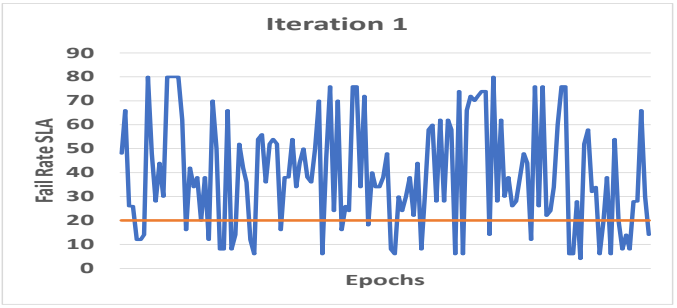



**Figure 3.3:** Training iteration 1

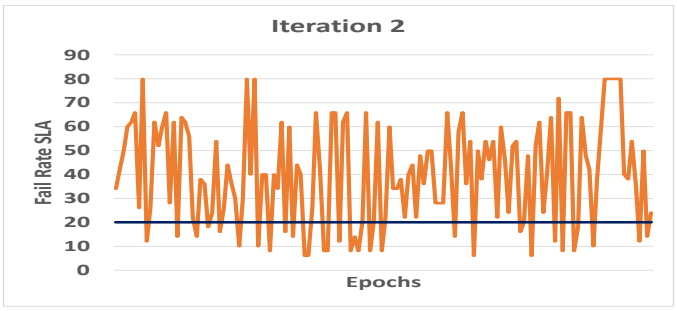



**Figure 3.4:** Training iteration 2

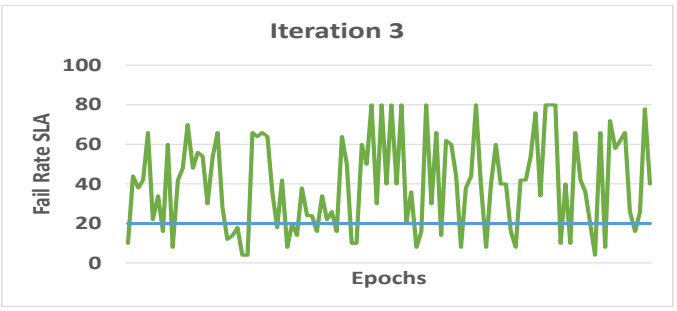



**Figure 3.5:** Training iteration 3

### 3.5.2 RL Environment Setup

To initialise the proposed RL-based environment, we first analyse and set up the function requirements according to deployed resource limits. After preliminary analysis, we configure the function requirements as 1 vCPU, 128 MB memory, and 60 seconds function timeout, where timeout represents the maximum execution period for a function until failure. To experiment we assume a timeframe of 10 minutes to analyse the demand pattern of 100 requests during 5 iteration windows of 2 minutes. Based on

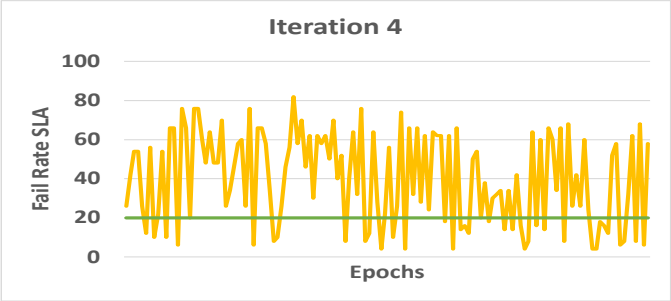



**Figure 3.6:** Training iteration 4

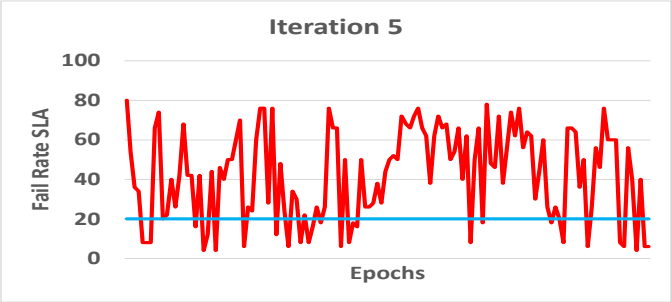



**Figure 3.7:** Training iteration 5

the resource analysis and underlying Kubernetes assets we assume the function limit $N = 7$. These constraints allow us to put a considerable load or pressure on the different techniques discussed and effectively evaluate them against each other.

As discussed in Section **??**, the RL-environment components depend upon resource metrics (average CPU utilisation), response failure rate, number of available functions and expected threshold values, summarised in Table 3.3. Since the proposed agent maintains a Q-table, these considerations help to minimise the risk of state-space explosion related to Q-Learning. The actions signify the addition or removal of functions based upon the function limit and the reward is modelled around the expected threshold values. We configure the Bellman Equation hyper-parameters: learning rate and discount factor as 0.9 and 0.99, respectively, based on the results of hyper-parameter tuning in [72]. The agent is structured to explore the environment and exploit the acquired knowledge. We use $\epsilon$-greedy action selection policy to randomly select an action with initial $\epsilon = 1$ probability and exploit this information with a decay rate of 0.0025. These RL system parameter values were chosen after careful consideration of discussed workload

and invocation pattern, according to the underlying resource capacity, and to showcase the applicability of RL-based agent in a serverless environment.

**Table 3.3:** RL-Environment parameter values

| **Parameter** | ***Value*** |
|---|---|
| $cpu_f, mem_f, tout_f$ | 1, 128M, 60 seconds |
| $N$ | 7 |
| $T$ | 10 minutes |
| $i$ | 5 |
| $t_i$ | 2 minutes |
| $\phi_o$ | 75% |
| $\tau_o$ | 20% |
| $\alpha$ | 0.9 |
| $\gamma$ | 0.99 |
| $\epsilon$ | 1 |
| $decayRate$ | 0.0025 |

### 3.5.3 Q-Learning Agent Evaluation

We train the RL-based agent for a timeframe of 10 minutes over 500 epochs to analyse an application demand and learn the ideal number of functions to reduce frequent cold starts. The agent is structured according to the RL-based environment design explained in section **??** and around the implementation constraints. The quality of the RL-based agent is evaluated during a 2-hour period to reduce the effect of any bias and performance bottlenecks.

We assess the effectiveness of our approach against the default scaling policy and commercially used function keep-alive policy on the serverless platform. Kubeless leverages the default resource-based scaling (HPA) implemented as a control loop that checks for the specific target metrics to adjust the function replicas. HPA has a default query period of 15 seconds to check and control the deployment based on the target metrics like

average CPU utilisation. Therefore, the HPA controller fetches the specific metrics from the underlying API and calculates the average metric values for the available function instances. The controller adjusts the desired number of instances based on threshold violation but is unaware of the demand and only scales after a 15-second metric collection window. The expected threshold for function average CPU utilisation is set to be 75% with maximum scaling up to 7 instances. Therefore, whenever the average CPU utilisation of the function violates the threshold, new function instances are provisioned in real-time, representing a potential cold start in the system.

Also, HPA has a 5-minute down-scaling window and during that period resources are bound to the platform irrespective of incoming demand which represents potential resource wastage. Therefore, it is worthwhile to analyse the performance of the RL-based agent against the function queuing or keep-alive approach that keeps enough resources bound to itself for an idle-container window.

Fig. 3.3 - 3.7 illustrates the learning curve of the agent over multiple epochs and we observe that the agent continuously attempts to meet the expected thresholds. This highlights the agent's capability to obtain positive rewards and move towards the desired configuration. We compare the RL-based agent with HPA and successfully demonstrate the agent's ability to improve the function throughput i.e., reduce the failure rate by up to 8.81%, Fig. 3.8. The RL-based agent further targets to maintain the expected CPU utilisation thresholds, Fig. 3.9, by reducing CPU stress up to 55% while determining the required function instances in Fig. 3.10. For example, in Fig. 3.10 during iteration windows 1 and 2, the HPA scales functions based on CPU utilisation threshold, unaware of the actual requirement for upcoming iteration and results in resource wastage. Similarly, Fig. 3.10 illustrates the resource wastage by HPA during iterations 3 and 4.

Similar results are observed against function queuing or keep-alive policy, where we evaluate two queues with $N = 4$ and $N = 7$. The RL-based agent scales and prepares the function according to demand needs while the queue results in resource wastage of up to 37%, as shown in Fig. 3.11. Although the queuing policy manages to reduce the request failure rate to zero, it is due to extra resources available, as depicted in Fig. 3.12, but can not be precisely captured by HPA metrics and shows over CPU utilisation of up to 50% in Fig. 3.13. The proposed agent analysed the demand pattern by consum-

ing sufficient function resources, preparing the ideal number of functions and trying to keep the desired CPU utilisation under control. Hence, the learning and testing analysis support our hypothesis that reducing on-demand cold starts can be directly linked to the throughput improvement.

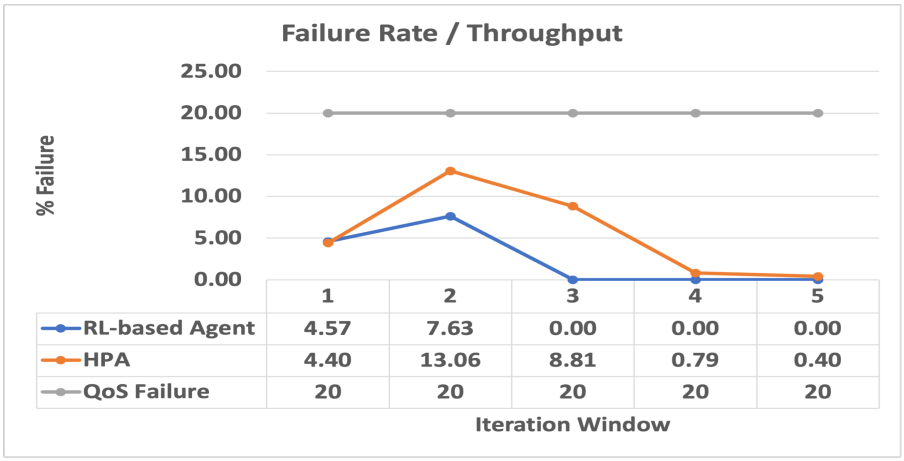



**Figure 3.8:** RL Agent v/s HPA: Failure Rate

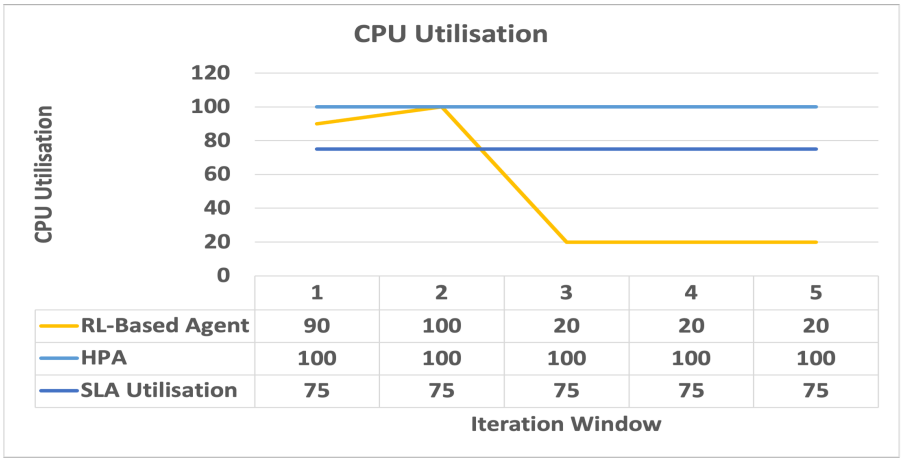



**Figure 3.9:** RL Agent v/s HPA: CPU Utilisation

## 3.6 Discussion

Function cold start is an inherent shortcoming of the serverless execution model. Thus, we have proposed an RL-based technique to investigate the demand pattern of the ap-

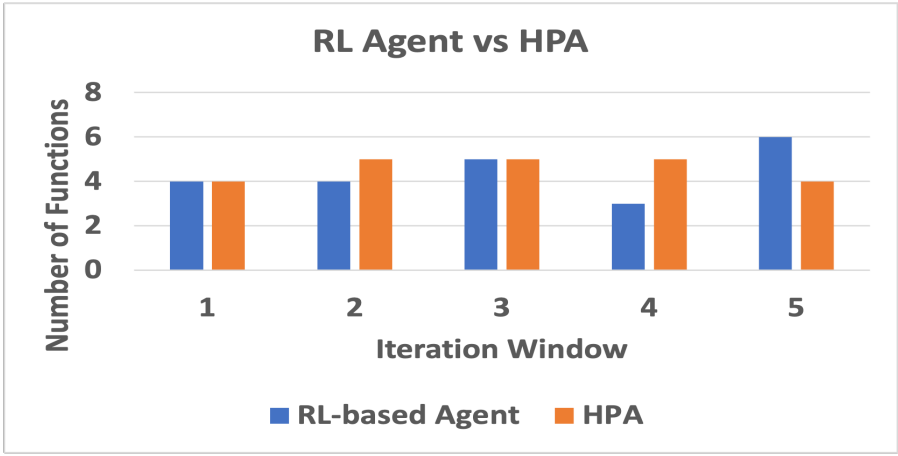



**Figure 3.10:** HPA: Function Provision

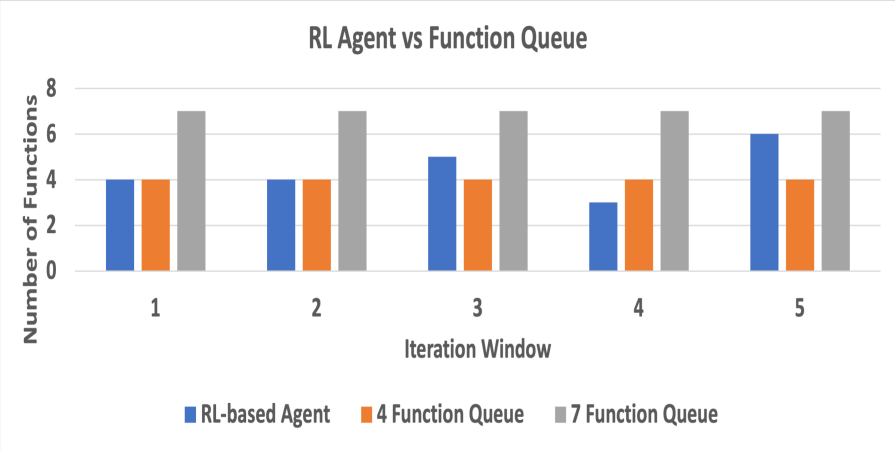
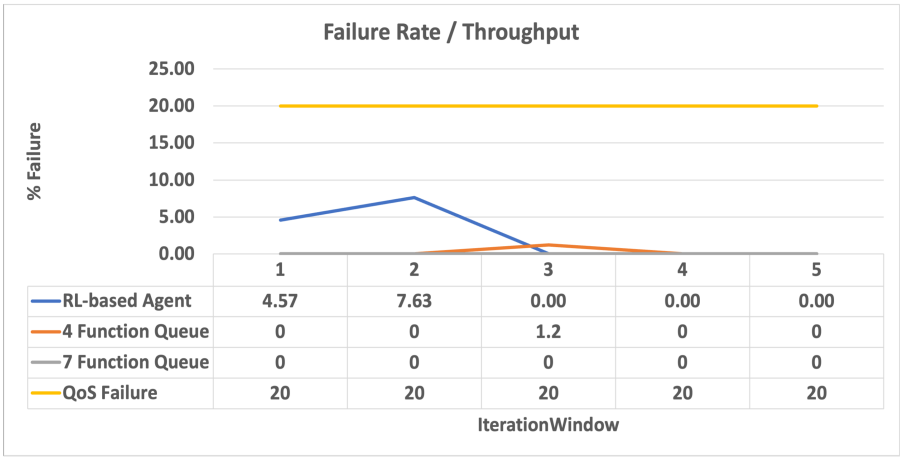



**Figure 3.11:** Function Queue: Function Provision

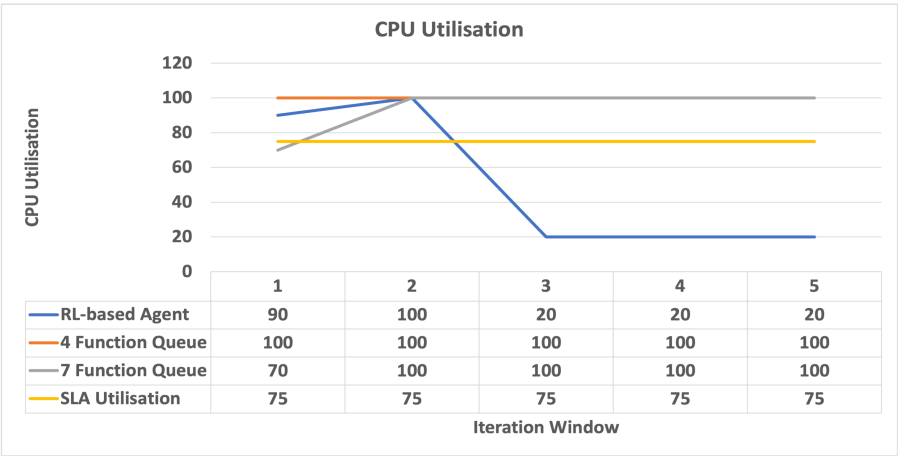
plication and attempt to reduce the frequency of function cold starts. The proposed agent performs better than the baseline approaches under a controlled experimental environment. But there are certain points to recollect associated with the real-time appropriateness of the proposed solution.

We leverage the RL environment modelling, specifically Q-Learning constraints [138], [142], and in general, these algorithms are expensive in terms of data and time. The agent interacts with the modelled environment to acquire relevant information over multiple epochs that signify a higher degree of exploration. Hence, as evidenced in this chapter, for an RL-based agent to outperform a baseline technique, a training period of 500




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| RL-based Agent | 4.57 | 7.63 | 0.00 | 0.00 | 0.00 |
| 4 Function Queue | 0 | 0 | 1.2 | 0 | 0 |
| 7 Function Queue | 0 | 0 | 0 | 0 | 0 |
| QoS Failure | 20 | 20 | 20 | 20 | 20 |

**Figure 3.12:** RL Agent v/s Function Queue: Failure Rate




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| RL-based Agent | 90 | 100 | 20 | 20 | 20 |
| 4 Function Queue | 100 | 100 | 100 | 100 | 100 |
| 7 Function Queue | 70 | 100 | 100 | 100 | 100 |
| SLA Utilisation | 75 | 75 | 75 | 75 | 75 |

**Figure 3.13:** RL Agent v/s Function Queue: CPU Utilisation

epochs is exploited for satisfactorily analysing the workload demand for a timeframe (10 minutes). Therefore, RL-based approaches are considerably expensive in practical applications with stringent optimisation requirements.

A classical Q-Learning approach is applied to discrete environment variables [138]. To constrain the serverless environment within the requirements of the Q-Learning algorithm, we consider the discrete variables to model cold starts. The size of the Q-table is large and is a function of state space and action space. But with the expansion of state space or the action space, the size of the Q-table grows exponentially [138], [142]. Therefore, Q-Learning experiences state explosion, making it infeasible to perform updates

on Q-values and degraded space and time complexity.

The proposed agent analyses individual application demand, so the learning cant be generalised for other demand patterns and requires respective training to be commissioned. Furthermore, the agent is trained for 500 iterations and evaluated, but the chance of exploring every state is bleak with limited iterations of training. Therefore, the agent expects to be guided by certain approximations to avoid acting randomly. The agent utilises resource-based metrics that affect the cold starts, so the availability of relevant tools and techniques to collect instantaneous metrics is essential. Also, the respective platform implementation of a serverless environment, such as metrics collection frequency, function concurrency policy, and request queuing, can extend support to the analyses.

The difference between the approaches can be attributed to the following characteristics of the proposed RL-based agent

1. The process of elimination of invalid states during the RL environment setup and lazy loading of Python, helps the agent to productively use the acquired information about the environment.

2. Although the RL-based agent outperforms HPA and function queue policy, there is a lack of function container concurrency policy. The CPU-intensive function workload is configured with an execution time of 60 seconds and thus affected by the concurrency control of the instance.

3. The composition of state space and reward function incorporates the effect of failures during the training, and therefore, the agent tries to compensate for the failures in consequent steps of learning by exploiting the acquired knowledge.

On the account of the performance evaluation results, we can adequately conclude that the proposed agent successfully outperforms competing policies for the given workload and experiment settings. We strengthen this claim by analysing the training and testing outcomes of the RL-based agent, focused on examining the workload pattern to reduce request failure which is a direct consequence of appropriate function instances representing reduced function cold starts.

## 3.7 Summary

FaaS model relies on preparing new execution environments/containers on demand. This initialisation process, known as function cold start, introduces a non-negligible delay in the order of milliseconds to seconds, significantly impacting application responsiveness. This chapter visited the problem of reducing frequent cold starts by analysing application demand through a RL technique. We formally modelled the environment on the Kubeless framework and utilised Q-Learning to proactively manage instance provisioning based on relevant function metrics (CPU utilisation, instance count, failure rate). The proposed RL agent was designed to examine these metrics and make guided decisions in provisioning the appropriate number of function instances. Through iterative training, we successfully verified that a strong association exists between reducing the number of cold starts and improving the platform's success rate.

Our evaluation demonstrated the efficacy of the proposed agent against baseline strategies of default HPA and function queue policies. After training, the Q-Learning agent successfully improved throughput by up to 8.81% and reduced resource wastage by up to 37%, directly attributable to the reduced cold start occurrences. While the proposed strategy alleviates the frequent cold start problem, the continuous need to match application demand and execute within strict deadlines in a highly complex and multi-tenant FaaS environment highlights the opportunity to dynamically and proactively adjust scaling capacity. The following chapter builds upon this foundation by investigating advanced Recurrent RL models for intelligent autoscaling of functions.

# Chapter 4

# Deep Recurrent RL-based Proactive Function Scaling

*This chapter addresses the challenges of achieving effective autoscaling in FaaS. While CSPs offer near-infinite function elasticity, these systems are challenged by fluctuating workloads and stringent performance constraints. A typical CSP strategy for autoscaling involves empirically determining and adjusting desired function instances based on monitoring-based thresholds (such as CPU or memory). However, this threshold-based configuration requires expert knowledge and a complete view of the environment, rendering autoscaling a performance bottleneck that lacks adaptability. Therefore, we investigate the use of RL algorithms, proven to be beneficial in analysing complex cloud environments and producing adaptable policies. We recognise the partial observability of complex cloud environments due to operational interference and limited visibility, and model the decision process as a POMDP. We specifically investigate model-free Recurrent RL agents for function autoscaling and show that recurrent policies are able to capture the environment parameters and demonstrate promising results for function autoscaling.*

## 4.1 Introduction

*Autoscaling* is the process of adding or removing function(s) from a platform, as per the demand, and has a direct correlation with platform performance. CSPs usually employ general-purpose rule-based or threshold-based horizontal scaling mechanisms or utilise a pool of minimum running function(s) [42][143] to handle function start-up delays while serving workload.

This chapter is derived from:

- **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, "A Deep Recurrent-Reinforcement Learning Method for Intelligent Autoscaling of Serverless Functions", *IEEE Transactions on Services Computing*, Volume: 17, Number: 5, Pages: 1899-1910, September, 2024.

Autoscaling provides an opportunity for CSPs to optimally utilise their resources [144] and share unused resources in a multi-tenant environment. However, configuring thresholds involves manual tuning, expert domain knowledge, and application context that reduces development flexibility and increases management overhead. Since cloud workloads are highly dynamic and complex, threshold-based autoscaling solutions lead to challenges like function cold starts and hysteresis [31], failing to offer performance guarantees. A cold start is a non-negligible function instantiation delay that is introduced before processing the request, while *hysteresis* highlights the temporal dependency of environment states on the past. Therefore, providing an adaptive, flexible, and online function autoscaling solution is an opportunity to ensure efficient resource management with performance trade-offs in serverless computing. Furthermore, autoscaling approaches employed by existing FaaS frameworks are excessively dependent on monitoring solutions. Although researchers in [32] identify metric collection for thresholds as a bottleneck for autoscaling due to significant collection delay or unreliability, a self-corrective model is demanded to account for underlying variations.

Autoscaling has been actively investigated in the cloud computing domain [145][31] [146][81][77], particularly for VMs, and has periodically highlighted the need for appropriate resource scaling to minimise operational costs and improve performance. Resource scaling is an NP-hard problem [144][77] and necessitates the realisation of complex environmental factors while balancing the system performance between QoS and SLAs. In the past, RL algorithms have been applied in the context of VM autoscaling [144][31][32][78] and have demonstrated adaptable performance over traditional methods in capturing the workload uncertainty and environment complexity. But the application of RL for function autoscaling is yet underexplored [78]. RL-based solutions are known to interact with an environment, perform an action, learn periodically through feedback, and account for the dynamics of the cloud environment.

In this chapter, we investigate the application of RNN, specifically LSTM in a model-free POMDP setting for function autoscaling. Earlier works [31][74][72][73] employing RL-based autoscaling generally model decision making as MDP and fall short to discuss partial observability in realistic environments [147][148]. Furthermore, various existing studies discussed in [31][74] experiment with RL-based solutions in a simulated FaaS en-

vironment, with the research in [32] criticising this methodology. Simulated FaaS frameworks generally sample factors such as cold start and execution time from profiled data and are insufficient to capture the variability in real environments. Therefore, we examine the integration of LSTM with PPO, a state-of-the-art RL algorithm, to analyse partial observability and sequential dependence of autoscaling actions and find a balance between conflicting CSP and user objectives. We perform experiments with *matrix multiplication* function and compare LSTM-PPO against Deep Recurrent Q-Network (DRQN) and PPO (clipped objective) to infer that in our experimental settings, recurrent policies capture the environment uncertainty better and showcase promising performance in comparison to PPO and commercially adopted threshold-based approaches. We make use of OpenAI Stable Baseline's[149] standard implementation of the LSTM-PPO and PPO algorithms, and implement our compatible OpenFaaS serverless environment following Gymnasium [150] guidelines.

In summary, the key contributions of this chapter are:

1. We analyse the characteristics of FaaS environments to identify and model autoscaling decisions as a POMDP. We further hypothesise that scaling decisions have a sequential dependence on interaction history. We propose a POMDP model that captures function metrics such as CPU and memory utilisation, function replicas, average execution time and throughput ratio, as partial observations and formulate the scaling problem.

2. We investigate how function autoscaling works, highlight the differences between contrasting approaches and investigate a Deep Recurrent RL (LSTM-PPO) autoscaling solution to capture the temporal dependency of scaling actions and workload complexity. We deploy the proposed agent to the OpenFaaS framework and utilise open-source function invocation traces [129] from a production environment to perform experiments with a matrix multiplication function.

3. We implement a Gymnasium [150] compatible OpenFaaS serverless environment to be integrated directly with the proposed RL agent.

4. We perform our experiments on Melbourne Research Cloud (MRC) and evaluate the proposed LSTM-PPO approach against the state-of-the-art PPO algorithm,

**Table 4.1:** A Summary of Related Works and Their Comparison with Our Proposed Method.
H: Horizontal Scaling, V: Vertical Scaling

| Work | Type | Scaling | Technique | Objective | Environment |
|---|---|---|---|---|---|
| [151] | FaaS | H | Threshold-Based | CPU Utilisation | AWS Lambda |
| [152] | FaaS | H | Threshold-Based | CPU Utilisation | Google Cloud Functions |
| [77] | Microservices | H,V,Brownout | GRU + Q-Learning | QoS | Testbed |
| [78] | FaaS | H | Q-Learning | QoS | OpenFaaS |
| [74] | FaaS | H | Q-Learning, DQN, DynaQ+ | QoS + Budget | Simulation, Kubeless |
| [72] | FaaS | H | Q-Learning | QoS | Kubeless |
| [73] | FaaS | H | Q-Learning | QoS | Knative |
| [75] | FaaS | H | Bi-LSTM | Resource | Knative |
| [76] | FaaS | H, V | Q-Learning | QoS + Resource | Testbed |
| [79] | FaaS | H | Kneedle Algorithm | QoS + Budget | OpenFaaS |
| Our Method | FaaS | H | LSTM - PPO | QoS + Resource | OpenFaaS |

commercially offered threshold-based horizontal scaling, OpenFaaS' request-per-second scaling policy, and DRQN, to demonstrate LSTM-PPO's ability to capture environment uncertainty for efficient scaling of serverless functions.

The rest of the chapter is organised as follows. Section 4.2 highlights related research studies. In Section 4.3, we present the system architecture and formulate the problem statement. Section 4.4 outlines the proposed agents workflow and describes the implementation hypothesis and assumptions. In Section 4.5, we evaluate our technique with the baseline approaches and highlight training results and discuss performance. Section 4.6 concludes the chapter with summary.

## 4.2 Related work

In this section, we summarise (see Table 4.1) existing work on autoscaling in FaaS, and the application of RL in FaaS. We compare existing work based on their key features and provide a detailed background on the Deep Recurrent RL (RPPO) algorithm used in designing our autoscaling policy.

### 4.2.1 AutoScaling in Function-as-a-Service

Resource elasticity, analogously used with autoscaling, is a vital proposition of cloud computing that enables large-scale execution of a variety of applications. A recent survey [144] discusses the relevance of cloud resource elasticity for the IaaS model to express that autoscaling and pay-as-you-go billing enables infrastructure adjustments based on workload variation while complying with SLAs. On this basis, the study identifies that autoscaling addresses a set of associated challenges, namely, scaling and scheduling which are generally NP-hard problems. Additionally, the research explores the possibility of RL algorithms for autoscaling to approach the complexity and variability of cloud environments and workloads. It is emphasised that utilisation of such RL algorithms for scaling purposes can help the service providers to come up with a more transparent, dynamic, and adaptable policy.

Straesser et al. [32] conduct experiments related to cloud autoscaling and assert autoscaling to be an important aspect of computing for its effects on operational costs and QoS. The authors define scaling as a task of dynamically provisioning resources under a varying load and necessitates the automation of processes for highly complex cloud workloads. They discuss that commercial solutions usually operate with user-defined rules and threshold heuristics, and state that an optimal autoscaler is expected to minimise operational cost and SLA violations.

In addition to workload variability, QoS sensitivity is also identified as an enabler for increased operational costs and resource wastage. A microservices-focused autoscaling scheme is introduced in [77] where a trade-off between horizontal, vertical, and a self-adaptable brownout technique is determined based on the infrastructure and workload conditions. The researchers exploit GRUs for workload prediction and utilise Q-learning for making trade-off updates and scaling decisions. The study asserts that workload prediction is an important factor for autoscaling and acknowledges resource allocation to be an NP-hard problem with multi-dimensional objectives of QoS and SLAs.

In the context of FaaS autoscaling, work in [73] experiments with the concurrency-level setting of Knative, a Kubernetes-based serverless framework, and identify that function concurrency settings have varying effects on latency and throughput of function. Therefore, they utilise the Q-learning algorithm to configure functions with op-

timal concurrency levels to further improve performance. Another work [72] presents preliminary results of applying Q-learning to FaaS for predicting the optimal number of function instances to reduce the cold start problem. They utilise the function resource metrics and performance metrics and apply them to discrete state and action spaces for adding or removing the function replicas, with threshold-based rewards, to eventually improve function throughput.

Similarly, studies like [74][75][76] emphasise addressing the dynamicity, agility, and performance guarantees of FaaS by employing RL-based autoscaling solutions. The work in [74] follows a monitoring-based scaling pattern and explores algorithms like Q-learning, DynaQ+, and Deep Q-Learning, partially in simulation and practical settings, to reasonably utilise resources and balance between budget and QoS. They aid the agent's training process by sampling simulation data based on probability distribution and running parallel agents to speed up the learning process. The work in [75], discusses the concurrency level in the Knative framework and asserts that identifying appropriate thresholds is challenging, requires expert knowledge, and has varying effects on performance. Therefore, to efficiently use the function resources and improve performance, authors profile different concurrency levels for best performance and propose an adaptive, Bi-LSTM model for workload prediction and determine the number of function replicas using identified concurrency levels. Another study [76] focuses on function response time and states that threshold-based scaling cannot devise a balance between resource efficiency and QoS. Therefore, the authors explore Q-learning to propose adaptive horizontal and vertical scaling techniques by profiling different resource allocation schemes and their corresponding performance. Their proposed state space considers resource requests and limits, along with the availability of GPU components, to model rewards as the divergence from agreed SLO levels. Taking a different approach, the researchers in [78] utilise Q-learning in the context of Kubernetes-based serverless frameworks and propose a resource-based scaling mechanism to adjust function CPU utilisation threshold to reduce response time SLA violations. Taking a different approach, [79] proposes an online application profiling technique that identifies a knee point and adjusts resources until the point those changes reflect in performance gain using the Kneedle algorithm in conjunction with binary search. Further, a survey [31]

summarises autoscaling techniques for serverless computing under different categories like rule-based, AI-based, analytical model, control theory-based, application profiling, and hybrid technique and envisions new directions like energy-driven and anomaly-aware serverless autoscaling.

These proposals are complementary yet contrasting to each other either in optimisation objectives, profiled metrics, or scaling policy. Some fail to address the performance dependency on complex workloads, while few rely on pre-configured thresholds [151][152] that require expert knowledge and application insights. Few studies focusing on workload prediction assume a fully observable environment and miss out on the temporal dependency of environment states where scaling decisions have been taken. Contradictory to these proposals, we examine a Deep Recurrent RL-based autoscaling solution, particularly LSTM-PPO, to hypothesise that FaaS environments are highly dynamic, partially observable with complex workloads, and that scaling decisions are influenced by environment uncertainty. We model function autoscaling as a POMDP and utilise monitoring metrics like average CPU and memory utilisation, function resource requests, average execution time, and throughput ratio to discover an optimal scaling policy. Our proposed RL-based autoscaling agent interacts with the FaaS environment, waits for a sampling period [32] to receive delayed rewards, and feeds the observed environment state to the recurrent actor-critic model. Although a few studies [77][75] have utilised recurrent networks like LSTM or GRU for workload prediction in serverless context but do not address the temporal relationship between scaling actions and their effect on environment state. Further, we take inspiration from [148][153][154] where recurrent models have been utilised to analyse the inter-dependence of environment states and retain useful information to learn optimal policies.

## 4.3 System Architecture and Problem Formulation

### 4.3.1 System Architecture

The main components of our autoscaling solution are the Prometheus [62] monitoring service and the Deep Recurrent RL-agent (DRL-agent), which are shown in Fig. 4.1. For

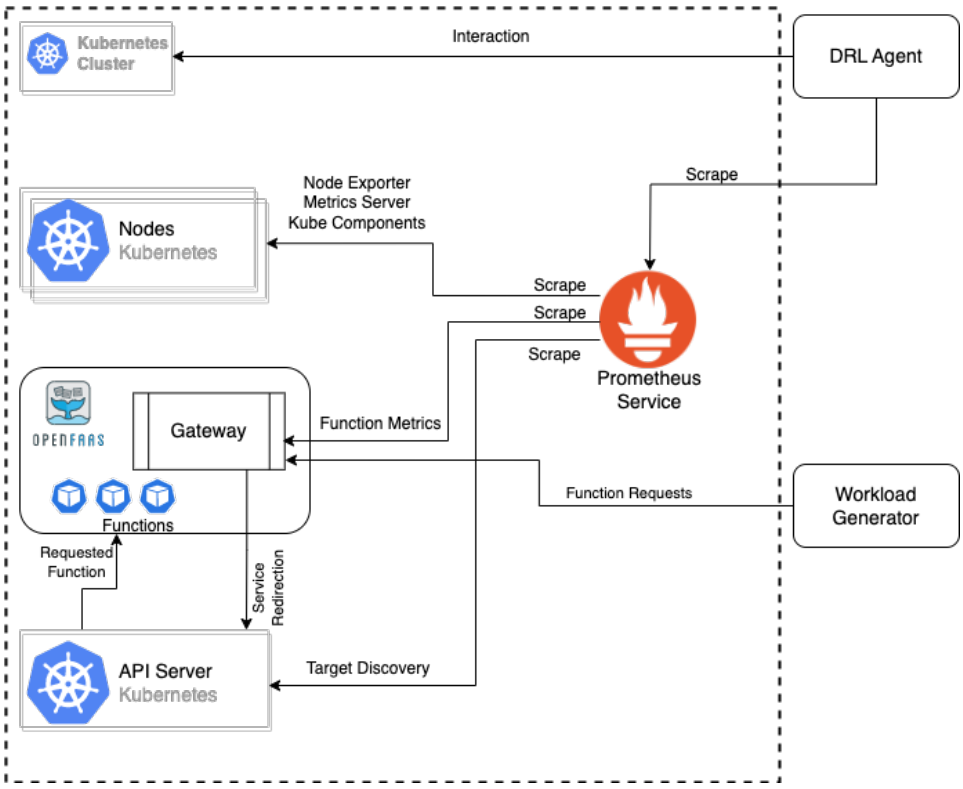



**Figure 4.1:** System Architecture

the serverless environment, we deploy OpenFaaS [9], a Kubernetes-based FaaS framework, over a multi-node MicroK8s [155] cluster, a production Kubernetes distribution. OpenFaaS includes a Gateway deployment to expose function performance metrics and Prometheus is configured to periodically scrape function metrics such as execution time, replica count, and throughput ratio. OpenFaaS also packs an *alertmanager* that periodically watches for pre-configured request-per-second scaling threshold to provide horizontal scaling capabilities. The monitoring service further scrapes resource metrics from the Kubernetes API Server, Kubelet, and Node exporters that are utilised by our DRL agent for observation collection at every sampling window. The DRL agent utilises the standard Stable Baseline3 (SB3) [149] implementation of LSTM-PPO [1] and models the FaaS environment following Gymnasium [150] guidelines, for the POMDP model to be directly used by SB3 algorithms. Additionally, we implement our own version of DRQN [1] using PyTorch [156][157] for evaluation. We also deploy an HTTP-request generator

[1] `https://github.com/Cloudslab/DRe-SCale`

tool to simulate online user behaviour to train and evaluate our DRL autoscaling agent.

### 4.3.2 Problem Formulation

Existing FaaS platforms generally exercise threshold-based scaling when a monitored metric exceeds the configured maximum or minimum. Autoscaling of resources is considered a classic automatic control problem and commonly abstracted as a MAPE (map-analyse-plan-execute) control loop [145]. At every sampling interval, the monitoring control loop collects the relevant metrics and may decide to scale based on the analysed observation. Autoscaling is a sequential process with non-deterministic results in a partially observable environment that is conditioned on historical interactions, therefore, we design FaaS autoscaling as a *model-free POMDP*. POMDPs are a mathematical model and an extension of MDP that account for uncertainty while maximising a given objective.

#### Model-Free POMDP

In a real-world scenario, it is hard to perceive the complete state of the surrounding environment and a MDP rarely holds true [147]. Instead, a POMDP better encapsulates environmental characteristics from incomplete or partial information about said environment. Formally, a POMDP model is defined as a 6-tuple $(S, A, O, T, Z, R)$ where: $S$ denotes the set of all possible environment states, $A$ denotes the set of all actions, $O$ denotes the set of all observations that an agent can perceive, $T$ and $Z$ represent the transition probability function and observation probability function, respectively, and $R$ denotes the reward function. Conceptually, the agent observes itself in some environment state $s_t$, hidden due to partial observability at each sampling interval $t$ and maintains a belief $b_t$, an estimate of its current state, to select an action $a_t$ and transition to a new state $\hat{s}_t$. The agent perceives the state information through observation $o_t$ and utilises the transition and observation probability function to update the state estimates. After transitioning to a new state $\hat{s}_t$, the agent receives reward $r_t$ that helps in maximising the objective.

Since probability functions are difficult to model in complex FaaS environments and

states cannot be perfectly represented to capture the estimates of belief or hidden states [148], we define the autoscaling problem as model-free POMDP. Model-free POMDP attempts to maximise the cumulative reward without explicitly modelling the transition or observation probabilities. Further, it needs function approximation techniques like neural networks, specifically RNN, to capture the uncertainty and temporal dependency. Therefore, we define the POMDP observations as a tuple of $(O, A, R)$ and utilise recurrency to model and infer transition probabilities, observation probabilities and hidden states to fulfil the conflicting objectives of resource utilisation, operational cost and QoS objectives.

**Deep Recurrent-Reinforcement Learning**

A possible solution to learning effective policies in a model-free POMDP is the application of model-free RL algorithms. Here, the agent directly interacts with the environment and does not explicitly model the transition or observation probabilities. Vanilla RL algorithms like Q-learning and Deep Q-Networks (DQN) have no mechanism to determine underlying state [148] and speculates that fed observation is a complete representation of the environment. To capture sequential or temporal dependencies, often recurrent units are integrated with vanilla RL approaches, known as Recurrent Reinforcement Learning (RRL) [158]. Prior studies [148], [153] [154],[159] [160], have introduced and applied RRL approaches to a variety of application domains such as T-maze task, financial trading, network resource allocation and Atari games, to address sequential nature and partial observability of environment, i.e., a non-Markovian or POMDP setting. In RRL, an agent follows the basic principle of performing an action in the environment, establishing its state and receiving feedback to improve the policy, but, additionally employs RNN units/cells to model uncertainty. Theoretically, POMDP has an underlying dynamics of MDP with an additional constraint of state uncertainty or observability that makes the process non-Markovian. Therefore, we define the core RL components as observation $O$, action $A$, reward $R$ (guiding signals) and FaaS environment.

We model the observation space as $o_t = (\tau_t, \phi_t, q_t, n_t, c_t, m_t) \in O$ where $\tau_t$ is aver-

**Table 4.2:** Notations

| Symbol | Definition |
|---|---|
| $f_l$ | Function for training {matmul}. |
| $S$ | State space for POMDP agent. |
| $A$ | Action space for POMDP agent. |
| $O$ | Observation space for POMDP agent. |
| $T, Z$ | Transition and Observation probability functions. |
| $R$ | Reward function for POMDP agent. |
| $N$ | Maximum number of function replicas possible (function quota). |
| $n_{min}$ | Minimum number of function replicas. |
| $Q$ | Maximum requests possible in a sampling window. |
| $t$ | sampling window. |
| $n_t$ | All available functions during $t$. |
| $\tau_t$ | Average execution time of $n_t$ functions. |
| $c$ | Average CPU utilisation of $n_t$ functions. |
| $c_{max}$ | Maximum CPU utilisation of a function. |
| $m_t$ | Average memory utilisation of $n_t$ functions. |
| $m_{max}$ | Maximum memory utilisation of a function. |
| $\phi_t$ | Throughput of function. |
| $q_t$ | Requests during $t$. |
| $k$ | Scaling limits. |
| $s_t$ | Environment state at $t$. |
| $b_t$ | Belief state for POMDP agent. |
| $o_t$ | Environment observation tuple $(\tau_t, \phi_t, q_t, n_t, c_t, m_t) \in O$ at $t$. |
| $a_t$ | Agent action $\in \{-k, \ldots, +k\}$ at $t$. |
| $r_t$ | Reward for action $a_t \in R$ at $t$. |
| $r_{min}$ | Negative immediate reward (-100). |
| $\alpha, \beta, \gamma$ | Objective weight parameters. |

age execution time of $n_t$ available function replicas with $c_t$ average CPU and $m_t$ average memory utilisation, while successfully serving $\phi_t$ proportion of $q_t$ requests in the sampling window $t$. The agent adjusts the number of function instances in the upcoming

sampling window $t+1$ using suitable actions in an attempt to maximise the reward. Therefore, we define scaling action $a_t$ as the number of function instances, $k$, to add or remove and represent it as $a_t \in A = \{-k, \cdots + k\}$ such that $n_{min} \leq (n_{t-1} + a_t) \leq N$, where $N$ is function quota. This estimate helps the agent to control the degree of exploration by maintaining replication within quota $N$.

The objective of the DRL agent is to learn an optimal scaling policy, and therefore, we structure the rewards $r_t \in R$ over monitored metrics - $c_t$ average CPU utilisation, $m_t$ average memory utilisation, $\phi_t$ successful proportion of total requests and number of available function replicas $n_t$. Our proposed agent does not work towards achieving a specific threshold. Instead, it learns to maximise the returns, i.e., improve resource utilisation, throughput and economically scaling function replicas. After performing an action $a_t$, the agent receives a delayed reward $r_t$ at every sampling window $t$ and updates its network parameters.

RL application for model-free POMDP does not explicitly estimate the probabilities, instead, RNNs are incorporated to analyse environment uncertainties and model time-varying patterns [158][160]. The structure of RNNs is made-up of highly-dimensional hidden states that act as network memory and enables it to remember complex sequential data. These networks map an input sequence to output and consist of three units - input, recurrent and output unit, serving towards memory goal.

## 4.4 LSTM-PPO based AutoScaling Approach

As discussed in section **??**, we introduce recurrency to handle system dynamics, complex workloads, and hidden correlation of components based on POMDP model in autoscaling tasks. We select PPO, a popular state-of-the-art on-policy RL algorithm for autoscaling agents. While model-free off-policy algorithms such as DQN, Deep Deterministic Policy Gradient (DDPG) have been studied with recurrent units [148] [154], we explore a model-free on-policy PPO in our setting due to its ease of implementation, greater stability during learning, better performance across different environments [161] and support for discrete actions while providing better convergence [149]. Although on-policy methods are known to be sample inefficient and computationally expensive,

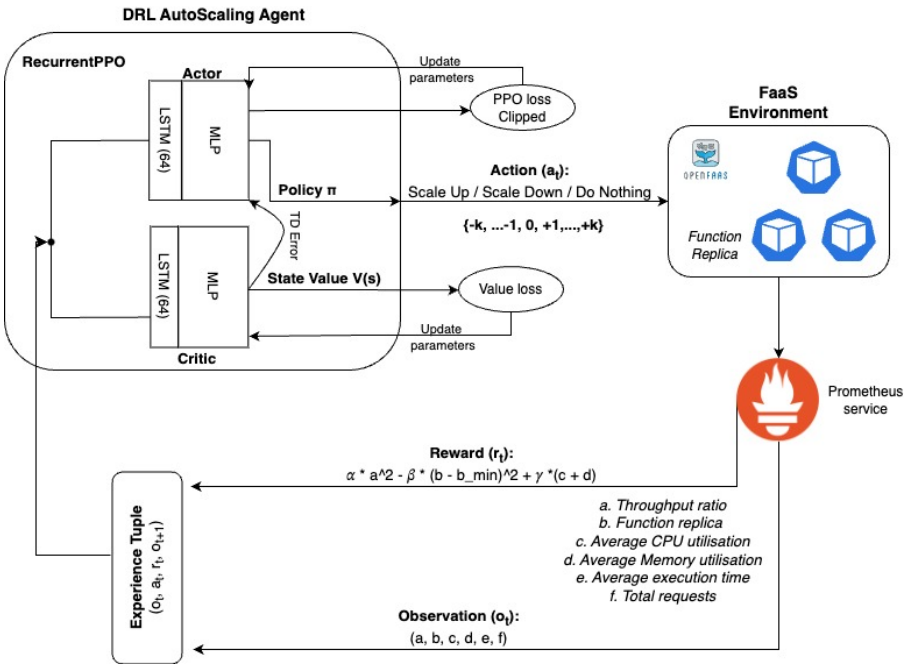



**Figure 4.2:** DRL agent structure for Autoscaling

our agent continuously collects samples for timely policy updates. Also, off-policy algorithms tend to be harder to tune than on-policy because of significant bias from old data and Schulman at el. [162] suggests that PPO is less sensitive to hyperparameters than other algorithms. PPO has found its application in domains like robotics, finance and autonomous vehicles, and takes advantage of the Actor-Critic method to learn optimal policy estimations. However, for partial observability or temporal dependence, general RL algorithms struggle to capture underlying correlations and patterns effectively. Therefore, we utilise RNN units, specifically LSTM, to address partial observability in the FaaS environment and improve the agent's decision-making capabilities. This integration is expected to enhance PPO's ability to capture historical data and make informed decisions while improving its policy via new and previous experiences.

The core component of the proposed autoscaling solution is the integration of recurrent units with a fully-connected multi-layer perceptron (MLP) that takes into environment observation and maintains a hidden internal state to retain relevant information. The LSTM layer is incorporated into both actor and critic networks to retain information i.e., the output of the LSTM layer is fed into fully-connected MLP layers, where the ac-

tor (policy network) is responsible for learning an action selection policy and the critic network serves as a guiding measure to improve actor's decision. The network parameters are updated as per PPO clipped surrogate objective function [163] (Eq. 4.1) which helps the agent balance its degree of exploration and knowledge exploitation. It further improves network sample efficiency and conserves large policy updates.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(\hat{r}_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{4.1}$$

$$\hat{r}_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|_t)} \tag{4.2}$$

$$r_t = \begin{cases} \alpha.\phi_t^2 - \beta.(n_t - n_{min})^2 + \gamma.(c_t + m_t) & ;1 \leq a_t + n_{t-1} < N \\ r_{min} & ;\text{otherwise} \end{cases} \tag{4.3}$$

The proposed autoscaling technique has two phases: an agent training phase and a testing phase. Fig. 4.2 demonstrates the agent training workflow. The environment setup process precedes the agent training, where the agent interacts with the environment and obtains information. After initial setup, the agent is trained for multiple episodes of sampling windows, where it assesses the function demand $q_t$ over individual sampling window $t$ and ascertains appropriate scaling action. During a sampling window $t$, the agent collects the environment observation $o_t$ and samples an action $a_t$ according to LSTM-PPO policy. If the agent performs an invalid action, it is awarded an immediate negative reward $r_{min}$, else the agent obtains a delayed reward $r_t$ (Eq. 4.3), for sampling window $t$, calculated using the relevant monitored metrics (**??**). This reward helps the agent in action quality assessment, transition to a new state and has significant effects on the function's performance. These rewards are essential for improving the agent's decision-making capability. The critic network estimates the agent state and helps update the network parameters. The agent continues to analyse the demand over multiple sampling windows, repeating the interaction process and accumulating the relevant information in recurrent cells for learning. Once the agent is trained for sufficient episodes and rewards appear to converge, we evaluate the agent in the testing phase.

In the testing phase, the agent is evaluated for its learnt policies. It collects current

environment observation, samples the action through actor policy and scales the functions accordingly. We hypothesised the relationship between QoS and resource utilisation and deduce that appropriately scaling the functions improve throughput, resource utilisation and reduce operational costs (number of function replicas used).

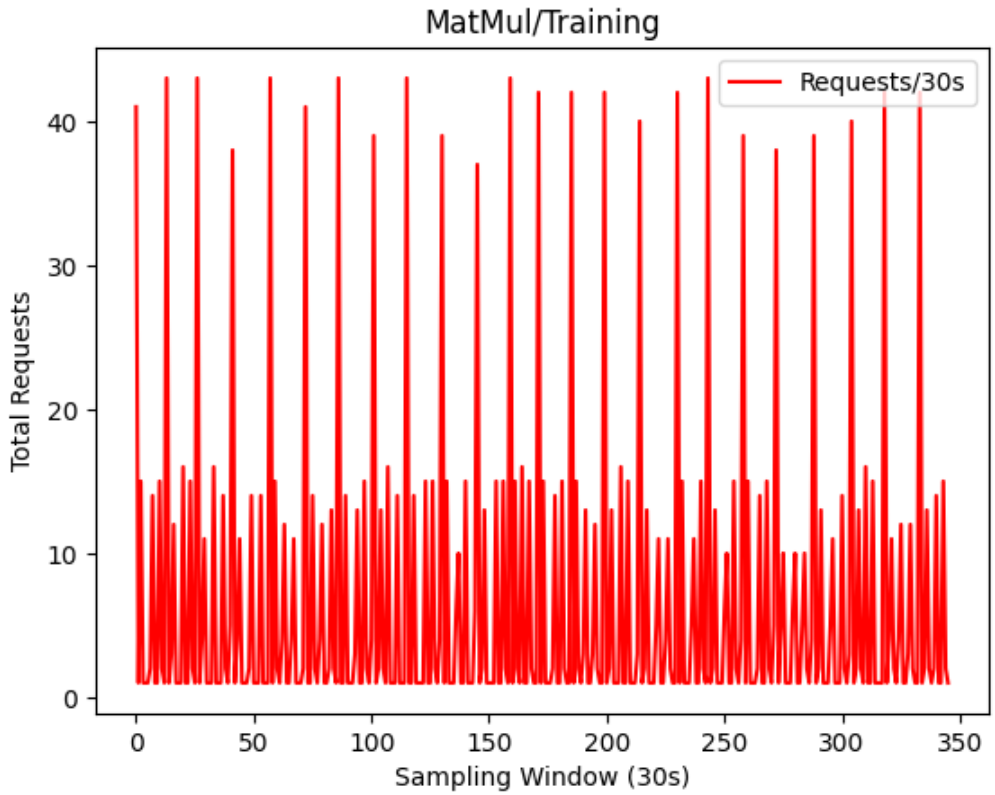



**Figure 4.3:** Workload for Matrix Multiplication function

## 4.5 Performance Evaluation

In this section, we provide the experimental setup and parameters, and perform an analysis of our agent compared to other complementary solutions.

### 4.5.1 System Setup

We set up our experimental multi-node cluster, as discussed in Section **??**, using NeCTAR services on the MRC. It includes a combination of 2 nodes with 12/48, 1 node with 16/64, 1 node with 8/32 and 1 node with 4/16 vCPU/GB-RAM configurations. We deploy OpenFaaS [9] along with Prometheus [62] service on MicroK8s (v1.27.2), however, we used Gateway v0.26.3 due to scaling limitations in the latest version and remove its alert manager component to disable rps-based scaling. The system setup parameters are listed in Table 4.3.

**Table 4.3:** Parameters for System Setup

| **Parameter Name** | ***Value*** |
|---|---|
| MicroK8s version | v1.27.2 |
| OpenFaaS Gateway version | v0.26.3 |
| Nodes | 5 |
| OS | Ubuntu 18.04 LTS |
| vCPU | 4,8,12,16 |
| RAM | 16,32,64,48 GB |
| Workload | Matrix Multiplication ($m \times m$) |
| m | 10(small), 100(medium), 1000(large) |
| CPU, memory, timeout | 150 millicore, 256 MB, 10 seconds |

As FaaS is beneficial for short-running, single-purpose functions that require few resources, we consider *matrix multiplication* function with three different input sizes $small, medium, large$-$(10, 100, 1000)$ and configure it with 150/256 millicore/MB resources approximately as AWS Lambda offering and a maximum timeout of 10 seconds. Additionally, we generate the user workload using the Hey [164] load generator tool, a lightweight load generator written in Go language. For the workload we leverage an open-sourced, 14-day function trace [129] by Azure functions, Fig. 4.3, that largely represents an invocation behaviour of a production-ready application function running on a serverless platform. Although it appears stationary due to its repetitive nature, it is

representative of real cloud invocation patterns with relevant variations for scaling decisions. Since the Poisson distribution has been shown to approximately sample online user behaviour, request inter-arrival times are sampled from it. Prometheus service is configured with relevant discovery and target points to regularly scrape metrics from OpenFaaS gateway, function instances and Kubernetes API server.

As discussed in Sec. **??**, the agent assesses the function demand during a sampling window of 30 seconds for a single episode of 5 minutes. Based on the deployed infrastructure capacity, we fix the maximum function instances as 24 in isolation, to reduce the performance interference. Since frequent scaling can result in resource thrashing, we explore scaling actions within a range of 2 instances, i.e., $a_t \in \{-2, -1, 0, 1, 2\}$, avoiding resource wastage during acquisition and release of function instances. Further, the observation space is composed of the throughput ratio $\phi_t \in [0, 100]\%$, number of function instances $n_t \in [1, 24]$ and resource utilisation (CPU, $c_t$ and memory, $m_t$) $\in [0, 2] * 100\%$ that contributes towards over-burdened CPU and out-of-memory scenarios. The LSTM-based PPO agent takes advantage of a single LSTM layer of 256 units and is integrated with both Actor and Critic networks with identical network architectures having 2 fully connected MLP layers of 64 neurons each, i.e., in[64,64] and out[64,64].

### 4.5.2 Experiments

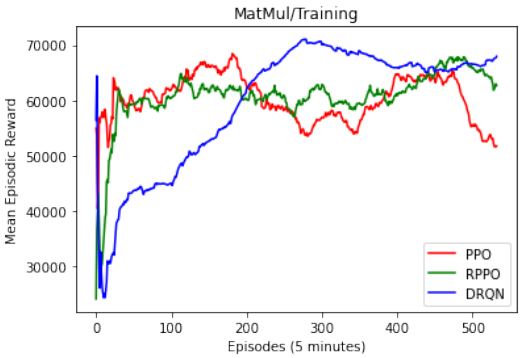
Function autoscaling is a continuous and non-episodic process, however, we set an episode based on the default scaling window of 5 minutes by Kubernetes' horizontal scaling mechanism. To demonstrate the effectiveness of recurrency in autoscaling tasks, we chose a workload with varying resource requirements at different sampling windows. After careful consideration of network parameters and sensitivity analysis, listed in Table 4.4, the DRL agent is trained for more than 500 episodes to determine a scaling policy to maximise the throughput while using minimal resources. The agent is expected to retain workload information and perform in accordance with the received feedback. Further, the agent is evaluated against a state-of-the-art, PPO-based autoscaling agent, with the same the Actor/Critic network architecture, (Table 4.4) as the RPPO agent, i.e., having 2 MLP layers with 64 neurons each. In addition to it, we evaluate a DRQN agent

**Table 4.4:** RL Environment and Network Parameters

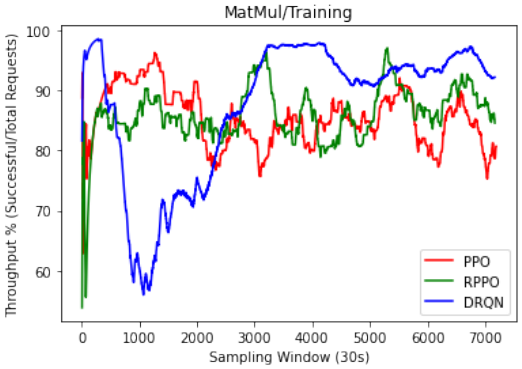
| **Parameter** | ***Value*** |
|---|---|
| $N$ | 24 |
| $t$ | 30 seconds |
| $\tau_t$ | (0 - 10) seconds |
| $\phi_t$ | (0 - 100) % |
| $q_t$ | {0,. . . ,Q} requests |
| $n_t$ | {1,. . . ,$N$} functions |
| $c_t$ | (0 - 2) *100% |
| $m_t$ | (0 - 2) *100% |
| $a_t$ | {-2, -1, 0, +1, +2 } |
| LSTM Network | Layer 1(256 cells) |
| Actor Network | Layer 1(64 cells), Layer 2(64 cells) |
| Critic Network | Layer 1(64 cells), Layer 2(64 cells) |

that integrates a LSTM layer (256 cells) with regular off-policy DQN, and has 2 MLP layers with 128 neurons, each for target network and q-network. Fig. 4.4 shows the training results of these competing approaches in terms of mean episodic rewards. The rewards are given as per Eq. 4.3, and it is evident that the mean episodic reward for PPO (60190) begins to diminish after 400 episodes as compared to LSTM-PPO(RPPO) (60540) agent. Additionally, a similar pattern is visible for the throughput of RPPO and PPO approaches, Fig. 4.4(b) where PPO struggles to keep a higher success rate by provisioning more functions. Also, we observe that mean episodic reward for the DRQN (59564) approaches that of RPPO while exploring the search space and gradually serves more workload successfully, Fig. 4.4(e), but closely tailing the trend of other approaches, Fig. 4.4(b). As mentioned in section **??**, matrix multiplication is performed for three input sizes - *small*, *medium*, *large* and similar input randomness is followed for competing approaches that are evident in execution time ( 3.7 and 4 seconds) of successful requests in Fig. 4.4(c), (d) and (e).

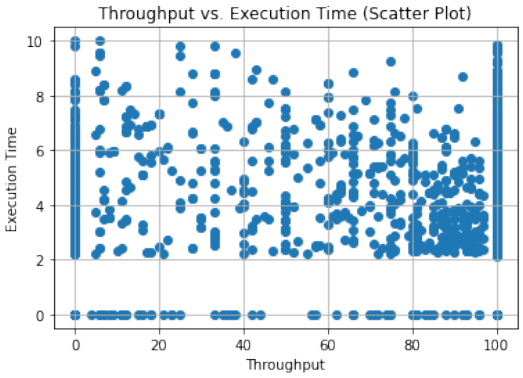
We evaluate the agents for 200 sampling windows and present the results in Fig. 4.5.



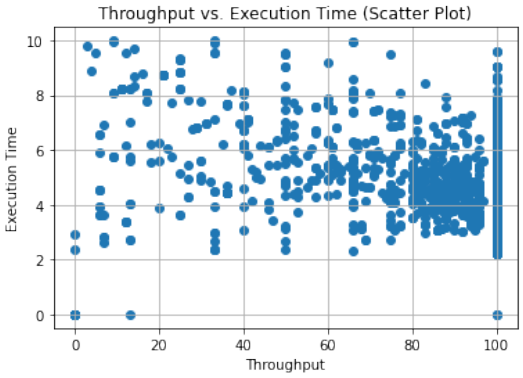

(a) Mean episodic reward



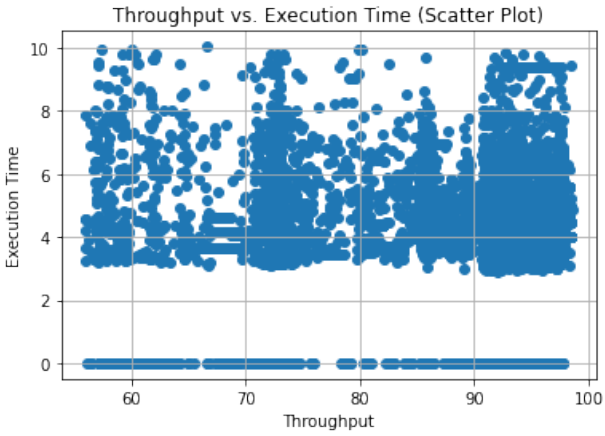

(b) Training throughput




(c) PPO throughput vs execution time




(d) RPPO throughput vs execution time




(e) DRQN throughput vs execution time

**Figure 4.4:** Training and execution performance comparison for PPO, RPPO, and DRQN: (a) Mean episodic reward, (b) Training throughput, (c) PPO throughput vs execution time, (d) RPPO throughput vs execution time, (e) DRQN throughput vs execution time.

Out of the 200 sampling windows, RPPO based autoscaling agent performed 18% better in terms of throughput, while having an average of 85% mean success ratio as compared to 67% of the PPO agent. On the other hand, the DRQN agent fell short to serve 22% of the workload with a mean success rate of 66% as compared to RPPO agent. In serving the evaluated workload, the RPPO agent utilised at least 8.4% more resources than the PPO agent and improved average execution time (seconds) by 13%, while it utilised at least 8% more resources than DRQN and slightly improved the average execution time (seconds) by 2.6%. Although the DRQN agent tries to capture sequential dependency of the workload, we suspect it fails to explore the search space and only exploits minimal replica count. Hence, as evident in Fig. 4.5(d), the DRQN agent keeps utilising lesser function resources. This agent behaviour is in-line with training results where it could serve better with less requests, thus receiving higher reward for that sampling window and eventually, accumulating higher episodic reward and throughput percentage.

We also assess the effectiveness of our approach against a default commercial scaling policy, CPU threshold-based horizontal scaling. Kubernetes-based serverless platforms like OpenFaaS [9] and Kubeless [128] can leverage underlying resource-based scaling, known as HPA implemented as a control loop that checks for target metrics to adjust the function replicas. HPA has a pre-configured query period of 15 seconds to control deployment based on target metrics like average CPU utilisation. Therefore, the HPA controller fetches the specific metrics from the underlying API and empirically calculates the number of desired functions. However, the controller is unaware of workload demand and only scales after a 15-second metric collection window. The expected threshold for function average CPU utilisation is set to be 75% with maximum scaling up to 24 instances. Therefore, whenever the average CPU utilisation of a function exceeds the fixed threshold, new function instances are provisioned. Also, HPA has a 5-minute down-scaling window during which resources are bound to function irrespective of incoming demand, representing potential resource wastage. Similarly, we compare our scaling methods with another metric-based autoscaling supported by OpenFaaS based on request-per-second processed. It is also implemented as a control loop and watches for processed requests per second (rps) and raises an alert if rps is above 5 for 10 seconds (default). Therefore, it is worthwhile to analyse the performance of the DRL-based

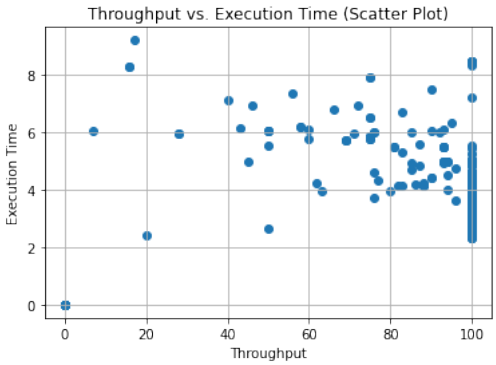
agent against HPA that reserves enough resources for either idle time or low resource utilisation.

The results for both threshold-based scaling are presented in Fig. 4.6, and both approaches struggle to keep up with the incoming workload. The rps could only manage to serve 50% of incoming load at any sampling window while only using a single instance. This happens as a single request takes approximately 4 seconds to process, and rps never goes beyond the set threshold, failing the majority of requests. On the other hand, HPA could serve 80% of incoming load on average, but fluctuates due to its set cooldown period. Although HPA tries to scale its resources to 5 replicas, its performance is degraded by 35% against RPPO and similarly, rps degrades throughput performance by 58%.

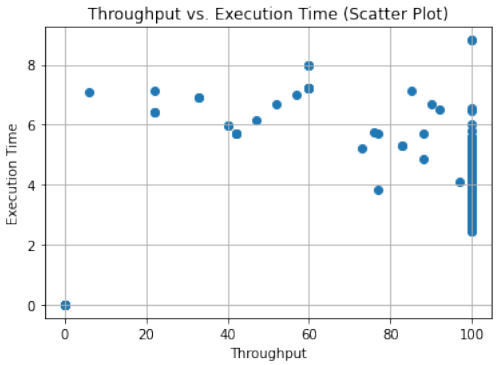
### 4.5.3 Discussion

Autoscaling is an essential feature of cloud computing and has been identified as a potential research gap in serverless computing models. As compared to service-oriented architectures where the services are always running, FaaS functions run for shorter duration and release resources, if unwanted. Hence, an adaptive scaling solution is critical in handling complex workloads for these small and ephemeral functions. Thus, we investigate a DRL-based autoscaling agent, LSTM-PPO, to work in complex FaaS settings and utilise relevant environmental information to learn optimal scaling policies. We train and evaluate the proposed solution against a state-of-the-art on-policy PPO approach, alongside commercial default, and infer that LSTM-PPO is able to capture environment dynamics better and shows promising results. Although, we argue that real-time systems are hard to model and transparent to a certain degree and that RL approaches can analyse these uncertainties better. There are certain points to remember associated with the appropriateness and application of RL methods to real systems.

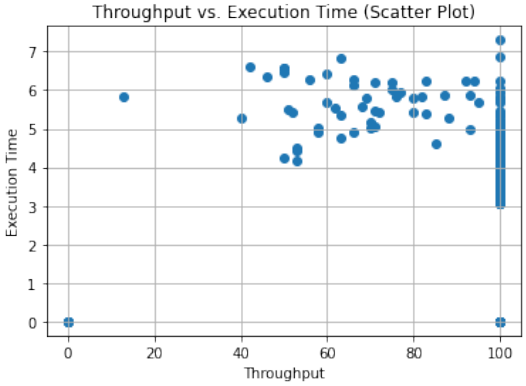
We model function autoscaling in FaaS as a model-free POMDP and leverage monitoring tools, like Prometheus, to collect the function-related metrics and apply model-free RL methods to learn the scaling policy. In general, RL algorithms are expensive in terms of data, resource and time, where an agent interacts with the modelled envi-



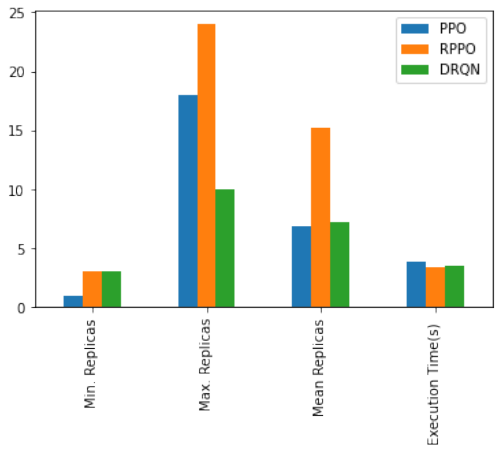

(a) PPO throughput vs execution time



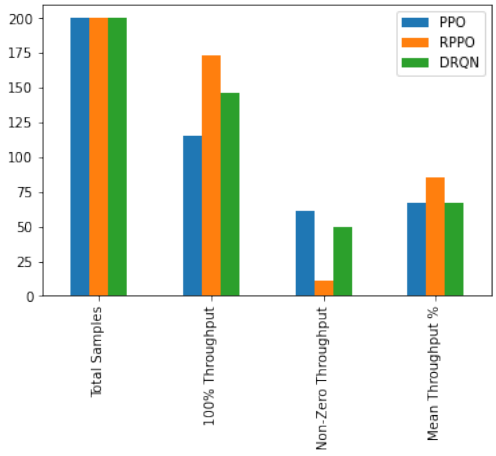

(b) RPPO throughput vs execution time




(c) DRQN throughput vs execution time

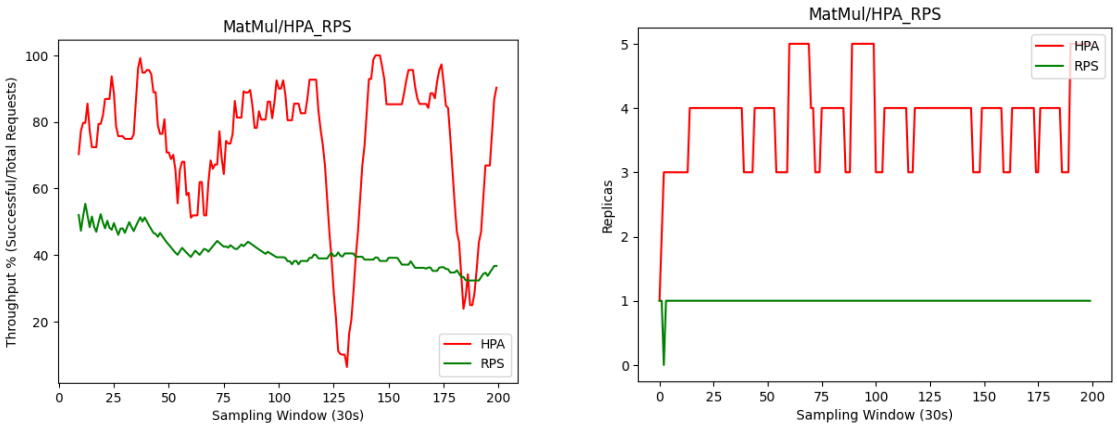



(d) Function replicas vs execution time




(e) Throughput comparison

**Figure 4.5:** Evaluation results: (a) PPO throughput vs execution time, (b) RPPO throughput vs execution time, (c) DRQN throughput vs execution time, (d) Function replica counts, (e) Throughput comparison.




(a) Throughput HPA vs RPS (b) Replica usage HPA vs RPS

**Figure 4.6:** Comparison of HPA vs RPS behaviour: (a) Throughput patterns, (b) Replica usage patterns.

ronment and acquire relevant information over multiple episodes that signify a higher degree of exploration. Although, as showcased through results, the proposed RL approach took more than 500 episodes (>6000 sampling windows) to slightly improve the performance over baselines, RL methods in real-time systems are considerably expensive following stringent optimisation goals.

The current training time has an episode of 5 minutes that consists of 10 iteration windows or epochs of 30 seconds each, where a decision to scale is taken by the agent and feedback is calculated for learning. This duration of an episode is chosen keeping in mind the minimum resource scaling and cooldown time of Kubernetes-based serverless platforms and an industry insight [32] for taking scaling actions in production environments. In addition to these settings, an agent training could further be affected by the invocation pattern and set of actions to be explored. In the current setup, the agent explores 5 discrete actions that follow a conservative approach to avoid resource thrashing while scaling function instances. In a particular state, an agent could take all the possible actions from the action space and would be penalised for an infeasible action. This static behaviour of action modelling elongates the training process since the agent explores infeasible actions in a state and only learns from negative experience. To overcome this, an action masking technique could be integrated that prevents the agent to take certain

infeasible actions in particular states, based on defined rules like the total number of function instances to remain within function limits. Therefore, different functions do not necessarily show similar behaviour for training and realised quality of results under similar settings.

The proposed DRL method is a composition of two different neural network techniques, recurrent and fully-connected layers, and these models are known to be sensitive to respective hyper-parameters or application/workload context. Therefore, configuring hyper-parameters can also be an intensive task in real-world settings. Additionally, the proposed agent analyses individual workload demand for a particular function, the learning cannot be generalised to other functions with different resource requirement profiles and therefore requires individual training models to be commissioned. However, techniques like transfer learning or categorising functions with similar resource and workload profile to use a trained agent as a starting point could be explored. Moreover, these agents could be deployed in similar fashion to tools like AWS Compute Optimiser [165] to gradually obtain experiences and build models with high confidence, from real-time data before making any recommendation/autonomous decision.

Furthermore, the agent is trained for a limited number of episodes, approximately 500 episodes and evaluated, but the chances of exploring are limited. Therefore, the agent expects to be guided by its actor-critic network policies in making informed decisions. Additionally, the agent utilises resource-based metrics that affect the cold starts, so the availability of relevant tools and techniques to collect instantaneous metrics is essential [32] in reducing the observation uncertainty. Also, the respective platform implementation, such as metric collection frequency, function concurrency policy, and request queuing, can extend support to the analyses. Hence, based on performance evaluation results and discussion, we can adequately conclude that the proposed LSTM-PPO agent successfully performs at par with competing policies for given workload and experimental settings.

**Software Availability:** Our environment setup code and algorithms we implemented for OpenFaaS can be accessed from: `https://github.com/Cloudslab/DRe-SCale`

## 4.6 Summary

FaaS platforms typically rely on reactive, threshold-based autoscaling mechanisms (based on metrics like CPU utilisation or rps) to manage dynamic demand. However, these methods are suboptimal because they neglect system complexity and specific workload characteristics when making scaling decisions. Therefore, an adaptive autoscaling mechanism is required to analyse workload patterns and system uncertainty to optimally scale resources, thereby improving system throughput and maintaining execution constraints.

In this chapter, we addressed this limitation by investigating a recurrent RL approach for function autoscaling. We modelled the FaaS environment as partially observable and leveraged the OpenFaaS serverless framework to present evidence that integrating recurrent networks with model-free RL algorithms can maximise the objective. We performed our analyses using a matrix multiplication function and successfully validated our hypothesis that recurrent techniques capture system dynamicity and uncertainty to yield superior autoscaling policies. However, the performance of a function and the associated scaling decisions made by the CSP are fundamentally constrained by the resources configured for the function. Moreover, the resources required for successful execution are highly influenced by external factors, such as the workload characteristics like function input. Therefore, the next chapter explores the function configuration space by investigating input-based resource prediction for performance and cost optimality.

# Chapter 5

# Input-based Dynamic Function Memory Configuration

*In FaaS, a programmer is typically responsible for configuring function memory for its successful execution, which allocates proportional compute resources such as CPU and network. However, right-sizing the function memory force developers to speculate performance and make ad-hoc configuration decisions. Research has highlighted that a function's input or payload (size, type) significantly impact the resource requirement, function performance and run-time costs with fluctuating demand. This correlation further makes memory configuration a non-trivial task. On that account, an input-aware function memory allocator not only improves developer productivity by completely hiding resource-related decisions but also drives an opportunity to reduce resource wastage and offer optimised pricing scheme. In this chapter, we propose a framework that estimates function memory requirement with input-awareness. It profiles the function in an offline stage and trains a multi-output Random Forest Regression (RFR) model on the collected metrics to invoke input-aware optimal configurations. We evaluate our proposal against state-of-the-art approaches on AWS Lambda to find significant cost savings and reduction in resource wastage .*

## 5.1 Introduction

The operational model of FaaS hides the complex infrastructure management from end users and does not signify the absence of servers. A serverless function still requires resources, including computing, network and memory, for successful execution. In

This chapter is derived from:

- **Siddharth Agarwal**, Maria A. Rodriguez, and Rajkumar Buyya, "Input-Based Ensemble-Learning Method for Dynamic Memory Configuration of Serverless Computing Functions", *in Proceedings of the 17th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2024),* Sharjah, UAE, December 16-19, 2024.

the current FaaS implementations, a developer is responsible for requesting the right combination of resources to guarantee successful function execution. However, service providers only expose a small set of resource knobs, usually memory [1] that proportionally allocates CPU, disk I/O, network bandwidth, etc. [83]. Prior studies [84][108][20] have identified that a higher memory configuration speeds up function execution and has a significant impact on its start-up performance and costs. However, the execution speedup is non-linear and has a diminishing marginal improvement with increasing memory allocation [93]. With limited observability into short-running functions and unaware of function performance, developers usually resort to speculative decisions for memory configuration or make experience-based ad-hoc decisions with an expectation to fulfil SLO [124]. To validate such developer behaviour, an industry insight [92] reports the ease of controlling function execution duration via memory configuration, while 47% of production-level functions still run with the lowest memory configuration (default) without exploring higher configuration space. Additionally, selecting an optimal memory configuration from an exponentially large configuration search space requires a careful understanding of the correlation between function performance and resource requirements. Hence, configuring the function with the right amount of memory that guarantees shorter execution times and lower execution costs is an intricate task.

Recent research [20][18][17][33] that seek to optimise the function resource allocation process have highlighted that a function's memory requirement vary drastically with input parameters and adversely impact its performance. Additionally, same memory configuration is used for concurrent function invocations while expecting similar performance for distinct function inputs. Therefore, setting a static memory configuration for all function invocations, regardless of their input, leads to fluctuating performance with varying workload and input arguments. This performance unpredictability demands an input-argument-aware approach in determining the memory configuration for function invocations that balances execution cost and running time while reducing excess resource allocation. This input-based memory configuration has a two-fold effect of providing a more autonomous developer experience and a chance for CSPs to

[1] We refer to FaaS platforms like AWS Lambda that allow developers to provide only memory configuration and allocate CPU, network bandwidth, etc., in a proportional fashion.

maximise resource utilisation and deliver a finer-grained, cost-effective pricing model for users. Additionally, these exiting efforts either focus on an average-case function execution to recommend maximum used memory/resources or propose to re-run their solution for specific input parameters to optimise the memory allocation process. This may lead to higher run-time costs and resource wastage and on the other hand, running multiple models for previously unseen input values extends the data collection process as well as increases the model training and tuning complexity. Therefore, a solution is warranted that captures the relationship of input parameters with function resources to precisely model and predict the required memory configuration for successful execution and reducing excess resource allocation.

To this end, we present MemFigLess, an end-to-end estimation and function memory allocation framework that makes input-aware memory configuration decisions for a serverless function. MemFigLess takes as an input the function details, such as the representative function input argument values, expected running time and cost SLOs and a range of memory allocations to explore. The framework executes an offline profiling loop to take advantage of a robust tree-based ensemble learning technique, multi-output RFR, which analyses the relationship between input parameters and other function metrics such as execution time, billed cost, and function memory requirement. The RFR model is then exploited in an online fashion to make an optimal selection of memory configuration for individual function invocations. Additionally, the framework provides a utility to keep track of its performance deviation via a feedback loop that retrains the model in a sliding-window manner, with a new set of collected metrics to capture the variation in function performance. This deviation may occur due to changes in SLOs, underlying infrastructure or co-location effect.

## 5.2 Motivation

To identify and establish the effect of input parameter on the memory requirement of the function and execution time [2], we experiment with the industry-leading FaaS platform, AWS Lambda [6] and conduct a large-scale initial study by deploying three benchmark

[2]memory requirement here mean resource requirement as other resources are allocated proportionally

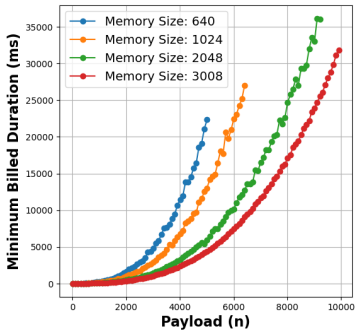



**Figure 5.1:** Payload vs Duration *matmul*

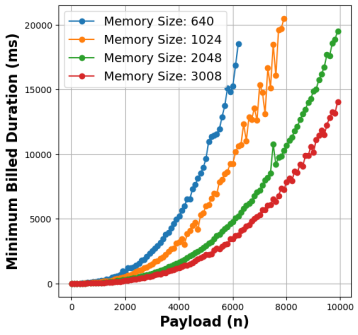



**Figure 5.2:** Payload vs Duration *linpack*

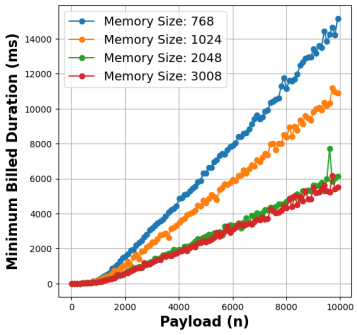



**Figure 5.3:** Payload vs Duration *graph-mst*

functions, (1) matrix multiplication (*matmul*), (2) graph minimum spanning tree (*graph-mst*), and, (3) *linpack*, from [166]. *matmul*, a CPU-bound function, calculates the multiplication of a $n * n$ matrix, whereas *linpack* measures the system's floating-point computing power by solving a dense $n$ by $n$ system of linear equations. *graph-mst* is a scientific computation that is offloaded to serverless functions and uses the python-igraph library to generate a random input graph of $n$ nodes using BarabsiAlbert preferential attachment and process it with the minimum spanning tree algorithm. To observe the effect of the payload parameter on the performance of benchmark functions, we execute them with input set $N = \{n|10 \leq n \leq 10000, n \leftarrow n + 100\}$ and vary the memory configuration of the function $M = \{m|128 \leq m \leq 3008, m \leftarrow m + 128\}$ MB. We take advantage of monitoring solution, Amazon CloudWatch [167] to gather function-level performance metrics.

We collect relevant function metrics as described in Table 5.1 and plot the function payload against the minimum billed duration and minimum memory used in Fig. 5.1 - 5.6.

### Insight 1:

*There is a **strong** correlation between the payload and the function execution time. This relationship varies in proportion to distinct memory allocations.*

We find a strong correlation between the function payload and the corresponding billed duration for all the benchmark functions. It can be inferred from Fig. 5.1 - 5.3

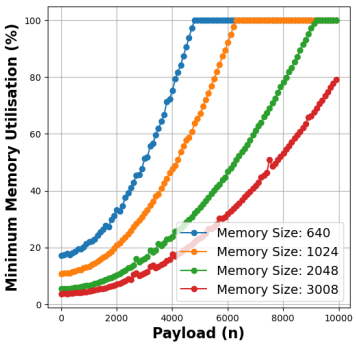



**Figure 5.4:** Payload vs Memory Utilisation *matmul*

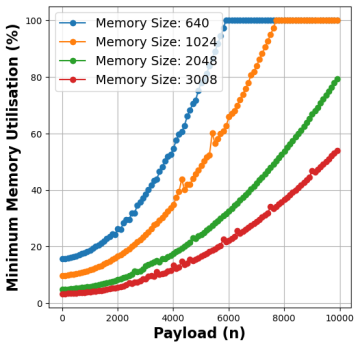



**Figure 5.5:** Payload vs Memory Utilisation *linpack*

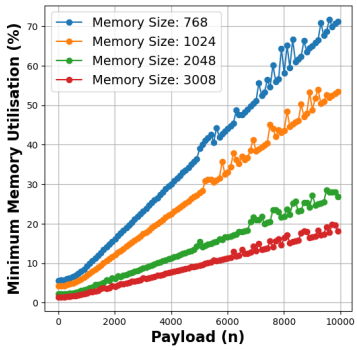



**Figure 5.6:** Payload vs Memory Utilisation *graph-mst*

that the minimum billed duration, i.e., the execution time of a function, is directly proportional to its input and has a tendency to increase with increasing payloads at distinct memory configurations. Therefore, we can confirm that different memory configurations lead to proportional resource allocations and thus, the function performance, i.e., execution time, also varies in proportion to these available resources. This complex relation of payload-dependent execution time further aggravates the overall function run-time cost as it is calculated based on the execution time and allocated memory.

### Insight 2:

*There is a **positive** correlation between the payload and minimum memory utilised for successful execution of the function, which has a direct and complex relationship with resource wastage and run-time cost.*

In Fig. 5.4 - 5.6, we observe the effect of function payload on memory utilisation. A higher memory utilisation is observed for all the payload values at the lower memory allocations, while a lower memory utilisation can be seen at higher memory allocations for all benchmark functions. Therefore, an under-utilised function resource depicts an inherent resource wastage, where the function payload has a direct effect on it. However, the relationship may not be the same for a function and payload combination, and thus requires thoughtful resource allocation to reduce excess resource wastage and associated run-time costs. In addition to this, it is well established in previous research

**Table 5.1:** List of collected function metrics

| Metric Name | Description |
|---|---|
| request_id | unique function invocation ID |
| payload | function input parameter(s) |
| memory_size | amount of memory allocated to function |
| memory_utilisation | maximum memory measured as a percentage of the memory allocated to the function |
| memory_used | measured memory of the function sandbox |
| billed_duration | function execution time rounded to nearest millisecond |
| billed_mb_ms | total billed Gb-s, a pricing unit for function |
| cold_start | function cold start (true/false) |
| init_duration | amount of time spent in the init phase of the execution environment lifecycle |
| function_error | any function run-time error |

studies [93][33][34] that a function experiences a speed up in performance with additional allocated resources and hence, a complex association exists between the function resource allocation and pricing schemes. Therefore, this chapter attempts to address the key challenges identified in the motivation.

## 5.3 Related Work

In this section, we summarise existing work on function configuration and, specifically, input-focused resource configuration for performance enhancement.

The authors in [83] propose a multi-regression model generated on synthetic function performance data and use it to predict execution time and estimate cost at distinct memory configurations. Similarly, [105] explores search algorithms like linear/binary search and gradient descent to determine the optimal configuration for cost-focused or balanced optimisation goals. However, neither of them considers the effect of payload on function performance and resource demand and either rely on synthetic data or per-

form a search across configurations.

Jarachanthan et al. [96] explore the performance and cost trade-off of the function at different configurations for Map-Reduce-style applications. They propose a graph-theory-based job deployment strategy for DAG based applications to optimise the resource configuration parameters. Wen et al. [108] focus on multicore-friendly programming for workflows to estimate the inter- / intra-function parallelism based on weighted sub-SLOs. Safaryan et al. [45] focus on the SLO-aware configuration of workflows and use a max-heap data structure to find a configuration repeatedly. However, these works address function workflows and either fit specific application style or do not configure function resources and fall short to realise the payload effect.

The work in [114] introduces an urgency-based heuristic method where a particle swarm optimisation technique is used for time/cost trade-off. Lin and Khazaei [88] propose a probability refined critical path greedy algorithm for selecting optimal function configuration from the profiled data. The authors in [124] discuss the concept of CPU time sharing and resource scaling with memory configuration to propose a memory-to-vCPU Regression model. However, these works model the behaviour of functions and fall short to capture the payload effect on function performance or assume an homogeneous function resource relationships. Raza et al. [93] explore a BO based model to optimally configure and place functions considering their execution time and cost. However, they disregard the payload effect on resource configuration and performance.

In [84], the authors establish a relationship between input/output parameters and response time to predict their distribution via offline Mixture Density Networks (MDN) and utilise online Monte-Carlo simulations to estimate best workflow costs. However, these estimates consider fixed memory configuration for performance and cost prediction. Researchers in [18] discuss a high correlation between input arguments and memory configuration for Extract-Transform-Load (ETL) pipeline functions using Decision Trees and re-claim unused memory to resize worker-node cache for locality-aware function executions. However, this is application specific and focuses on cache memory management. Bhasi et al. [17] presents an input-size sensitive resource manager that perform request batching, re-ordering and rescheduling to minimise resource consumption and maintaining a high degree of SLO. However, they focus on request management and

discard the opportunity of input-sensitive function configuration.

Bilal et al. [20] re-visits BO models and Pareto front prediction, among few to discuss a complex challenge of input dependency for resource allocation. However, they primarily answer design space questions to showcase potential opportunities for flexible resource configuration. Sinha et al. [34] presents an online supervised learning model to allocate the minimum amount of CPU resources for individual invocations based on input characteristics and function semantics. However, they introduce an overhead of supervising individual invocations for SLOs in addition to adjusting only CPU resources.

Moghimi et al. [33] criticise available function right-sizing tools for reducing developer efficiency and explores a characterisation-driven modelling tool that takes advantage of parameter fitting, adaptive sampling and execution logs to find the right function configuration. Although they consider relative payloads while suggesting optimal configuration, they recommend re-execution of their model for individual payloads. This introduces a run-time overhead to find payload specific configuration in terms of profiling time and costs. Therefore, with the proposed framework we attempt to address payload-aware function memory configuration, where other resources are proportionally allocated, to satisfy run-time performance constraints and make these configuration decisions online.

## 5.4 Problem Formulation and Architecture

In this section we formally describe the function configuration problem and introduce the system architecture.

### 5.4.1 Problem Formulation

In current FaaS environments, developers must configure memory settings [3] for their functions to ensure successful execution. However, determining the appropriate memory configuration is challenging due to factors like variability in function input or payload characteristics (e.g., input size, type, and number of inputs) and invocation fre-

[3]We refer to FaaS platforms like AWS Lambda that allow developers to provide only memory configuration and handle the CPU, network bandwidth, etc. in a proportional fashion.

quency, which significantly impact resource demands, run-time performance and cost. To handle this, developers generally make ad-hoc decisions to either configure platform defaults [92] or speculate the right-sizing of functions based on past experience. These decisions lead to sub-optimal resource allocations, over- or under-provisioning, which may result in resource wastage, added run-time costs, and throttled function performance. Additionally, FaaS platforms scale a function with static resource configuration with fluctuating workload which further makes the resource scaling and allocation challenging.

Existing research [83] [93][168][34] have repeatedly accentuated the complex relationship of function resource demand, execution time guarantee and run-time cost, which is further complicated with the attention to function payload [33]. Therefore, we formulate the problem of payload-aware function configuration as a multi-objective optimisation (MOO) problem where the objective is to select a memory configuration that guarantees a successful execution within an advertised function deadline, while reducing excess resource allocation and run-time costs for incoming function payloads. The problem can be mathematically represented as Eq. 5.1, such that the constraints Eq. 5.2 are satisfied.

$$\min_{m \in M} G(m, P) = (C_f(m, P), T_f(m, P)) \tag{5.1}$$

**Subject to:**

$$T_f(m, P) \leq D, \tag{5.2}$$

$$C_f(m, P) = T_f(m, P) * C_m + \beta \leq B, \tag{5.3}$$

$$\text{Success}(m, P) = 1. \tag{5.4}$$

In FaaS, a function $f$ may expect a number of inputs, $P = \{P_1, \ldots, P_n\}$ where an input belongs to a range of values $P_n = \{p_n^{min}, \ldots, p_n^{max}\}$, either continuous or discrete, which influences the function memory requirements $m \in M = \{m^{min}, \ldots, m^{max}\}$. We define the objective of payload-aware memory configuration to minimise the run-time cost $C(m, P)$ and the function execution time, $T(m, P)$ at memory allocation $m$ and pay-

load(s), $P$ such that a function executes successfully within a specified deadline, $D$ and budget, $B$, constraints. The run-time cost $C_f(m, P)$ is directly proportional to the execution time, $T_f(m, P)$ and cost of memory configuration $C_m$ plus a constant invocation amount, $\beta$ (provider dependent).

### 5.4.2 System Architecture

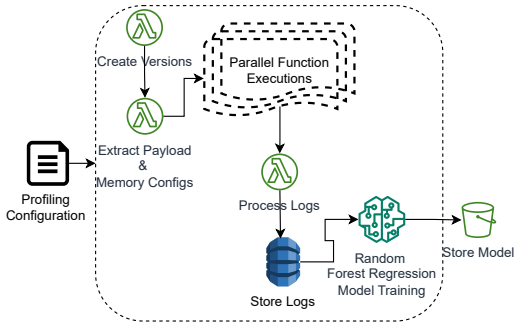
The proposed MemFigLess system architecture consists of both offline and online components designed to optimise memory allocation for FaaS offerings based on the payload. The model of the proposed framework is a $MAPE$ control loop, i.e., *Monitor*, *Analyse*, *Plan* and *Execute*. In the online stage, a periodic monitoring of function performance metrics is performed, which are then analysed by the resource manager via the feedback unit to plan and execute the resource allocation decisions to meet the performance SLOs.

#### Offline Profiling and Training Module

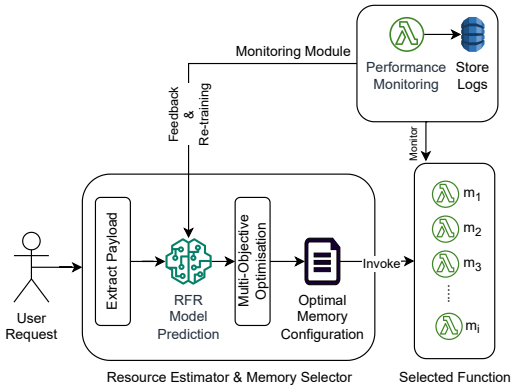
This module is responsible for profiling the functions and training the RFR model, Fig. 5.7(a). Functions are executed with a variety of inputs to collect data on their performance and resource usage. Metrics such as input size, number of inputs, memory consumption, execution time and billed execution unit are recorded. The collected metrics are stored in a structured format to serve as training data for the model. A tree-based ensemble learning RFR model is trained on the collected data to learn the relationship between the functions input/payload and its memory requirement and execution time. The model captures the impact of input size distribution on resource usage and is used for online payload-aware memory estimation.

#### Online Prediction and Optimisation Module

This module leverages the trained RFR model to select the optimal function memory based on the model prediction and constraint optimisation, and invoke functions in real-time, as shown in Fig. 5.7(b). Incoming function requests are analysed to extract payloads which are fed to the trained RFR model to predict the execution time at dis-




(a) MemFigLess offline profiling and data collection workflow.




(b) MemFigLess online workflow.

**Figure 5.7:** MemFigLess System Architecture

tinct function memory configurations. The selected memory configurations, based on SLO constraints, are used for online constraint optimisation, either cost or execution time, provided by the user. This helps in reducing the potential resource wastage and overall cost of execution.

### Dynamic Resource Manager

This component is embedded within online prediction module to handle the invocation and allocation of resources based on the predictions made by the RFR model. The Dynamic Resource Manager dynamically allocates function resources or selects the available function instance based on the constraint-optimised memory selection, ensuring minimal wastage and optimal performance.

A continuous monitoring and feedback mechanisms are also incorporated to improve the accuracy and performance of the system over time. This module can monitor and log the performance of functions during execution within a configured *monitoring_window* to ensure that model captures performance fluctuations, periodically. The gathered performance data is leveraged by training module to periodically re-train and improve the RFR model predictions.

## 5.5 Multi-Output Random Forest Regression

To accurately select the memory configuration of serverless functions, a Random Forest (RF)-based regression algorithm can be employed. The RF, initially presented by Breiman [169], is one of the most popular supervised ML algorithms and has been successfully applied to both classification and regression in many different tasks, such as VM resource estimation [170], VM resource auto-scaling [171] and computer vision applications [172]. The RF algorithm uses a combination of decision trees (DT) to model complex interactions between input parameters and identify patterns in the data. It works by training multiple DTs on subsets of the input parameters and then aggregating their predictions to generate the final estimation. This method has been demonstrated to have the ability to accurately approximate the variables with nonlinear relationships and also have high robustness performance against outliers. In addition, compared to other ML techniques, e.g., Artificial Neural Networks (ANN), Support Vector Machine (SVM), Deep Learning and RL, it only needs a few tunable parameters and therefore requires low effort for offline model tuning. Furthermore, the algorithm can handle noisy or incomplete input data, and reduces the risk of overfitting which might occur with other ML algorithms [173].

The RF model is an ensemble-learning method that can be modelled as a collection of DTs. A DT makes a prediction for the input feature vector $\vec{x} \in F$, where $F$ represents a subset of $\kappa$-dimensional feature space [169]. A DT recursively partitions the feature space $F$ into $L$ terminal nodes or leaves that represents the region $R_l, 1 \leq l \leq L$ where every possible feature vector $\vec{x}$ may belong. Therefore, an estimation function $f(\vec{x})$ for a DT can be summarised as Eq. 5.5, where $I(x, R_l)$ represents the indicator function of whether the feature vector $\vec{x}$ belongs to region $R_l$. The function $f(\vec{x})$ indicates how the DTs returns the value of leaf corresponding to the input $\vec{x}$ and typically learns a response variable $c_l$ for each region $R_l$ where $\vec{x}$ belongs, to assign an average value to that region in the regression tree [171].

$$f(\overrightarrow{x}) = \sum_{l=1}^{L} c_l I(\overrightarrow{x}, R_l) \tag{5.5}$$

$$I(\overrightarrow{x}, R_l) = \begin{cases} 1; & \overrightarrow{x} \in R_l \\ 0; & \overrightarrow{x} \notin R_l \end{cases} \tag{5.6}$$

However, a more interpretative representation is Eq. 5.7 where $c^{full}$ is the average of all learned response variables during the training and $C(\vec{x}, k)$ is the contribution of the $k^{th}, 1 \leq k \leq \kappa$ feature in $x$.

$$f(\overrightarrow{x}) = c^{full} + \sum_{k=1}^{K} C(\overrightarrow{x}, k) \tag{5.7}$$

Therefore, the average prediction $F(\vec{x})$ for a RFR model over an ensemble of DTs can be summarised as Eq. 5.8, where $S$ is the number of DTs, $C_s^{full}$ is the contribution of $k^{th}$ feature in vector $x$ in $j$-th DT.

$$F(\overrightarrow{x}) = \frac{1}{S} \sum_{s=1}^{S} c_s^{full} + \sum_{k=1}^{\kappa} \left( \frac{1}{S} \sum_{s=1}^{S} C_s(\overrightarrow{x}, k) \right) \tag{5.8}$$

To apply the RFR in a serverless framework, first, we need to collect relevant function performance metrics at distinct representative payloads. This data is gathered through a series of experiments, Sec. 5.4.2, where each input parameter is varied, and the resulting function metrics like memory consumption are measured. Once we have gathered enough data on the memory configuration of the serverless function, we train a RFR model, as outlined in Sec. 5.4.2. In our problem context, the input vector $\vec{x}$ has multiple components, which are *total_memory* allocation $m$ and function *payload(s)*, $P$. This RFR model predicts the *billed_duration* and *memory_utilisation*, which directly computes run-time cost and execution duration, and *function_error_status* for determining successful execution. These predictions align with the objectives of function run-time cost $C(m, P)$ and execution time $T(m, P)$ while ensuring a successful execution.

To address the conflicting objectives i.e., executing a function within a deadline, $D$ and with a run-time budget, $B$, commonly, the concept of Pareto dominance and Pareto optimality are used [174]. This optimisation is integrated with online prediction mod-

ule 5.4.2, to select a payload-aware and constrained optimised memory configuration. Pareto dominance is a method for comparing and ranking the decision vectors. A vector $\vec{x_u}$ is said to dominate vector $\vec{x_v}$ in the Pareto sense, if the objective vector $G(\vec{x_u})$ is better than $G(\vec{x_v})$ across all objectives, with atleast one objective where $G(\vec{x_u}) > G(\vec{x_v})$, strictly. A solution $\vec{x}$ is said to be Pareto optimal if there does not exist any other solution that dominates it and then the objective $G(\vec{x})$ is known as Pareto dominant vector. Therefore, a set of all Pareto optimal solutions is called Pareto set and the corresponding objective vectors are said to be on a Pareto front.

We approach our MOO by transforming the multi-objective problem into a single objective problem and employing the classical Weighted Aggregation Method (WAM), where a function operator is applied to the objective vector $G(\vec{x})$. As a user is responsible for providing the relative importance of objectives, we select a linear weighted combination as the utility function for objective optimisation. Therefore, the final optimisation problem can be simplified as Eq. 5.9 where $J_z$ represents $z^{th}$ objective with a relative weight of $w_z$ and a weighted combination of all the objectives are jointly minimised.

$$\min_{\overrightarrow{x}} Z = \sum_{1}^{z} w_z * J_z(\overrightarrow{x}) \tag{5.9}$$

**Subject to:**

$$w_z \geq 0 \; ; \; \sum_{1}^{z} w_z = 1 \tag{5.10}$$

Once the Pareto front is obtained using the discussed MOO, we select the memory configuration that is cheapest in terms of resource allocation, i.e., the lowest memory configuration from the Pareto optimal solutions and execute it via the dynamic resource manager, Sec. 5.4.2.

## 5.6 Performance Evaluation

In this section, we briefly discuss the implementation along with experimental setup, model parameters, and perform an analysis of the proposed RFR-based framework compared to other complementary solutions.

### 5.6.1 Implementation and System Setup

We setup our proposed solution using AWS serverless services [175], such as AWS Lambda [6], AWS DynamoDB [52], AWS Step Functions Workflow [56] and Amazon Simple Storage Service (S3) [51] for an end-to-end serverless function configuration solution. The framework can be assumed a CSP service where users can subscribe to it for an end-to-end optimisation, based on the desired deadline and run-time cost of the individual function. The offline step is implemented as a Step Functions Workflow that takes function details such as resource name, memory configurations to explore, number of profiling iterations and the representative payload(s) as a *json* input file. This information is used to create and execute a function with different payloads at distinct configurations, and collect the performance data using AWS CloudWatch [167]. However, the payload for different functions may be of different types and thus, the workflow utilises the AWS S3 service to fetch any stored representative payload. The individual workflow tasks of creating, executing and updating the function in addition to log collection and processing, are implemented as AWS Lambda functions. All the functions are configured with a 15 minutes timeout, 3008 MB (maximum free tier) memory configuration, and 512 MB ephemeral storage, to avoid any run-time resource scarcity. The processed logs are stored in AWS DynamoDB, a persistent key-value datastore. These logs are utilised by RFR training function and the trained model is placed in S3 storage for online estimation.

The online step makes use of the trained RFR model which is also implemented as a function, assuming shorter executing functions. The RFR model is implemented using Scikit-Learn [176], a popular ML module in Python. This function loads the RFR model for inference, estimates resources, performs the optimisation and invokes the selected function configuration. In addition, performance monitoring can be scheduled to collect

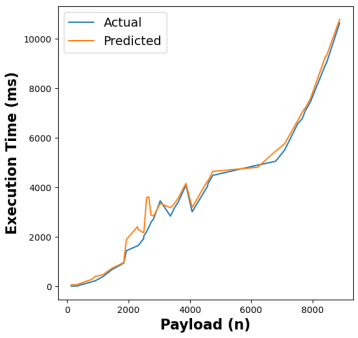
**Table 5.2:** List of functions and payload value

| **Function Name** | **Payload** |
|---|---|
| matmul | $n$, size of matrix |
| linpack | $n$, number of linear equations to solve |
| pyaes | $\{n, m\}$, length of message to encrypt and number of iterations |
| graph-mst | $n$, size of random graph to build |
| graph-bfs | $n$, size of random graph to build |
| graph-pagerank | $n$, size of random graph to build |
| dynamic-html | $n$, random number to generate an HTML page |
| chameleon | $\{n, m\}$, number of rows and columns to create an HTML table |

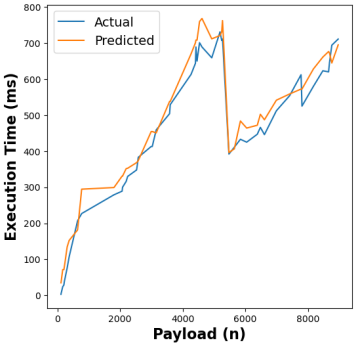
and store the logs in the key-value datastore and a *monitoring_window* can be setup for periodic log processing and model re-training.

The RFR model assumes that the payloads provided in the offline step are representative of actual payload and therefore, inference at any anomalous/outlier value is defaulted to 3008 MB or the estimated configuration for the smallest payload value seen. The inference model expects a function deadline $D$, run-time budget $B$ and their relative importance, $w_z$, for optimisation and configuration selection. Based on the initial analysis we configure the deadline $D$ as the mean function execution time, and the run-time budget $B$ as the mean execution cost across memory and payload combination.

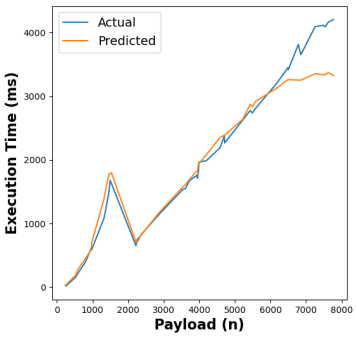
We perform our experimental analysis on a range of functions implemented in Python v3.12, taken from serverless benchmarks [166] and [107], including CPU/memory intensive (*matmul, linpack, pyaes*), scientific functions (*graph-mst*, *graph-bfs*, *graph-pagerank*) and dynamic website generation (*chameleon, dynamic-html*). The explored functions and required payloads are listed in Table 5.2 and the experimental payload values range between $[10, 10000]$ with a step_size of 200, for individual variables. This *step_size* was randomly chosen for the experiments and must be provided by the user for the granularity of experiments and model generation.



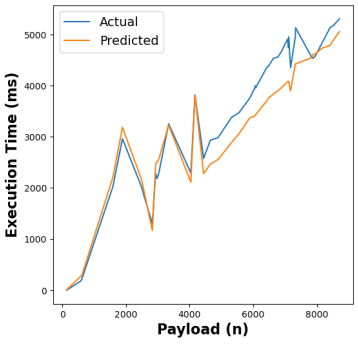

(a) RFR Execution time estimates for *linpack*



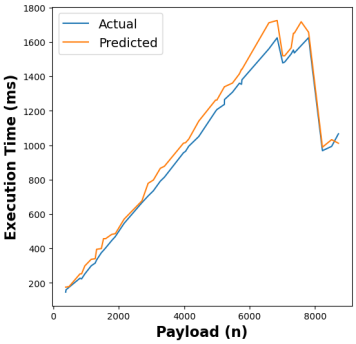

(b) RFR Execution time estimates for *graph-pagerank*



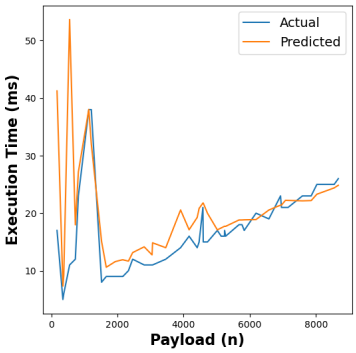

(c) RFR Execution time estimates for *graph-bfs*



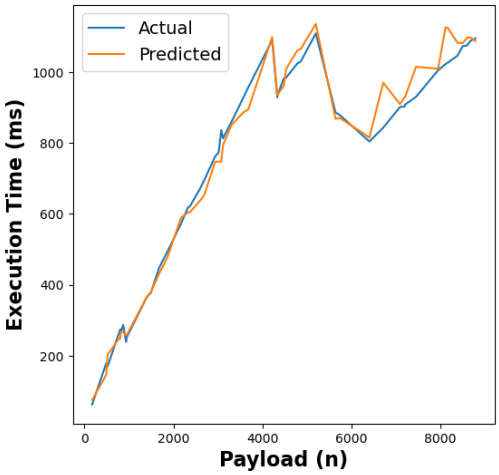

(d) RFR Execution time estimates for *graph-mst*



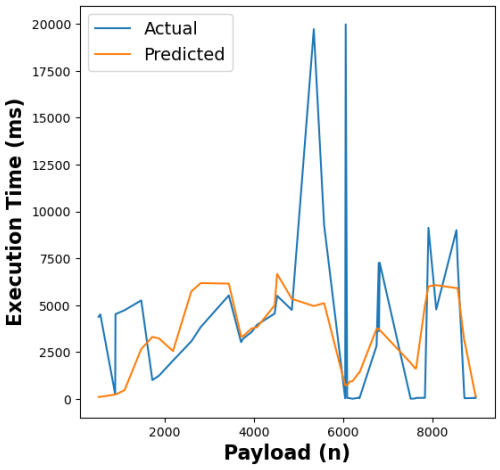

(e) RFR Execution time estimates for *chameleon*




(f) RFR Execution time estimates for *dynamic-html*

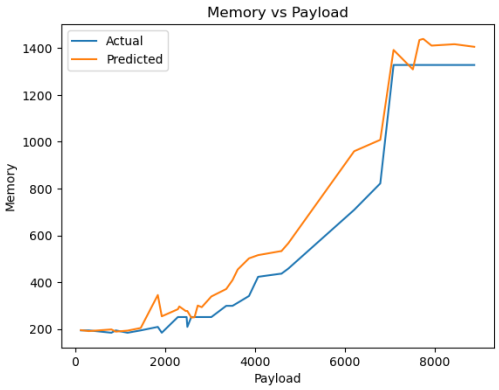



(g) RFR Execution time estimates for *pyaes*

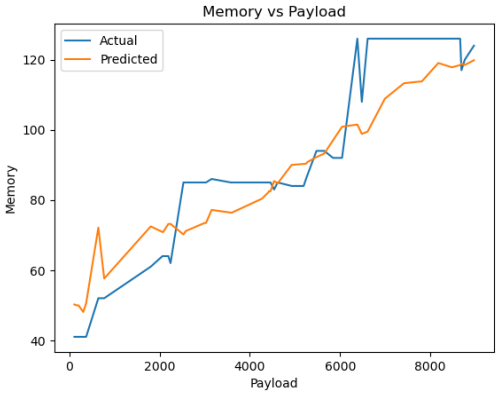



(h) RFR Execution time estimates for *matmul*

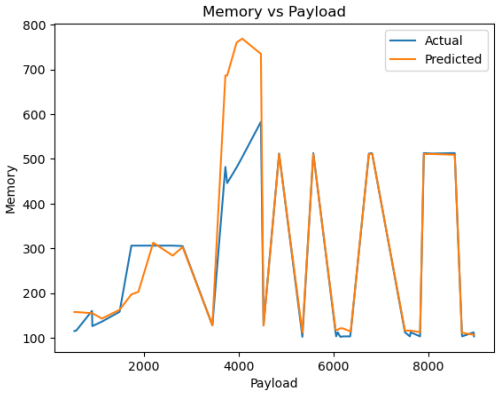
**Figure 5.8:** RFR Model Payload-Aware Execution Time Prediction.



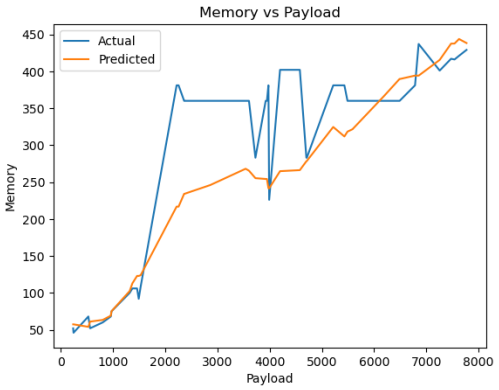

(a) RFR Memory utilisation estimates for *linpack*




(b) RFR Memory utilisation estimates for *graph-pagerank*




(c) RFR Memory utilisation estimates for *matmul*




(d) RFR Memory utilisation estimates for *graph-bfs*

**Figure 5.9:** RFR Model Payload-Aware Memory Utilisation Prediction.

### 5.6.2 Experiments

We run the profiling step of the proposed solution with the given function payload(s) at distinct memory configurations to capture the relationship between payload and resource demands. After the profiling step, a RFR model is trained on the collected data, accompanied by a hyper-parameter tuning that explores model parameters such as *n_estimators, max_depth, min_samples_split and min_samples_leaf* via *grid_search* and selects the best configuration for inference. The RFR model then estimates the memory utilisation and function execution time to perform online optimisation and selects the best possible payload-aware configuration to invoke functions.

We perform the payload-aware estimation and optimisation of memory configuration for 50 payload values, within the discussed range. We select the relative importance, $w_z$, as 0.5 to balance the function execution time, $T_f(m, P)$ and run-time cost, $C_f(m, P)$ constraints for this experiment. In Fig. 5.8, we showcase the ability of RFR model to predict the execution time of the functions. The proposed methodology estimated the execution time with a $R^2$ score of as high as 98% for *linpack* and *pyaes* function while having $R^2$ scores of 97%, 94% and 91% for functions such as *graph-pagerank, graph-mst, graph-bfs* and *dynamic-html*. This statistical measure of $R^2$ score demonstrates the goodness of the fit by the regression model. Additionally, we observe in Fig. 5.9 that for memory-intensive functions, the RFR model is able to capture the relationship of payload and memory with high $R^2$ score of 96% in case of *linpack*, 87% for *graph-pagerank*, 79% for *matmul* and as low as 73% for *graph-bfs* function with a mean absolute error (MAE) of 269 MB, 36MB, 2609 MB and 192 MB, respectively. These observations are based on the actual performance data acquired after invoking functions with RFR estimated values. Therefore, we can conclude from the observations that RFR model can be utilised for predicting payload-aware function execution time and memory utilisation. In addition, the proposed RFR-based framework can take advantage of online estimation and optimised solutions to invoke the respective payload-aware configurations.

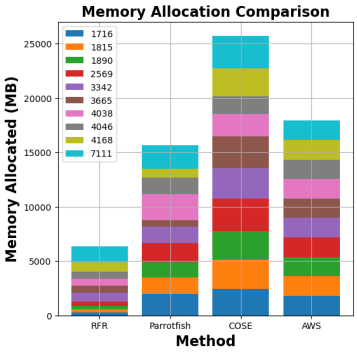
To evaluate our model's efficiency in reducing excess resource allocation and higher run-time costs, we compare our proposal with the following existing works -

1. COSE [93]: a BO-based function memory configuration tool that tries to select best configuration at each sample which maximises the model confidence.

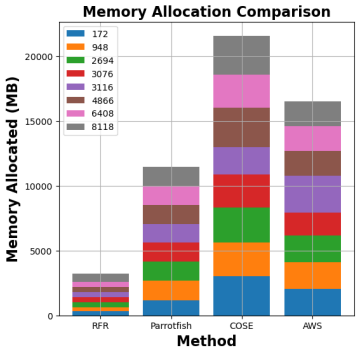
2. Parrotfish [33]: an online Parametric Regression based function configuration tool that selects optimal configuration while satisfying user-defined constraints.

3. AWS Lambda Power Tuning [168]: a recommendation and graphical tool by AWS that performs an exhaustive search of memory configurations to suggest a cheaper and lower execution time configuration.

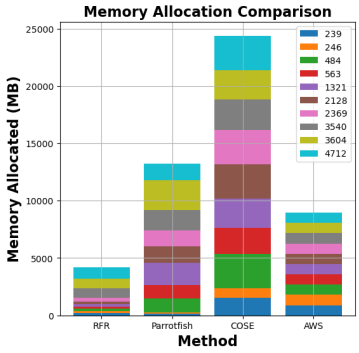
For the brevity of this evaluation, we run the respective approaches for atmost 10 payload values to find the payload-aware optimal memory configuration for 4 functions i.e., *graph-mst, pyaes, matmul* and *graph-bfs*. Similar results are reported for other



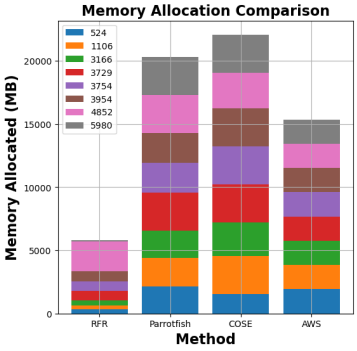

(a) Memory allocation for *graph-mst*



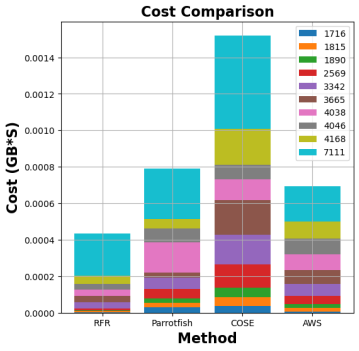

(b) Memory allocation for *pyaes*



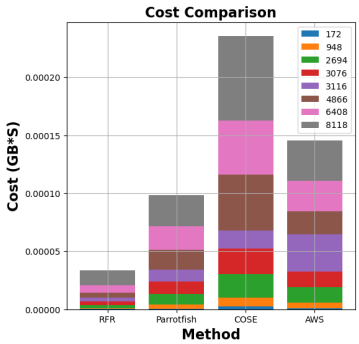

(c) Memory allocation for *graph-bfs*



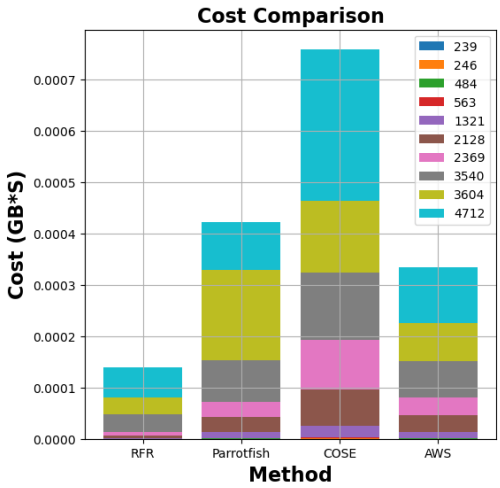

(d) Memory allocation for *matmul*



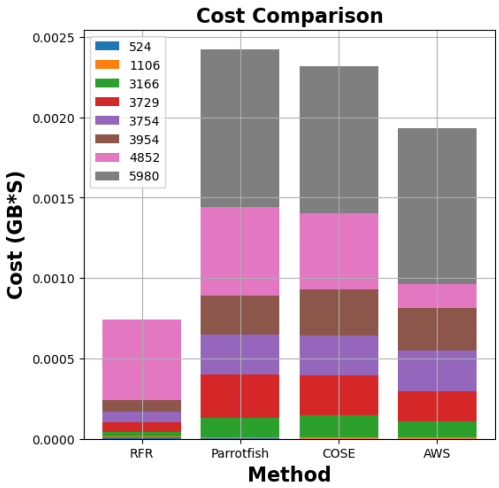

(e) Run-time cost for *graph-mst*




(f) Run-time cost for *pyaes*




(g) Run-time cost for *graph-bfs*




(h) Run-time cost for *matmul*

**Figure 5.10:** Comparison of Memory Allocation and Run-time Costs for competing approaches.

functions and thus have been skipped from this discussion. We select the function execution time as the objective and utilise distinct payloads to predict the execution time. However, COSE does not provide any utility or provision to optimise for specific payloads, therefore, we run the COSE tool to probe 20 sample points for each payload value. On the other hand, Parrotfish samples and tries to optimise the memory configuration based on weighted representative payloads via parametric regression. The tool recommends the individual optimal memory configuration found i.e., the cheapest configuration within the deadline, for that specific run with the weighted payload(s). However, we run Parrotfish for each distinct payload to find the optimal configuration and ignore any interference effect of other payloads. We also run [168] for individual payloads with a set of memory configurations that do not raise any runtime errors.

In Fig. 5.10 we share the results of MemFigLess estimation and execution as compared to COSE and Parrotfish. The results are optimised for 1.5 times the function deadline, $D$ and no weight is given to the run-time cost. However, we observe that the proposed RFR-based MemFigLess is able to estimate and utilise the Pareto optimal results to allocate a lower memory configuration as compared to other works, given a function deadline. In terms of memory allocation, MemFigLess allocates 54%, 75% and 65% less cumulative memory as compared to Parrotfish, COSE and AWS Power Tuning for *graph-mst*. Additionally, this allocation allows to save 57%, 79% and 58% in cumulative run-time costs of *graph-mst* against when run with Parrotfish and COSE selected configuration. The gains are more visible for *pyaes* function, where MemFigLess is able to save 82% additional resources as compared to COSE leading to 84% cost benefits. Similar results are achieved for *graph-bfs* where MemFigLess saved 65% and 75% resources as compared to Parrotfish and AWS Power Tuning, and was 87% cost efficient in comparison to COSE. For *matmul* function, the resource savings are approximately 73% as compared to COSE and Parrotfish. Therefore, we conclude based on the experimental analysis that the RFR-based solution is able to reduce the run-time costs and excess resource allocation as compared to SOTA techniques, [93][33][168], while satisfying the deadline constraint.

### 5.6.3 Discussion

We investigate a payload-aware function configuration methodology and present MemFigLess, a RFR-based workflow to estimate the payload-aware optimal resource allocation schemes. The experiments are performed with distinct functions deployed on the AWS Lambda platform and take advantage of workflows to create the offline profiling and training stages of MemFigLess. We assume that representative payload(s) are provided for profiling to supplement the regression model. In addition to this, we make a heuristics-based decision to allocate maximum memory, 3008 MB to an unseen payload larger than the profiled limit or to configure the memory of the smallest payload seen. This builds on the idea [33] that if a configuration is good for an input $x$, then it is also good, if not better, for inputs smaller than $x$.

We profile and train the functions at memory intervals of 128 MB, however, in the online inference, estimates are made for every possible memory configuration with a step size of 1 MB, in line with [33] and [6]. This makes the inference time complexity $O(n + k \log k)$ where $n$ represents the number of explored memory configurations during optimisation with $k \leq n$ configurations in Pareto front calculation. As we employ a RFR model that generally requires prior data for precise estimates, these finer estimates may suffer in case of complex payload and resource relationships. Therefore, the accuracy of the estimates is highly dependent on the memory intervals and representative payloads used in the initial profiling to capture complex resource relationships. Furthermore, finer online inferences may suffer from increased cold starts if the estimated configurations are largely distinct for incoming payloads. To support this, MemFigLess intelligently checks for existing functions with estimated configuration to execute. However, it does not control the degree of container reuse that utilises a warm function configuration to avoid a cold start. In addition to this, a sequential workload is considered for analysis in anticipation of minor performance fluctuations owing to AWS Lambda guaranteed concurrent invocations. With the discussed experimental assumptions and setup, the proposed solution is able to reduce the excess resource allocation and reduce the run-time cost of functions in comparison to Parrotfish and COSE, which are used as is with their defaults.

**Software Availability:** Our results and the benchmark functions adapted to work

with AWS can be accessed from: `https://github.com/SidAg26/MemFigLess`

## 5.7 Summary

In this chapter, we present MemFigLess, a RFR-based payload-aware solution to optimise function memory configuration. This solution is implemented using AWS serverless services and deployed as a workflow. A motivation study is conducted to highlight the importance of payload-aware resource configuration for performance guarantees. We identify a strong and positive correlation between function payload, execution time and memory configuration to formalise the resource configuration as a MOO problem. A concept of Pareto dominance is utilised to perform online resource optimisation.

The MemFigLess framework successfully solves the static configuration dilemma by enabling input-aware resource prediction and identifying cost-optimal resource settings, significantly reducing excess memory allocation and saving run-time costs. However, applying this strategy at the platform level necessitates deploying a fleet of functions with diverse, input-specific configurations. This version-proliferation creates a final, critical scheduling challenge of how to intelligently and adaptively orchestrate incoming requests to balance the cost of cold starting a new instance with specific configuration against the performance benefit of reusing an existing, warm instance. The following chapter addresses this end-to-end request orchestration problem by integrating input-aware predictions with a multi-objective scheduling framework.

# Chapter 6

# Optimising End-to-End Serverless Workloads

*Despite the benefits of FaaS, challenges such as startup latencies, static configurations, sub-optimal resource allocation and scheduling still exist due to coupled resource offering and workload-agnostic generic scheduling behaviour. These issues often lead to inconsistent function performance and unexpected operational costs for users and service providers. This chapter introduces Saarthi, a novel, end-to-end framework that intelligently manages the dynamic resource needs of function workloads, representing a significant step toward self-driving serverless platforms. Unlike platforms that rely on static resource configurations, Saarthi is input-aware, allowing it to intelligently anticipate resource requirements based on function input. This approach reinforces function right-sizing and enables adaptive request orchestration across available function configuration instances. Saarthi integrates a proactive fault-tolerant redundancy mechanism and employs a multi-objective ILP model to maintain an optimal function quantity, aiming to maximise system throughput while simultaneously reducing overall operational costs.*

## 6.1 Introduction

In FaaS, functions require compute resources for their execution, and developers are responsible for statically configuring them. However, it is a well-researched fact that the amount of configured resources significantly impacts the performance of functions [20][33], affecting both execution time and operational cost. Consequently, determin-

This chapter is derived from:

- **Siddharth Agarwal**, Maria Rodriguez Read, and Rajkumar Buyya, "Saarthi: An End-to-End Intelligent Platform for Optimising Distributed Serverless Workloads", *40th IEEE International Parallel and Distributed Processing Symposium*, May 25-29, 2026, New Orleans, USA.[Under Evaluation, September, 2025]

ing function resources emerges as a pivotal deployment decision that directly influences the user's perceived latency and costs incurred by developers. For example, allocating more resources than required improves function execution time, but it may lead to increased operational cost. Conversely, under-provisioning a function's resources can throttle its performance, potentially leading to failed executions. Furthermore, a function's resource requirements are not static, as they dynamically vary with the workload characteristics such as input payload, its data type, size, and complexity [18]. This makes a single static configuration inefficient across different executions, as a setting optimised for one workload may be significantly insufficient or over-provisioned for another. Since FaaS platforms often do not provide service-level guarantees [45] for execution time, developers resort to extensive manual profiling to determine optimal configurations. This introduces a significant overhead in terms of both developer time and operational cost, as the process is tedious, error-prone, and rarely accounts for the full range of dynamic inputs.

However, acknowledging input-aware memory variability leads to a paradigm where multiple function versions with different configurations co-exist. This input and version-aware function management makes scheduling decisions non-trivial and introduces a new set of operational challenges. It is impractical to instantiate a new function instance for every incoming input as it incurs an expensive cold start overhead. Additionally, frequent creation of distinct function versions leads to excessive resource fragmentation for the service provider and impacts their ability to efficiently make scheduling and placement decisions [20]. Therefore, it only makes sense to strike a balance between reusing existing versions with sufficient resources or deploying new function versions tailored to the incoming workload. This strategy ensures that the system not only adapts to the dynamic workload needs, but also reduces redundant overheads with minimal developer intervention while keeping the user latency and provider costs in check.

To address the aforementioned function configuration challenge, a number of recent works [177][33][108][17] offer solutions that can be utilised to optimise operational cost or performance guarantees for different workloads. While [108] provides a cost-optimised function parallelism for workflows, [33] determines optimal working memory range in a cost-efficient way. On the other hand, Chapter 5 exploits the work-

load characteristics to furnish input-aware function configurations that reduce operational cost and allocated resources. Although existing studies provide independent approaches to recommend or determine the optimal resources based on distinct objectives, they either fail to orchestrate the workload to the desired configuration or do not weigh the benefits of reusing existing versions. Additionally, scholarly works [18][95][86] explore application-specific configurable parameters, such as video frames or model hyperparameters, to propose an optimal value of these settings. However, there exists no single, automated framework that can manage dynamically changing function requirements and simultaneously provide optimisation for both performance and cost while ensuring fault-tolerance. Additionally, the manual effort involved in profiling and setting the resource configuration warrants an adaptive approach to ensure consistent performance and cost efficiency. This research gap highlights the need for an integrated, end-to-end solution that can intelligently orchestrate function requests and resources to meet complex service-level requirements, leading towards *self-driving* serverless platforms. The idea of self-driving platforms is to completely abstract the resource management tasks from the function lifecycle and enable the developers to truly focus on building business logic.

This chapter proposes **Saarthi**, an end-to-end FaaS framework that leverages input-aware resource predictions to adaptively orchestrate incoming workloads, optimise the number of active functions and ensure service continuity by compensating for potential function failures. It introduces a prediction module to determine the input-aware resource requirements of an incoming request. An adaptive request balancer that intelligently re-uses an existing version or cold-starts a new one. Alongside, an independent ILP-based function optimiser is proposed that continuously monitors and maintains a healthy and desired cluster state. Furthermore, a redundancy-based scaling component provides fault-tolerance by compensating for function failures. We build Saarthi atop open-source OpenFaaS [9] serverless framework and offer different modules as standalone services acting in unison. We experiment with a set of functions from standard benchmark suites [166] [107] to evaluate their performance with the Saarthi framework. We further evaluate the impact on operational cost of the functions and compare it with the baseline OpenFaaS platform. Additionally, an ablation study is done with different

components of the Saarthi framework to identify the overheads introduced by individual services and highlight their overall benefit. In doing so, we make the following contributions:

- We propose a hybrid optimisation for request orchestration that combines a heuristic scoring with probabilistic element. It balances between exploration and exploitation to either use immediately available function versions or occasionally cold starting new instances.
- We propose a fault-tolerant request handling with a $G/G/c/K$ buffer to temporarily queue requests and prevent immediate dropping of requests with a retry mechanism to improve system resilience.
- We propose an ILP-based function optimiser that continuously monitors and adjusts the number of function instances, while adapting to dynamic workloads and operational costs.
- We employ a redundancy-based, fault tolerant function scaling mechanism to ensure service continuity and faster service recovery by mitigating mis-predictions and instance failures.
- We incorporate the discussed components into *Saarthi*, a complete framework to automate and integrate multiple aspects of serverless function management, and evaluate its performance against the baseline OpenFaaS using the established benchmark functions [107][166].

## 6.2 Motivation

The motivation for building Saarthi as a complete function lifecycle manager stems from the inherent shortcomings of the current FaaS platforms, which generally fail to deliver the operational simplicity and efficiency of serverless. Although FaaS introduces certain benefits, these platforms still exhibit critical limitations that lead to inconsistent performance, high operational cost and significant developer overhead. Additionally, these inefficiencies result in resource fragmentation, sub-optimal resource allocation and

scheduling, and high maintenance costs for providers. One of the shortcomings is the workload-agnostic nature of most FaaS platforms [53]. They treat every function invocation in the same way, regardless of the distinct workload characteristics such as input parameters and set a static, one-size-fits-all resource configuration. Existing studies [33][18] discuss the effect of workload characteristics, in addition to allocating proportional CPU, network bandwidth and other compute resources to memory on function performance and highlight the complex relation between input payload and overall performance. Therefore, a configuration optimised for one type or size of input will either over-provision the resources for a simple request, leading to economic waste, or under-provision for a complex request causing a degraded performance and execution failure in the worst case. Fig. 6.1 demonstrates the performance of a compute-intensive function, *linpack*, from [166] and plots its memory utilisation and execution time for various input payloads. Fig. 6.1 (left) shows that memory utilisation of a function varies with input payload values at different memory settings and consequently affects the execution time (right). Thus, a memory configuration of 640 MB that ensures a successful execution of input payload, $n$ (number of linear equations to solve), up to 6000, results in failed execution for larger inputs, Fig. 6.1 (right). On the other hand, setting the memory configuration to 3008 MB leads to highly under-utilised resources for smaller inputs, however, it improves the execution duration of the function.

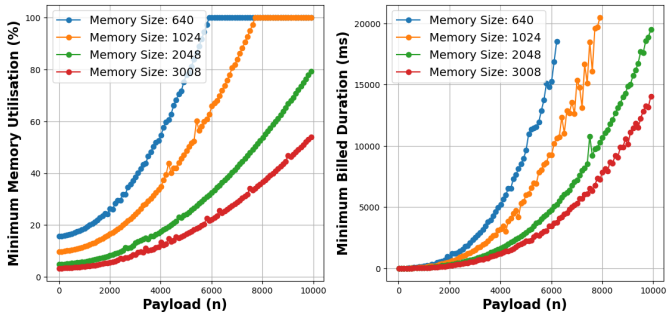



**Figure 6.1:** Comparison of Input Payload vs Memory Utilisation (left) and Billed Execution Duration (right).

Another limitation of current FaaS platforms is the lack of service-level guarantees for performance metrics such as latency or throughput [108][45]. Although providers such as AWS and Azure typically cover infrastructure availability and reliability [178],

they do not explicitly ensure function SLOs. The lack of predictable performance makes it difficult for developers to build and deploy latency-sensitive applications. In Fig. 6.1 (right), if we consider the execution time SLO as 5 seconds, then for some invocations i.e., smaller inputs ($n \leq 4000$), 640 MB memory setting would suffice, however, larger memory configurations such as 2048 MB or 3008 MB would be required for other invocations to successfully execute within the threshold. In addition to this, current platforms do not guarantee that functions run within a specified cost budget [123][93], while meeting execution time thresholds. FaaS billing creates a critical trade-off where a function with a larger, more expensive memory setting might run for a shorter time, potentially resulting in a lower overall cost than a function with a smaller, cheaper setting that executes for longer. This forces developers to take on the heavy burden of manual profiling to find the 'right' resource configuration, a time-consuming and often inaccurate process. Consequently, this overhead defeats the purpose of the serverless paradigm, which is to abstract away such execution complexities and avoid developer overheads. This highlights the need for an intelligent framework that can navigate these complex, often conflicting objectives.

The challenge of resource configuration in FaaS is equally pressing for CSP. When users manually specify resource allocations, often by overestimating to avoid timeouts, or underestimating to save on perceived costs, it leads to suboptimal utilisation across the provider's shared infrastructure. Such mis-configurations cause *resource fragmentation*, making efficient bin-packing and function scheduling difficult at scale [2][83]. This wastage of physical resources reduces overall system efficiency, drives up operational costs, and complicates performance prediction and service quality assurance. In particular, the hard-to-predict resource requirements lead to design trade-offs for startup speeds, hardware isolation and cost efficiency in the cloud [71]. Providers are thus motivated to move beyond the developer-driven model towards automated resource management, which optimises infrastructure use while safeguarding SLOs for all tenants. Following the idea of tailored resource configuration for incoming function input, the frequent start-up of distinct function versions adds further complexity to the fragmentation challenge. This is because these versions, coupled with the dynamic idle timeout for each function, can lead to numerous simultaneously active or idle instances. Given

this, it makes little sense to always create a new instance based on every workload's resource requirement. Instead, it is warranted to first check for any existing, idle version that can potentially fulfil the SLO, and use that instance to avoid a cold start and save provider costs while maintaining the defined SLOs. This highlights the need for a trade-off approach that can consider this overhead and make appropriate, informed decisions.

Furthermore, a service provider must also consider the function concurrency limits and instance count for individual versions. Typically, for platforms like OpenFaaS, function concurrency or the number of concurrent requests a single instance can serve, plays a critical role in resource consumption and utilisation [179]. The rationale here is that instead of creating many small instances to serve diverse workloads, it may be more efficient to consolidate these onto a smaller number of larger, more capable instances that can serve multiple workload types within the same SLO limits. This gives the provider greater control over resource fragmentation, utilisation, and allocation. Therefore, to fulfil both cost and performance goals, the platform needs a multi-objective optimisation approach that can proactively understand the workload and maintain the right number of function instances that complement the platform's ability to autoscale.

These limitations highlight a critical research gap: the absence of a single, end-to-end framework that can holistically and intelligently manage the entire function lifecycle. An effective solution must be able to autonomously adapt to dynamically changing function requirements, balance the competing objectives of performance and cost, and ensure fault-tolerance without manual intervention. Saarthi is designed to fill this gap, moving beyond basic function execution to provide a comprehensive, self-driving serverless solution.

## 6.3 Saarthi System Architecture

The core of Saarthi is built on top of Kubernetes-based OpenFaaS [9] framework, whose two critical components Gateway and Kubernetes provider (*faas-netes*) ensure the serverless experience. Saarthi introduces a number of loosely coupled independent components, Fig. 6.2, to provide a self-driving serverless platform where input-aware resource

configuration decisions are automatically handled for the workload.

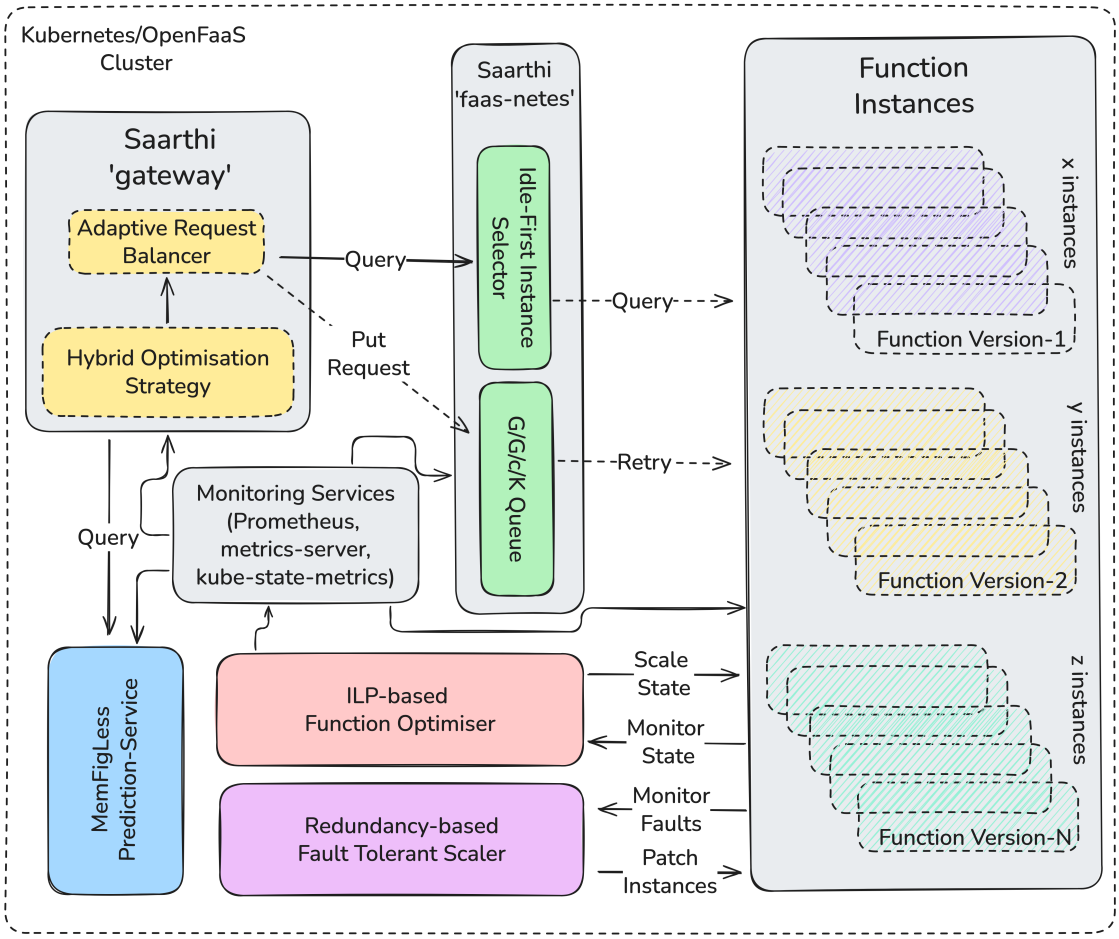



**Figure 6.2:** Saarthi System Architecture

### 6.3.1 Saarthi Request Lifecycle: A Conceptual Overview

To understand how Saarthi works, it's helpful to follow the journey of a single function request. When a request arrives, the system first utilises a Prediction Service to determine the function's resource needs based on the request input/payload. This information is then passed to an Adaptive Request Balancer, which acts as the system's primary decision-maker. The balancer makes a crucial choice: should it reuse an existing, *warm* function instance or should it create a new one to handle the request? This decision is informed by a hybrid optimisation strategy. If the balancer is unable to im-

mediately find a suitable instance, it places the request in a queue to prevent immediate failure. While the balancer handles individual requests, a separate ILP-based Optimisation Engine continuously monitors the entire cluster to manage the overall pool of function instances to meet long-term performance and cost objectives. Finally, a proactive Fault-Tolerance Mechanism acts as a safety guardrail, detecting and compensating for failures to ensure the service remains available and reliable. By orchestrating the flow of requests and resources through these interconnected components, Saarthi automates the entire function lifecycle.

### 6.3.2 Input-Aware Prediction Service

Saarthi's goal of creating a self-driving serverless platform builds on a deep understanding of workload characteristics. As established by prior research [17][20], a function's performance is critically tied to its resource configuration, which in turn is heavily influenced by the request input/payload. To enable automated decisions for both request orchestration and resource allocation, Saarthi incorporates a Prediction Service.

This service takes advantage of input-aware ensemble learning models, an online RFR pipeline, adapted from previous chapter, Chapter 5. This allows Saarthi to perform real-time input-aware inference by predicting an incoming request's ($Q$) resource needs, $R_p$ before it is executed. The details of RFR and performance are discussed in Chapter 5. Furthermore, our training workflow supports incremental learning, which means the Prediction Service can be configured with a *refresh interval* to continuously synchronise with and utilise the latest, most accurate models. This ensures that Saarthi remains adaptive and efficient as workload patterns evolve over time.

### 6.3.3 Adaptive Request Balancer and $G/G/c/K$ Queue

Once the platform determines the input-aware resource requirement, $R_p$, of the incoming request $Q$, an *Adaptive Request Balancer* (ARB) comes into play that is integrated into OpenFaaS *Gateway*. Saarthi leverages this component to orchestrate the request to an appropriate function version. This orchestration mechanism is a critical feature, as it highlights a fundamental distinction between OpenFaaS and commercial FaaS platforms

such as AWS Lambda. While these platforms achieve fine-grained, per-request scalability through automated, infrastructure-level provisioning for nearly every incoming event, OpenFaaS remains inherently constrained by the scaling semantics of the underlying Kubernetes architecture. OpenFaaS cannot technically scale for every individual request due to a control-plane overhead, database lock contention in *etcd* and scheduler back-pressure stemming from phenomena like the *thundering herd problem* when many scaling events occur close together. Even in commercial OpenFaaS Pro plans that offer advanced scaling metrics such as capacity mode [179], these advanced approaches still operate atop standard Kubernetes primitives and remain subject to underlying resource and scheduling bottlenecks. As a result, OpenFaaS often exhibit inefficient resource utilisation and can demonstrate suboptimal performance. For example, it could lead to delayed autoscale reactions or an inability to immediately allocate capacity for spikes of concurrent requests [180] as compared to managed solutions like AWS Lambda.

To overcome these limitations, Saarthi's ARB employs a **hybrid optimisation strategy** that balances exploration and exploitation of the current function instances, $F$, based on the acquired knowledge of resource requirements, $R_p$. First, a heuristic scoring algorithm identifies the most efficient function version to handle the request input, typically preferring a version with predicted configuration, $f_{exact}$ or the smallest version with sufficient resources among the other available versions, $F_{available}$. This is achieved by calculating a score for each function version based on the difference between its resources and the predicted requirements. This is a critical feature as Saarthi's input-aware design often deploys multiple function versions to handle different workload characteristics. Therefore, this process efficiently *exploits* already existing ('warm') function versions by prioritising their reuse over creating new ones, leading to reduced cold-starts. However, to avoid a long-term sub-optimal state, the system occasionally introduces a probabilistic decision-making step with a pre-defined tolerance (e.g., 20%) to *explore* cold starting an instance with the predicted resources. This strategy improves upon both the standard OpenFaaS community/professional autoscaling [181], which often relies solely on trailing metrics like requests-per-second (RPS) or simple in-flight request counts, and naive greedy schemes which can rapidly overload available functions, leading to queuing, timeouts, or overloaded instances. In contrast, the probabilistic exploration path

proactively cold-starts new instances even in the absence of immediate pressure, ensuring balanced distribution and adaptiveness for real workload surges. By doing so, Saarthi mitigates the risk of local minima where requests are routed only to a saturated set of function instances, a frequent challenge faced by both baseline OpenFaaS scaling and conventional heuristics. An overview of request orchestration is demonstrated in Algorithm 2.

**Algorithm 2** Saarthi Adaptive Request Balancer

**Require:** Incoming request $Q$ for function $S_{name}$, predicted configuration $R_p$

**Ensure:** Reuse or start a function version

Define $F$ as the set of all existing function instances.
**/* Tier 1: Find an idle instance of the exact service */**
$f_{exact} \leftarrow$ an idle instance $p_i^{R_p}$ of exact configuration
**if** $f_{exact}$ exists **then**
    Route request to function version $S_{name}^{R_p}$
    **return**
**/* Tier 2: Hybrid decision for resource reuse or cold start */**
$F_{available} \leftarrow$ find all available alternative function versions
**if** $F_{available} \neq \emptyset$ **then**
    $f_{best} \leftarrow$ select the instance from $F_{available}$ with the lowest $S(f)$
    $S_{best} \leftarrow S(f_{best})$
    $S_{CS} \leftarrow$ random score from 20% window of $S_{best}$
    **if** $S_{CS} \leq S_{best}$ **then**
**/* Explore: Cold start is better */**
        Deploy new function version $S_{name}^{R_p}$
        Route request to new version $S_{name}^{R_p}$
        **return**
    **else**
**/* Exploit: Use the existing best version */**
        Route request to function version of $f_{best}$
        **return**
**/* Tier 3: No idle instances or alternatives found - fallback */**
**/* Optional: Log or fallback to default orchestration */**

Consequently, the balancer's high-level decision to forward a request to a function version, $S_{name}^{R_p}$, is based on the availability of its idle instances, $p_i^{R_p} \in \{1 \dots P^{R_p}\}$. The rationale for balancing between versions arise from the fact that the platform initially determines the input-aware resource configuration, leading to simultaneously active

distinct function versions. This approach reduces the decision overhead at the Gateway, allowing it to remain decoupled from the Kubernetes provider-level logic and allowing it to manage concurrent requests more effectively. The final selection of a specific function instance is delegated to the *faas-netes* component which employs an **idle-first instance** selection approach to serve the request. An instance, $p_i^*$ is considered idle if its concurrently active requests, $C_p$ is less than its configured function concurrency, $M_p$. The idle-first heuristic operates on two-stages where it first identifies all the idle instances for a function version and then attempts to claim one of them. In doing so, the system atomically adjusts the active connections counter of the selected instance, $p_i^*$ via *optimistic locking* mechanism where the request is only routed upon successful claim. The system re-tries with an updated set of idle instances if the claim fails due to a race condition and queues it if no idle instances can be claimed after a pre-defined number of re-tries.

When multiple requests simultaneously reach the same function version or if the system is unable to find an idle instance, a **G/G/c/K queue** is used to avoid immediate failures. The queue represents a generalised system where the request arrival rate, $G$ and processing rate, $G$ is unpredictable while accounting for a finite number of function instances, $c$ and a limited buffer size, $K$. The queuing of requests also align with the best-effort execution model, as some instances of a respective function version may become idle by the time a request is re-tried from the queue. This adaptive approach not only maximises resource utilisation and keeps latency low within a standard Kubernetes setup, but it also adapts to changing workloads. In doing so, it attempts to provide a robust request handling without being restricted by the typical limitations of Kubernetes-based FaaS.

### 6.3.4 ILP-Based Optimisation Engine

While the ARB targets the immediate handling of incoming requests, the system must manage the overall operational capacity to meet its long-term performance and cost objectives. To address this, Saarthi introduces an *ILP-based Optimisation Engine* to maintain the required number of function instances. A multi-objective ILP optimisation [20]

emerges as a natural choice for this task for several key reasons. First, it provides a robust mathematical model to optimise contrasting objectives, such as minimising cost while maximising performance [182]. Second, the decision variables in the model are inherently integers [183], as they represent the number of function instances, $x_{f_v}$ and the number of requests to be assigned, $y^r_{f_v}$ (used for internal calculation). Lastly, and crucially, the objective functions and all constraints, including operational cost, execution time, and throughput, can be accurately modelled as a linear function of these integer decision variables [93].

The optimisation engine runs as a standalone component, executing at a configurable interval (e.g., every minute) to ensure the long-term health of the function cluster. It complements the per-request decisions of ARB by taking a comprehensive, cluster-wide view of resources. By default, it follows a greedy approach, monitoring the workload from the last interval and making decisions based on this historical knowledge. This ensures a fast and efficient response to immediate changes. We formulate the objective of maintaining an optimal number of function instances, maximising throughput while minimising operational cost, as a multi-objective global optimisation problem, as follows:

$$\min \Big[ \sum_{f_v} \alpha \cdot x_{fv} \cdot cost_{fv} + \sum_{r} \beta \cdot (demand_r - served_r) \cdot penalty_r - \sum_{r} \gamma \cdot served_r \cdot utility_r \Big] \quad (6.1)$$

where $\alpha, \beta, \gamma$ are cost and utility parameters. To achieve its objectives, the component runs an ILP optimisation loop that calculates a combination of three distinct elements. First is the operational cost of all deployed instances ($\sum_{f_v} x_{f_v} \cdot cost_{f_v}$) of function version $f_v \in \{f_1 \dots f_n\}$ with execution cost $cost_{f_v}$. Second is a penalty for an unserved demand, i.e., $\sum_r (demand_r - served_r) \cdot penalty_r$ that models the cost of service failure, such as a missed service level objective thresholds for requests $r \in demand_r$ of a specific configuration. Finally, it accounts for the value (or utility) gained from every served request ($\sum_r served_r \cdot utility_r$), which motivates the optimiser to prioritise high-

utility tasks. To arrive at a solution, the optimiser uses decision variables to determine the optimal number of instances, $x^*_{f_v}$ to deploy or maintain for each function version and how to assign incoming requests $y^r_{f_v} \leq demand_r$ to them. These decisions are governed by a set of constraints that ensure the total CPU ($\sum x_{f_v} \cdot cpu_{f_v} \leq C_{cpu}$) and memory ($\sum x_{f_v} \cdot mem_{f_v} \leq C_{mem}$) used do not exceed the cluster's capacity. Additionally, it ensures that the number of requests assigned to any instance does not exceed its processing capacity $y^r_{f_v} \leq M_{f_v}$, i.e., the function concurrency. The optimiser's intelligence is particularly evident in its ability to handle trade-off scenarios, such as determining when a few larger function versions are more beneficial than maintaining numerous smaller ones. This strategy not only improves resource efficiency but also reduces resource fragmentation across the cluster. Furthermore, the optimiser is constrained by end-user limits as it conforms to a best-effort execution model when these constraints are exceeded. This ensures that the system performs the best possible optimisation within its given limits, prioritising overall system stability.

**Algorithm 3** Fault-Tolerant Redundancy Algorithm

**Require:** Set of functions $F$, Cooldown period $T_{cooldown}$

**Ensure:** Scaling actions for each function

1: **for all** $f_i \in F$ **do**
2: **if** Time since last scale action for $f_i < T_{cooldown}$ **then**
3: Continue to next function
4: failingPods $\leftarrow$ Count pods of $f_i$ with status $\in \{OOMKilled, CrashLoopBackOff\}$
5: **if** failingPods $> 0$ **then**
6: currentReplicas $\leftarrow$ Get current replicas for $f_i$
7: desiredReplicas $\leftarrow$ currentReplicas + failingPods
8: Scale $f_i$ to desiredReplicas
9: Record current time as last scale action for $f_i$

### 6.3.5 Fault-Tolerant Redundancy Mechanism

As discussed in Sec. 6.3.3, a function instance can be configured with concurrency, $M_p$, meaning it can serve multiple requests simultaneously by sharing its resources. The

system primarily determines the function version based on predicted resource requirements from the adapted models, Sec. 6.3.2, to route the incoming request, $Q$. However, when a mis-prediction happens and a request is directed to an under-provisioned function version, it can lead to a function failure. This has a cascading effect where other active requests are also terminated due to the instance's unavailability, overloading other available function instances with incoming workload.

To prevent this, Saarthi employs a proactive, fault-tolerant redundancy mechanism, Algorithm 3, that continuously monitors at a configured *interval* for early signs of service degradation $S_{fail}$, such as high memory usage. When a potential failure is detected, this mechanism autonomously compensates for the failing instances by adding additional function units, i.e., a simple *additive scaling strategy*. This rapid, proactive scaling helps to absorb the sudden surge of traffic and prevent a thundering herd problem where a large number of requests overwhelm the remaining, already burdened instances. To ensure stability and prevent unnecessary thrashing of resources, the component also implements a cool-down period, $T_{cooldown}$, which serves as a safeguard against conflicting scaling operations from the optimisation engine.

## 6.4 Performance Evaluation

In this section, we present Saarthi's implementation, along with the experimental setup and component configuration parameters. We discuss our assumptions and perform an experimental study to quantify the contributions of Saarthi's components to function performance. Our findings demonstrate that by working in coherence these components significantly impact key metrics such as operational cost, execution latency, and resource utilisation when compared to baseline OpenFaaS (Community Edition) (OpenFaaS-CE).

### 6.4.1 Implementation and System Setup

We implement the core components of Saarthi as an extension to the OpenFaaS-CE [9], leveraging a multi-language microservices architecture to ensure scalability and modularity. It utilises the underlying Kubernetes (v1.32.8) cluster with an overall capacity

of 68 vCPUs and 288 GB of memory, spread across 6 virtual nodes. For research purpose only, we override the OpenFaaS defaults to enable maximum 50 distinct function deployments scaling up to 100 instances, each and configuring longer timeout of 10 minutes for a function execution. Saarthi's primary components - adaptive request balancer with a hybrid optimisation strategy, coordinating with an idle-first instance selection approach, and a waiting queue, are implemented in *Go*. They are seamlessly integrated into the OpenFaaS *gateway* and the *faas-netes* provider (both configured with 100 millicore vCPU and 200 MiB memory), utilising programming language primitives for high-performance concurrency and thread management. The $G/G/c/k$ queue is highly configurable, allowing for fine-grained control over the frequency and duration of its retry attempts. The redundancy-based fault tolerance logic operates as a lightweight *subroutine* alongside the main *gateway* process to proactively monitor failures at regular configurable intervals. The prediction models, adapted from Chapter 5, are exposed as dedicated API-based *Python* services configured with sufficient resources to provide real-time inference at scale. The optimisation workflow is implemented as two coordinating services, a monitoring engine and an ILP optimiser. The monitoring engine (300 millicore vCPU and 256 MiB memory) is responsible for interacting with services such as *metrics-server*, *kube-state-metrics* and *Prometheus* to collect the current state of the cluster and a *Python*-based ILP optimiser [1] (250 millicore vCPU and 512 MiB memory) that consumes the monitored cluster state and runs a complex ILP optimisation loop to determine the desired state.

We evaluate the performance of Saarthi on a variety of serverless functions from various benchmarking suites [166][107], including CPU/memory intensive (*matmul, linpack, pyaes*), scientific functions (*graph-mst, graph-bfs*) and dynamic website generation (*chameleon*) by conducting an ablation study. The functions are developed in Python v3.12 and deployed using the official *python3-http* template from OpenFaaS. The template utilises OpenFaaS *watchdog* v0.10.7, a critical component to enable concurrent handling of requests. The resource configuration and function payload range is adapted from Chapter 5 and we utilise the AWS Lambda pricing [55] for cost calculations. To compare and validate the performance of proposed components, we analyse the open-

[1] The ILP optimisation is implemented using PuLP [184] library

source Azure function traces [13] and select unique workload patterns associated with different function events corresponding to every deployed function. Additionally, conforming to the *log-normal* distribution of workloads [13], we generate a *log-normally* distributed function input values and utilise *Poisson* distribution for request inter-arrival times to closely match the real-time traffic and application payloads.

All Saarthi components are designed to be highly configurable, allowing for fine-grained control over their behaviour. For instance, the prediction service can be configured with a *refresh interval* (default 2 hours) to actively synchronise with the latest learned models. The optimisation engine exposes a decision-making frequency (default every 1 minute) and allows future extension endpoints for optional services such as workload prediction or pricing. Additionally, it explicitly allows configuration of cold-start trade-offs when optimising function deployments to account for prediction-based workload optimisation, disabled by default due to greedy, history-based workload optimisation. To prevent overwhelming a function instance, we set the function concurrency to 10 and similarly, we fix the $G/G/c/K$ queue length to 10 with a retry interval of 10 milliseconds. This modular design of components further allow us to fine-tune the platform for application specific optimisation, if required.

### 6.4.2 Experiments

To evaluate the performance of our proposed solution, we conducted a comparative performance study with different versions of the Saarthi framework. The objective of this study is to measure the impact of key components on overall function and platform performance. For this study, all results are measured and evaluated against the baseline OpenFaaS-CE with its default settings and a resource configuration of 1769 MiB memory and 1 vCPU, consistent with standard AWS Lambda [42] quotas. We use a consistent set of functions, workloads, and metrics, including execution latency, operational cost, and overall system resource usage for comparison. No tests were performed in isolation to mimic a multi-tenant environment, which demonstrates the practical benefits of our framework's components when they work together in a realistic setting. The previous chapter, Chapter 5, discusses the value of input-based memory configuration, and

therefore, we do not re-iterate those results.

To strengthen Saarthi's position, we compare its performance against two different input-aware versions: Saarthi-MVQ, a version with the core request orchestration and a generalised waiting queue, and Saarthi-MEVQ, which adds the fault-tolerance mechanism. The full Saarthi framework with all components enabled is referred to as Saarthi-MOEVQ in our experiments. We selected different function triggers, such as *http* and *orchestration*, from available traces [13] to cover a variety of real-world usage patterns. Consequently, we conducted the experiments for 120 minutes of workload data corresponding to diverse functions and present our findings.

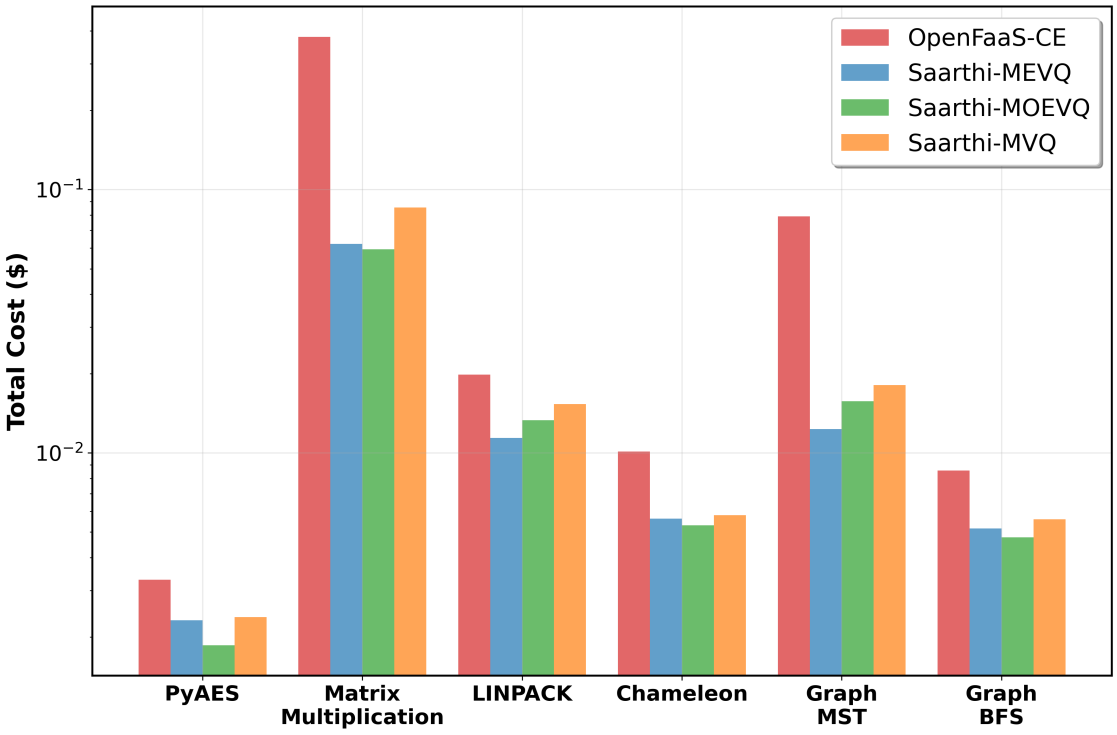



**Figure 6.3:** Operational Cost Comparison of Different Variants

**Results** Our comprehensive experiments across six diverse workloads reveal significant performance improvements and cost optimisations achieved by the Saarthi framework and its variants. We summarise the comparison results in Fig. 6.3-6.7 and focus on execution latency, operational cost, and resource usage across the entire workload. We

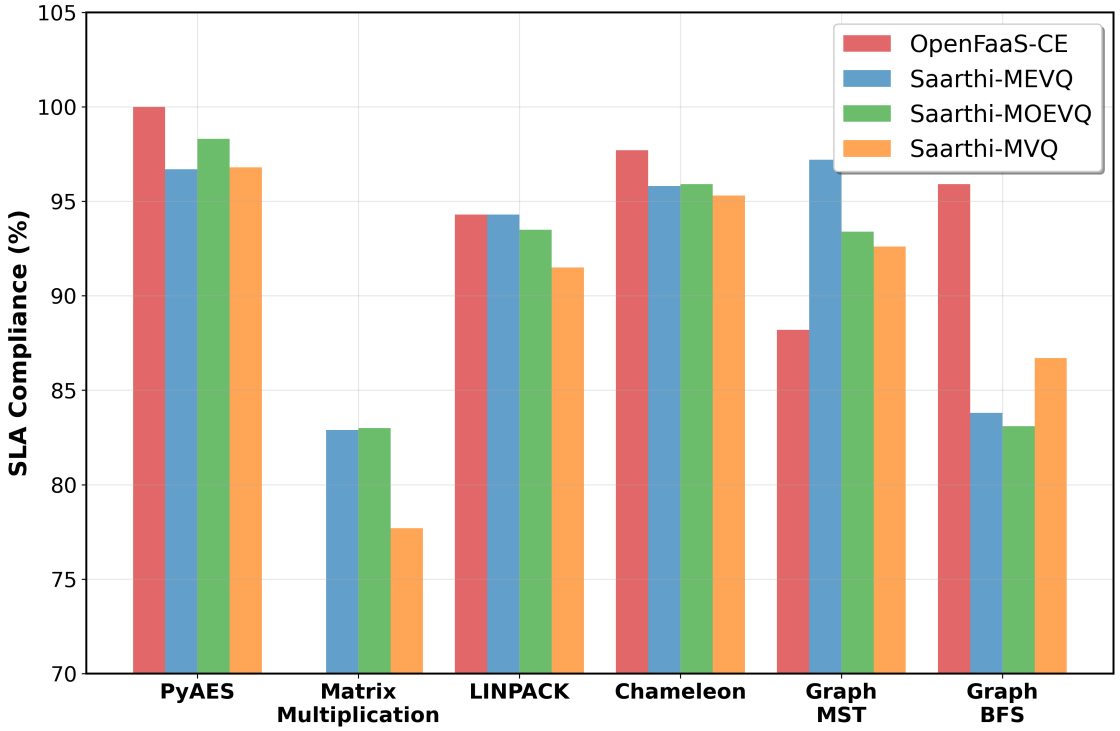



**Figure 6.4:** Execution Time SLA comparison of Saarthi variants as compared to OpenFaaS-CE

observe for *graphmst* with an *orchestration* trigger that Saarthi variants perform better, with Saarthi-MOEVQ and Saarthi-MEVQ achieving up to 93.4% and 97.2% execution latency SLA satisfaction compared to 88.2% for the baseline OpenFaaS-CE (Fig. 6.4). Furthermore, Saarthi-MEVQ successfully serves 0.2% more workload than Saarthi-MOEVQ while saving up to 21.2% of operational costs (Fig. 6.3). This reduction is further magnified for Saarthi-MVQ, which incurs a 1.32x times operational cost while satisfying 92.6% execution time SLA, and costing 1.84x times for OpenFaaS-CE, as compared to the best-performing Saarthi variant. In terms of resource usage, Saarthi-MOEVQ leverages 11 different function configurations compared to 14 by Saarthi-MEVQ (Fig. 6.6, which introduces more unique instances during the experiment due to its optimisation-based instance management, Fig. 6.7. Similar results are observed for *graphbfs* where Saarthi-MEVQ satisfies 0.7% more execution SLA as compared to Saarthi-MOEVQ with 7.2% extra operational cost. Additionally, Saarthi-MOEVQ explores 18% less function configurations while exploiting 36% more unique instances. Contrastingly, for *chameleon*

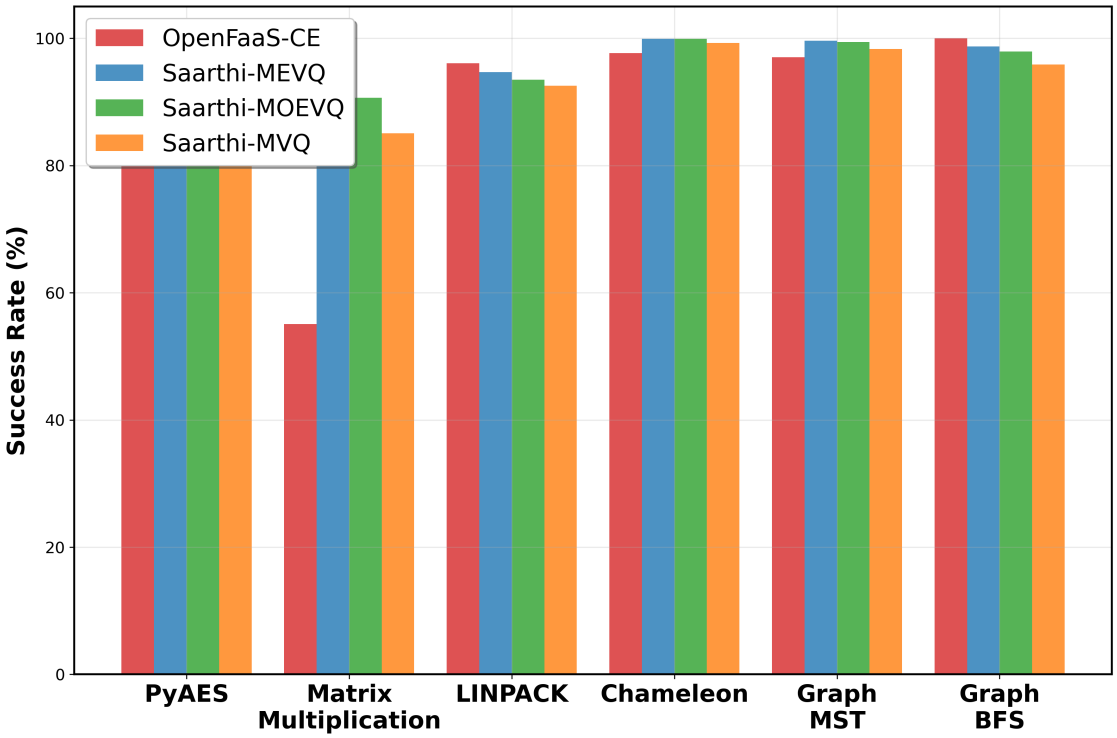



**Figure 6.5:** Request Success Rate Analysis per Workload for OpenFaaS-CE and Saarthi variants

with an *http* trigger, OpenFaaS-CE is unable to handle a burst of requests, resulting in 96% more failures, Fig. 6.5 and a 1.47x and 1.3x times more expensive execution than Saarthi-MOEVQ and Saarthi-MEVQ, respectively. However, OpenFaaS-CE is able to satisfy 1.18x times more execution time SLA out of successfully served requests due to its over-provisioned resources. OpenFaaS-CE did not scale any instances, but Saarthi-MOEVQ utilised 4 different versions while using 30 unique instances over the experiments.

For resource-intensive functions such as *matmul* and *pyaes*, Saarthi outperforms the baseline with a larger margin. For *matmul*, Saarthi-MOEVQ achieves 83% execution time SLA satisfaction under a heavy and bursty workload while OpenFaaS-CE struggles to keep up with the incoming requests, serving only around 42%, Fig. 6.5. Additionally, the operational cost surpasses up to 84% for OpenFaaS-CE as compared to both Saarthi-MOEVQ and Saarthi-MEVQ. This further highlights the advantage of the

proposed optimisations where up to 24 unique instances are used across 7 different versions by Saarthi-MOEVQ while scaling 3 of them. This exploration is 36% less for Saarthi-MOEVQ as compared to Saarthi-MEVQ, where both account for at most 3 function version failures. In the case of *pyaes*, even though Saarthi-MOEVQ satisfies 1.7% less execution time SLA, the trade-off between cost savings and SLA are higher with savings of up to 43% as compared to baseline OpenFaaS. Consequently, Saarthi variants are able to explore up to 8 different input-aware function versions, using up to 25 unique instances across the test for Saarthi-MOEVQ. Similar results are achieved for *linpack*, where the baseline and Saarthi variant satisfy up to 94.3% execution time SLA with a reduction of operational costs of up to 42%. In this case, Saarthi-MEVQ outperforms Saarthi-MOEVQ by 0.8% with a cost savings of up to 14.3%, however, exploring 30% more function versions and 20% less unique instances. Therefore, we observe a superior overall performance (normalised weighted sum of SLA, cost and success rate), Fig. 6.8, of Saarthi variants as compared to OpenFaaS-CE, with Saarthi-MOEVQ leading at a score of 0.812.

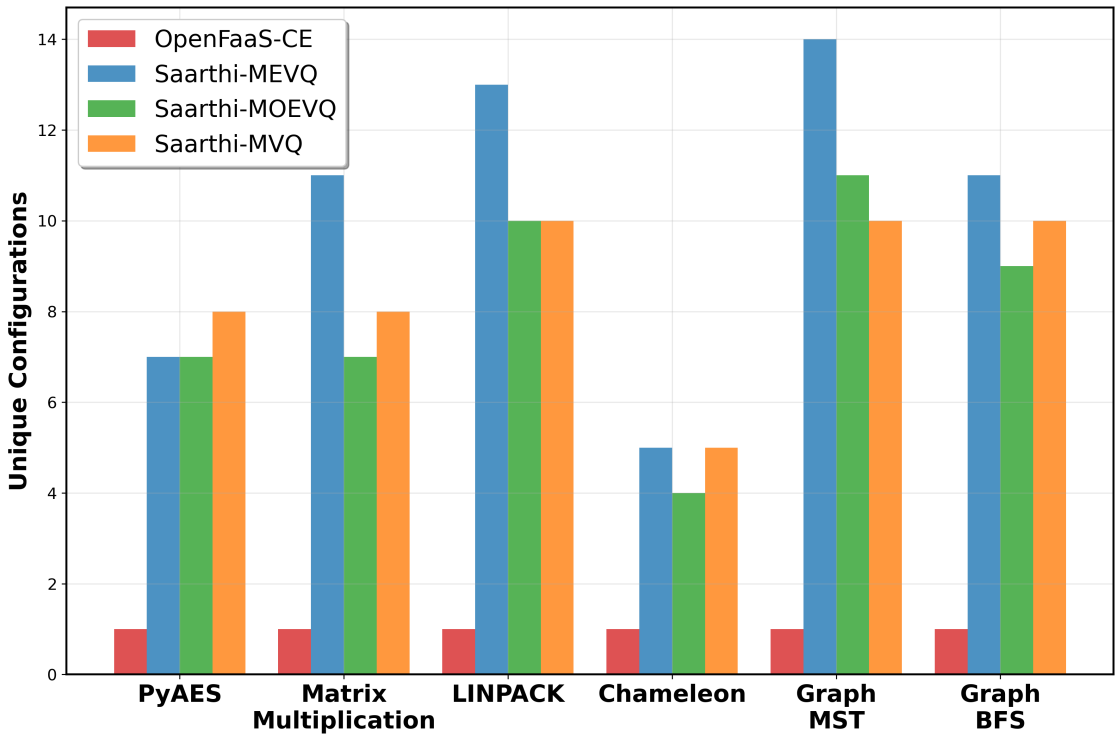



**Figure 6.6:** Unique Configurations used by OpenFaaS-CE and Saarthi variants

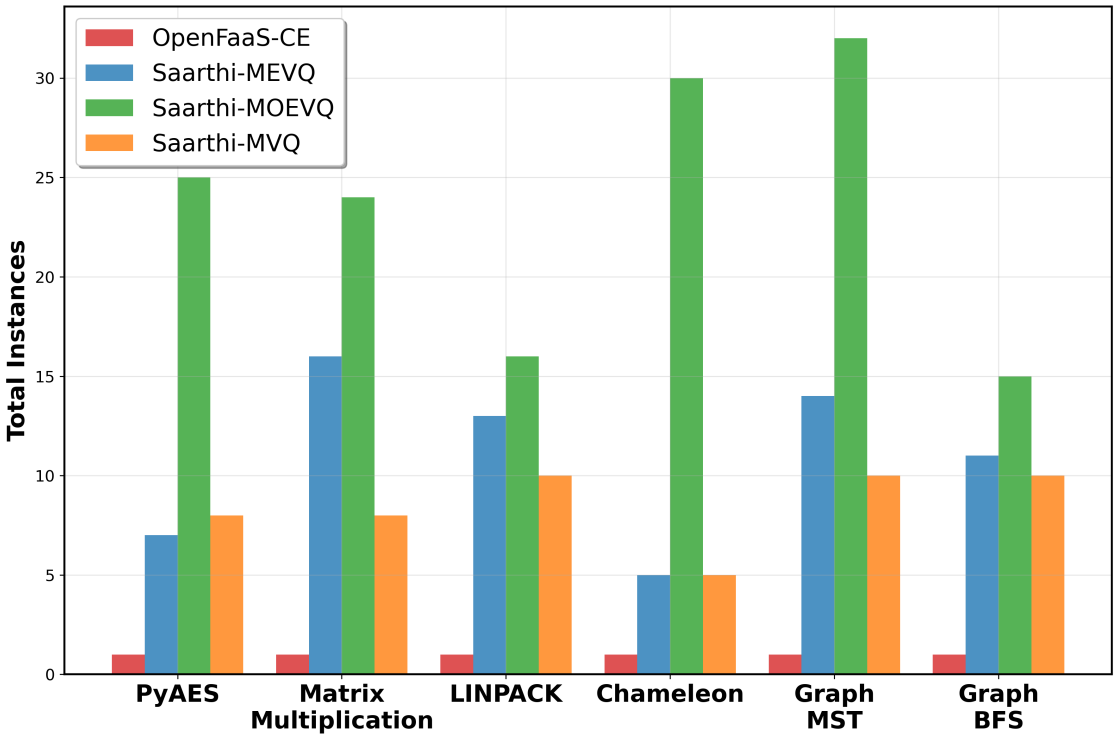



**Figure 6.7:** Total Instances used by OpenFaaS-CE and Saarthi variants across experiments

**Overhead Analysis** In addition to the performance boost that Saarthi provides over the baseline, the overheads introduced by different components are minimal and comparable to the short execution of functions. The critical strategies of various components, such as the adaptive request balancer, optimisation engine, and fault-tolerance mechanism, are input-aware predictions, ILP-based cluster optimisation, finding alternate and idle function versions suitable for successful execution, and applying scaling decisions. The prediction service has an overhead of 0.1 seconds for a unique inference that is followed by 0.1 milliseconds for cached inferences. Once the resource requirement is ascertained, the adaptive balancer incurs an overhead of an average of 40 milliseconds to determine available idle function instances or any alternate versions. In doing so, it might explore the possibility of a cold start, and applying that decision takes 0.2 seconds (part of the OpenFaaS-CE overheads).Similarly, the redundancy-based fault-tolerance mechanism also experiences an overhead of 0.2 seconds to apply its decisions. The in-

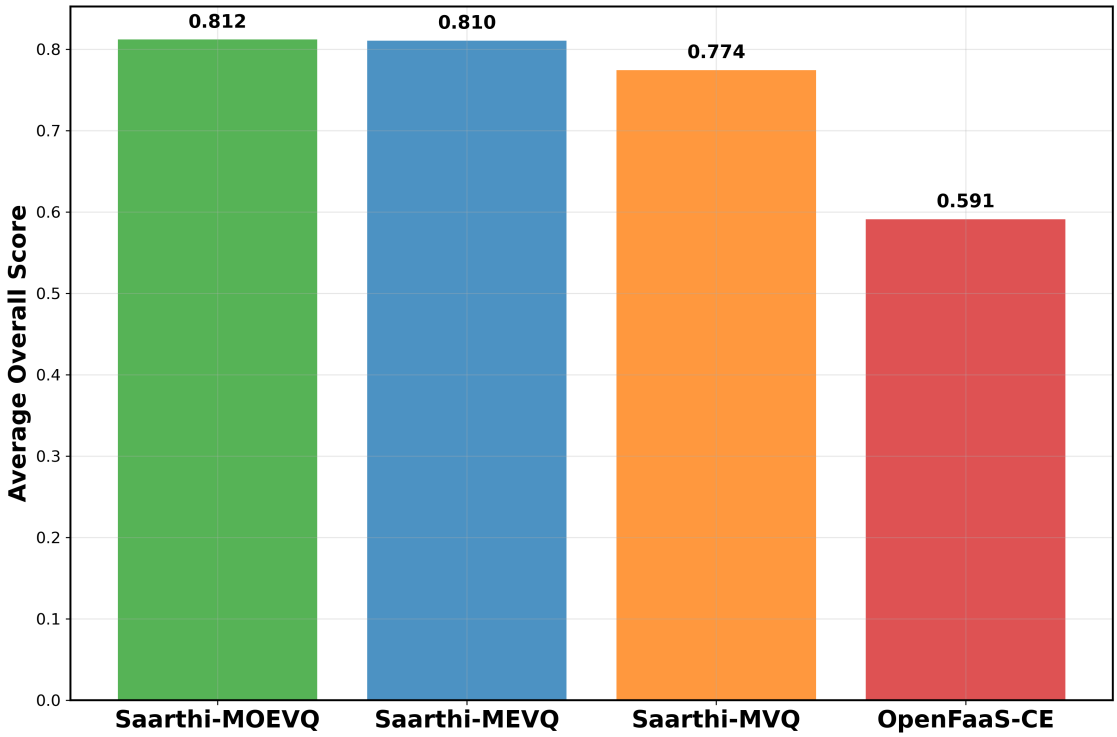



**Figure 6.8:** Average Overall Performance Score of OpenFaaS-CE and Saarthi variants

dependent ILP optimisation service that executes every minute to balance the cluster state takes 1.45 seconds to run a single optimisation loop, and the decisions are applied based on the overheads incurred by the base OpenFaaS backend. However, the cold-start overhead introduced by the Kubernetes in our setup varies between 2 to 6 seconds, depending on the cached function image at the underlying nodes.

### 6.4.3 Discussion and Lessons

Based on the results we have observed, we can clearly realise the benefit of Saarthi's components and its input-aware approach. However, the performance variation between Saarthi-MOEVQ and Saarthi-MEVQ highlights a crucial trade-off. The Saarthi-MOEVQ variant, with its ILP optimisation engine, proactively seeks an optimal state, which can sometimes result in cold-starts when a non-existent function version is predicted. As discussed, the cold start overheads may vary from 2 seconds to 6 seconds, this not only leads to a higher rate of failed requests but also contributes to increased ex-

ecution times, thus reducing the overall SLA compliance. However, these performance drawbacks can be addressed by tuning the platform for specific applications and by focusing on improvements to cold-start times. On the other hand, Saarthi-MEVQ does not employ a scale-down mechanism and continues to operate with a larger number of available function instances, which were added to compensate for earlier failures. This strategy ensures higher reliability and SLA compliance at the expense of operational cost. Conversely, in cases where OpenFaaS-CE is able to serve more traffic and achieve higher SLA rates, this is often attributed to its static configuration and over-provisioned resources. This results in faster and more SLA compliant executions but at a significantly higher and often unnecessary cost. Our comprehensive experiments across diverse workloads reveal significant performance improvements and cost optimisations achieved by the Saarthi framework and its variants. For compute-intensive functions like *matmul* and *pyaes*, Saarthi-MOEVQ already performs above par, demonstrating the clear benefits of its optimisation engine. An improvement in cold-start times would further benefit Saarthi-MOEVQ's performance for other lightweight functions, making its performance advantages more widespread across a variety of workloads.

Our implementation decisions were guided by a series of deliberate trade-offs to ensure the framework's effectiveness within a Kubernetes-based environment. The hybrid orchestrator, for instance, employs a simple, difference-based scoring heuristic to maintain fast decision-making. The exploration probability for cold-starts was set at 20% to avoid being overly restrictive while still allowing for the discovery of a better long-term state. To prevent resource thrashing, the redundancy-based fault-tolerance approach was configured with a 30-second cooldown period, running checks every 15 seconds to ensure a balance between responsiveness and stability. The ILP optimiser's decisions are constrained by a configurable average function throughput of 10 requests per minute and a pricing scheme based on AWS pricing [55]. To match the behaviour of OpenFaaS-CE, the optimiser does not scale down to zero.

The experimental setup was carefully designed to provide a realistic evaluation. The 2-hour test duration was selected to be consistent with the 2-hour model refresh window of our prediction service, MemFigLess. The request queue within the *faas-netes* provider was configured with a capacity of 10 requests and a 10 milliseconds wait time, aligning

with the provider's native retry behaviour. We also accounted for the inherent overheads of the underlying platform in ILP optimisation cycles, for example, a cold start takes between 2-6 seconds, depending on whether the image is cached. We found that despite our system's added complexity, the overheads are minimal and acceptable. The latency added by our components typically adds at most 0.2 seconds to the critical path of a cold start. This minimal overhead is a worthwhile trade-off given the substantial cost savings and SLA improvements observed across the board.

## 6.5 Related Work

A number of existing studies have explored the resource configuration aspect in FaaS for both multi- and single function applications, however, focusing on distinct tunable parameters and objectives. The researchers [105] employ various search methods such as linear or binary search to find the right function memory configuration optimised for operational cost and execution time. Another work [108] proposes to balance the inter- and intra- function parallelism while achieving the workflow SLO. Alternatively, studies [95][86][96] explore the video processing and analytics pipeline, where they discuss the criticality of function inputs and propose configuration tuning frameworks to find optimal or near optimal configurations to improve the processing time and reducing cost. Taking a different approach, [18] attempts to reclaim the extra function memory to maintain a data cache based on the input-aware resource prediction. The authors [93] exploit BO to learn the relationship between execution time and memory configurations and reduce sample size. In doing so, they propose a framework to find the near optimal function memory while satisfying customer objectives. Complementary to this, Moghimi et al. [33] presents an online parametric regression technique to recommend the right memory configuration for functions, however, not focusing on the function input. Consequently, Chapter 5 explores RFR learning for predicting input-aware memory settings, and provides a profiling and inference pipeline for online function scheduling while lowering operational cost and resource wastage. Similarly, ChunkFunc [177] also exploits the input-dependent function performance to build performance models and put forward a BO-based profiling pipeline that is leveraged to optimise the memory

**Table 6.1:** Summary of Related Works

| Paper | Focus Area | Key Contribution |
|---|---|---|
| [185] | Random forest regression | Input-aware prediction, online pipeline |
| [33] | Online regression | Memory configuration recommendation, not input-aware |
| [108] | SLO-aware conf. | Balances parallelism for workflow SLO |
| [95], [86], [96] | Video analytics, pipeline tuning | Input-aware configuration, processing time/cost |
| [18] | Opportunistic caching | Input-aware in-memory cache performance |
| [93] | BO | Near-optimal memory configuration, sample-efficient |
| [105] | Memory search (linear/binary) | Optimises function memory for cost and time |
| [177] | Input-dependent performance, BO | Profiling pipeline, workflow memory optimisation |
| [17] | Input-sensitive scheduling | Predictive scaling, batching, resource/energy use |
| [121] | Data-centric scheduling | Expresses data patterns, two-tier orchestration |
| [98] | Sizing, bundling, prewarm | Cold start reduction in Directed Acyclic Graphs (DAG) |
| [120] | Migration, placement | Optimises placement, migration under practical constraints |
| [186] , [118] | Queues, pre-warming, scaling | Reduces cold starts, manages contention |
| **Saarthi (Our Work)** | Routing, Scaling and Scheduling | Optimises request orchestration, Function re-use, Optimises Cluster state |

configurations for serverless workflows. However, none of the discussed works offer an end-to-end framework that manages the function request lifecycle, starting from re-

source prediction to smart orchestration, reducing resource wastage and cold starts to finally optimising the function deployments for determining right balance of instance counts.

In addition to resource configuration works, a significant effort has been spent by the community on function scheduling and placement [87][187], exploiting distinct aspects such as data locality [121], workload interference [21], cold starts [98] and migration [120]. Bhasi et al. [17] introduce input-size sensitive request re-ordering and batching to determine the number of function instances. They propose a set of predictive scaling components that support the request scheduling to improve cluster resource utilisation and energy consumption. Zhao et al. [21] proposes a QoS and performance predictor for co-located services and schedule functions under partial interference. Gunasekaran et al. [186] introduce queues for the incoming requests to reduce the number of function instances and pre-warm instances using LSTM-based prediction models. Similarly, [118] also addresses the resource contention experienced by a diverse set of functions and employ a scheduling and scaling component to regulate the resource usage of instances and prevent cold starts. Pheromone [121] follows a data-centric approach where data drives the function execution in a workflow. They introduce new primitives to express data consumption patterns and function orchestration in order to reduce the data transfer and utilise a two-tier coordination to execute functions. Although, these works address different aspects of function scheduling, none of them directly focus on input- and version-aware function scheduling and management. A summary of related works is listed in Table 6.1.

**Software Availability:** The details of environment setup, function code snippets, and algorithms we implemented atop OpenFaaS can be accessed from: `https://saarthi.siddharthagarwal.net/`

## 6.6 Summary

In this chapter, we presented **Saarthi**, an end-to-end framework designed to address the inherent challenges of resource management in FaaS platforms. This work explores beyond static, developer-driven configurations and simple reactive scaling, demonstrat-

ing a practical approach to building a *self-driving* serverless platform. The core of Saarthi lies in its harmonised components such as an adaptive request balancer that intelligently orchestrates per-request decisions, a proactive ILP-based optimisation engine that manages long-term cluster state, and a resilient fault-tolerant redundancy mechanism that mitigates service failures. Through a comprehensive comparative study, we demonstrated that Saarthi and its variants significantly outperform the baseline OpenFaaS-CE, achieving substantial improvements in key metrics. The hybrid orchestration strategy effectively managed dynamic and bursty workloads, resulting in lower end-to-end latency and higher SLA compliance. Furthermore, the ILP optimiser proved its ability to reduce operational costs and resource fragmentation by making globally optimal decisions. The redundancy mechanism was also shown to be effective at ensuring service continuity by preventing sustained failures.

While Saarthi presents a significant practical effort towards adaptive management of functions, a simulation environment is beneficial in testing various combination of request routing and function placement techniques and their influence on the function performance. Therefore, the next chapter explores a FaaS simulation engine to present an interactive tool to test different policy scenarios.

# Chapter 7

# A FaaS Simulation Engine and Visualiser

*The rapid adoption of serverless computing necessitates a deeper understanding of its underlying operational mechanics, particularly concerning request routing, cold starts, function scaling, and resource management. This chapter proposes Serv-Drishti, an interactive, open-source simulation tool designed to demystify these complex behaviours. Serv-Drishti simulates and visualises the journey of a request through a representative serverless platform, from the API Gateway and intelligent Request Dispatcher to dynamic Function Instances on resource-constrained Compute Nodes. Unlike simple simulators, Serv-Drishti provides a robust framework for comparative analysis. It features configurable platform parameters, multiple request routing and function placement strategies, and a comprehensive failure simulation module. This allows users to not only observe but also rigorously analyse system responses under various loads and fault conditions. The tool generates real-time performance graphs and provides detailed data exports, establishing it as a valuable resource for research, education, and the design analysis of serverless architectures.*

## 7.1 Introduction

Serverless computing has emerged as a transformative paradigm, abstracting away the complexities of infrastructure management and enabling developers to focus solely on their application logic. Major cloud providers such as AWS [6], Google Cloud [8], and Microsoft Azure [19] have made serverless computing a fundamental part of modern

This chapter is derived from:

- **Siddharth Agarwal**, Maria Rodriguez Read, and Rajkumar Buyya, "Serv-Drishti: An Interactive Serverless Function Request Simulation Engine and Visualiser", in *Proceedings of the 35th IEEE International Telecommunications Networks and Applications Conference*, Christchurch, New Zealand, November 26-28, 2025.

cloud-native architecture. However, the very abstraction that makes serverless attractive also introduces a significant challenge: the lack of transparency. The internal mechanisms governing request dispatching, resource provisioning, and dynamic scaling are often treated as a *black-box*. This leaves developers to deal with the unexpected delays of a cold start [188], the complexities of elastic scaling [19], and the nuances of request routing logic [189]. Understanding these behind-the-scenes processes is crucial for optimising serverless application performance, predicting behaviour under load, and designing resilient, cost-effective systems.

Existing approaches [104][190] predominantly focus on performance monitoring, modelling, and deployment, leaving a significant gap for demonstrative and educational instruments that visually explain the dynamic lifecycle of a serverless request. To address this gap, we propose *Serv-Drishti*, an interactive, end-to-end serverless workflow simulation engine and visualiser. By providing a lucid, hands-on experience, our tool demystifies serverless operations for students, researchers, and practitioners. This chapter details the architecture, simulation logic, and features of Serv-Drishti, including its advanced failure simulation, performance analysis, and data export modules, demonstrating its value as a powerful platform for understanding and analysing serverless architectures.

## 7.2 Related Work

The rapid evolution of serverless computing has driven considerable research into modelling, simulation, and performance analysis of FaaS platforms to facilitate deeper understanding. Therefore, simulators are crucial for comprehending scheduling behaviour and resource management decisions in FaaS environments without incurring cloud costs.

CloudSimSC [189] builds on the CloudSim [191] toolkit for serverless environments, modelling detailed resource management and scheduling policies. It delivers robust trace-driven experimentation for evaluating concurrent request handling and scaling strategies, but operates in a non-visual manner. Its outputs are logs and post-processed metrics, making it less suitable for immediate interactive demonstration or pedagogical visualisation of request flows.

DSLab FaaS [192] provides modular, trace-driven FaaS simulation focusing on reproducibility and extensibility. It supports custom plugin components (e.g., schedulers, auto-scalers) and has demonstrated efficient simulations for complex workloads. DSLab FaaS excels at rigorous, repeatable resource management policy evaluation but, like CloudSimSC, does not offer real-time visualisations. Additionally, the workload traces are identified as the source of simulation data and may not support generation of simulated data.

faas-sim [193] is a discrete-event, trace-driven framework built on SimPy, unique for integrating network latency modelling via Ether. faas-sim allows researchers to assess scheduling and autoscaling strategies, especially in distributed or edge environments, but its use of trace analysis and lack of animated visualisation limits accessibility for comparative and pedagogical study.

**Table 7.1:** Comparison of Related Serverless Simulation Frameworks

| **Framework** | **Primary Focus / Contribution** | **Visualisation Approach** | **Key Limitation** |
|---|---|---|---|
| CloudSimSC | Resource management & scaling | None (Log & metric output) | Lacks interactive demonstration |
| DSLab FaaS | Reproducibility & extensibility | None (Trace-driven analysis) | No real-time or visual feedback |
| faas-sim | Network latency modelling | None (Trace-driven analysis) | Limited pedagogical accessibility |
| OpenDC | Datacenter resource allocation | Static (Aggregate metrics/topology) | No dynamic request flow animation |
| ServlessSimPro | Energy consumption monitoring | None (Code-based framework) | Steep learning curve; non-interactive |
| **Serv-Drishti** | Pedagogical visualisation & interactive ”what-if” analysis | Live, dynamic animation of the full request lifecycle | Simplified model for conceptual understanding |

OpenDC [194] is a notable simulation platform for modelling emerging cloud datacenter technologies, including serverless workloads. It provides a valuable environment

for exploring different resource allocation and scheduling policies. While OpenDC offers an interactive web interface for model exploration, its visualisations are primarily focused on aggregate metrics and static topology maps rather than dynamically animated request flow. This key distinction is crucial for understanding the impact of scheduling and routing decisions on individual requests, which is central to analysing system bottlenecks and cold starts.

ServlessSimPro [195], a comprehensive simulation platform, addresses many of the shortcomings of prior simulators. It distinguishes itself by offering a wide range of scheduling strategies, including container migration and reuse, and provides a comprehensive set of performance metrics, notably including the first-ever monitoring of energy consumption. While ServlessSimPro provides a robust, code-based simulation environment, its core utility remains within a backend framework. The reliance on code execution to define experiments and visualise results presents a steep learning curve and limits accessibility for a broad audience of researchers, students, and practitioners. The purely code-based interface also hinders an intuitive, real-time understanding of the temporal dynamics of a serverless request flow.

## 7.3 System Architecture and Simulation Model

The visualiser's architecture abstracts the core components of a serverless platform into a simplified, yet functionally representative, model. The simulation is built around a discrete-event, request-driven model, where each incoming request is treated as a unique entity traversing the system and interacting with various components, triggering state changes and resource consumption events. This approach allows for detailed tracking of individual request latencies and resource usage across the system.

The core of the Serv-Drishti platform is its simulation engine, which models the end-to-end request flow through a serverless environment by abstracting key operational components into a layered architecture. Each component is responsible for a distinct phase of the request lifecycle, accurately reflecting the complexities of real-world FaaS platforms.

The **API Gateway** serves as the initial entry point for all incoming requests, which

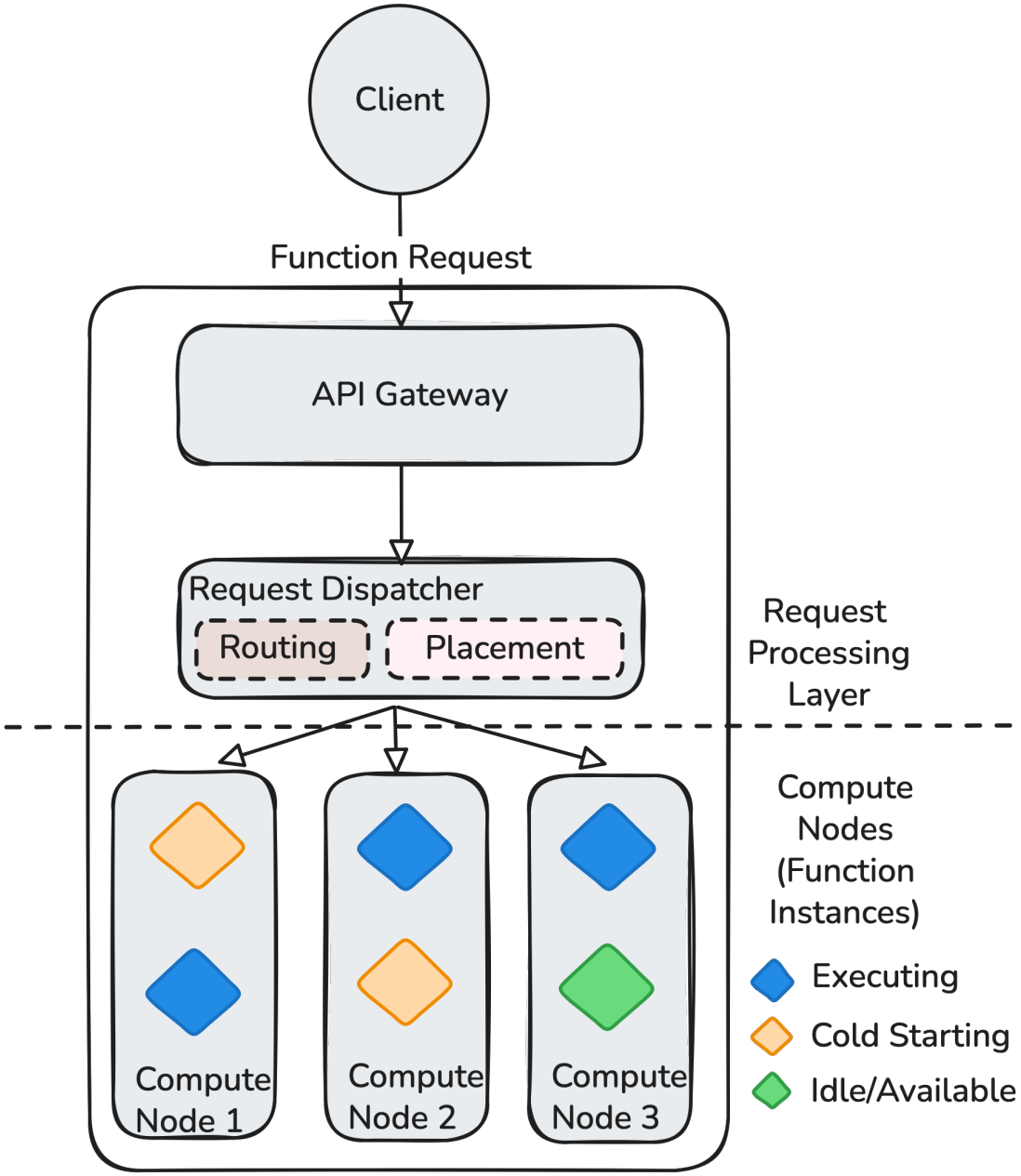



**Figure 7.1:** Request Flow across Serv-Drishti Components

originate from user actions or automated rates. It forwards requests to the **Request Dispatcher**, the central intelligence unit of the simulation. The dispatcher manages an internal FIFO Request Queue for buffering requests that cannot be immediately routed to an available function. It applies configurable routing policies, such as *Warm Priority*,

*Round Robin*, and *Least Connections*, to select the most suitable function instance. When new capacity is needed, the dispatcher triggers auto-scaling and uses a Placement Algorithm to provision new functions on an available Compute Node.

**Compute Nodes** represent the underlying infrastructure that hosts one or more **Function Instances**. They have a configurable, finite capacity of CPU and Memory that is pooled and shared among all the functions they host. Function Instances are the isolated execution environments for the serverless code, capable of processing requests concurrently up to a configurable limit. Each instance consumes a fixed amount of resources from its host node and transitions between states like cold-starting, busy, and active based on system events. This layered architecture allows Serv-Drishti to model the complete request flow and demonstrate the interplay between logical components and physical resources.

## 7.4 Core Simulation Logic and Features

The core of our platform's simulation logic is the interplay between Request Routing Strategies and Function Placement Algorithms. Request routing governs how incoming requests are dispatched to a function, while function placement determines where a function instance is provisioned on the available virtual nodes. Together, these two mechanisms define how workloads are managed, impacting performance, cost, and resource utilisation.

### 7.4.1 Request Routing Strategies

The request dispatcher module implements multiple configurable routing strategies to enable users to analyse and compare their effects. The Warm Priority strategy is designed to minimise latency by prioritising the reuse of active ('warm') function instances. A new request is immediately routed to a warm instance if one is available. If no warm instances exist, the request is queued until an instance becomes ready. This approach is common in real-world FaaS platforms to reduce the overhead of cold starts. The Round Robin algorithm, in contrast, is a simple, stateless method that sequentially distributes

requests among all currently available instances. While it evenly spreads the load, it does not prioritise warm instances, which can lead to more frequent cold starts, particularly during sudden bursts of traffic. A more intelligent, state-aware algorithm is Least Connections, which routes a new request to the function instance with the fewest concurrent requests. This dynamic load-balancing approach prevents any single instance from becoming a bottleneck, aiming to minimise latency by utilising the least burdened resources.

### 7.4.2 Function Placement Algorithms

When a new function instance needs to be created, the system uses a placement algorithm to decide which virtual node will host it. This decision is crucial for optimising resource utilisation, cost, and performance. The First-Fit algorithm places the new instance on the first suitable node found with enough resources. The Best-Fit algorithm selects the node that will have the least remaining capacity after placement, aiming to minimise resource fragmentation. Conversely, the Worst-Fit algorithm places the new instance on the node with the most remaining capacity, leaving room for larger future placements. For balancing the load, the Load-Balanced algorithm places the instance on the node with the lowest average CPU and memory utilisation. The Affinity strategy prefers to place the new instance on a node already hosting a function of the same type to improve resource sharing. Its counterpart, Anti-Affinity, prefers to place the instance on a node that does not host a function of the same type, increasing fault tolerance and isolating workloads. Lastly, the Cost-optimised algorithm is a more complex strategy that seeks to maximise resource utilisation while minimising waste, aiming for the most cost-effective placement decision. A sample comparison of different placement strategies is shown in Fig. 7.2.

### 7.4.3 Function and Compute Node Management

The lifecycle and state of both function instances and compute nodes are critical elements visually represented in the simulation. A function instance dynamically changes colour to indicate its status. It is *Orange* when in the cold start phase, simulating the

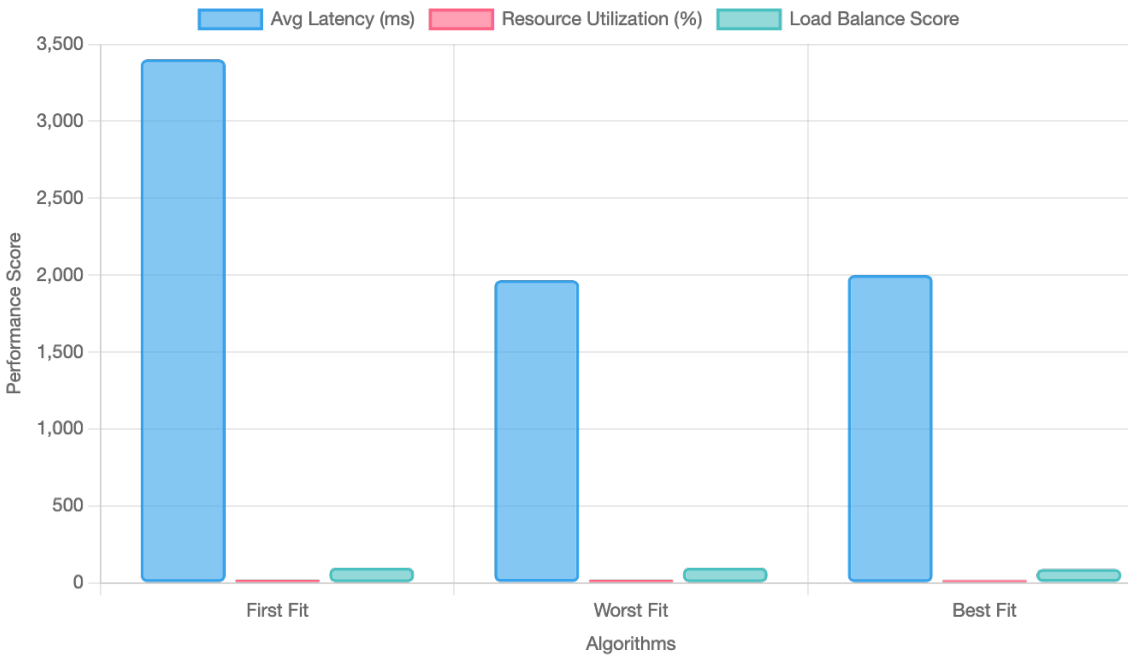



**Figure 7.2:** Different Placement Algorithm Performance Comparison

time required for initialisation, during which it is unavailable for processing. An instance is *Blue* when it is actively processing requests, up to its configurable concurrency limit, and is *Green* when it is available or warm, ready to receive new requests with minimal latency, Fig. 7.1. Compute Nodes represent the underlying physical or virtual machines with a configurable total CPU and memory capacity. A new node is dynamically provisioned by the request dispatcher when existing nodes are at capacity or lack sufficient resources to host new function instances. Each Function Instance consumes a fixed amount of resources, which is deducted from the node's capacity, and the visualiser updates the resource meters in real-time.

### 7.4.4 Scaling Behaviour

Serv-Drishti vividly demonstrates both the scale-up (provisioning) and scale-down (deprovisioning) aspects of elasticity, Fig. 7.3. When demand increases, new function instances are provisioned. If existing compute nodes lack capacity, new nodes are brought online, a process that respects user-defined limits on the maximum number of instances

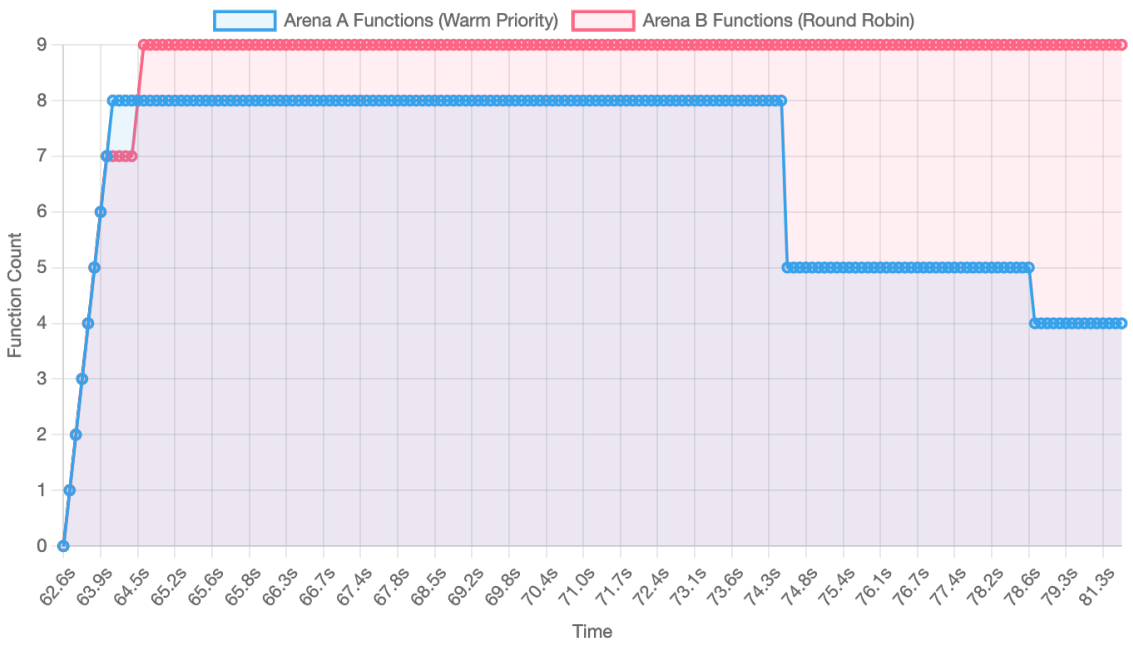



**Figure 7.3:** Function Scaling Over Simulation

and nodes. To optimise costs, idle function instances and compute nodes are automatically de-provisioned after a configurable Inactivity Timeout. This simulates the pay-as-you-go model by reclaiming idle resources when they are no longer in use.

### 7.4.5 Visualisation and Interaction

The platform's core strength is its interactive virtualisation, which provides an intuitive understanding of the simulated environment. The request traversal through the system is animated, providing a clear visual of their journey and highlighting potential bottlenecks. The request dispatcher also prominently displays a visual queue, offering immediate feedback on system load and back-pressure. A user-friendly, collapsible User-Interface (UI) panel allows for real-time adjustments of key simulation parameters, enabling *what-if* scenario analysis and fostering experimental learning in a risk-free environment.

## 7.5 Critical System Considerations

The design of Serv-Drishti is based on several key considerations aimed at balancing pedagogical value with a representative, yet simplified, simulation of serverless operations. This section delves into these design philosophies and the implications of our approach.

### 7.5.1 Abstraction versus Fidelity

A fundamental design decision was to create an abstract model rather than a high-fidelity replica of a specific cloud provider's implementation. Real-world FaaS platforms involve immense complexity, including multi-tenant scheduling and intricate networking, which would render the simulator overly complex and difficult for learners to grasp. Instead, Serv-Drishti abstracts away the underlying hardware to highlight the primary logical interactions: request queuing, dispatching, cold starts, concurrent execution, and dynamic scaling. For example, compute nodes abstract the physical infrastructure, while function instances represent the isolated execution environment. This simplification is intentional, as it allows users to focus on fundamental principles of serverless elasticity and resource management, providing a conceptual understanding that is transferable across different FaaS providers. This approach makes the tool highly valuable for educational purposes.

### 7.5.2 Configurability and Experimental Design

The extensive configurability of Serv-Drishti is a significant strength for both education and architectural analysis. By allowing users to adjust parameters in real-time, the visualiser becomes a powerful tool for what-if scenario analysis and hypothesis testing. Users can vary the *cold start delay* to understand its impact on latency, especially for different programming languages. The ability to switch between Warm Priority, Round Robin, and Least Connections routing strategies allows for direct comparative analysis of their performance under different load patterns, Fig. 7.4. Furthermore, users can adjust CPU and memory consumption to explore resource contention and the trade-

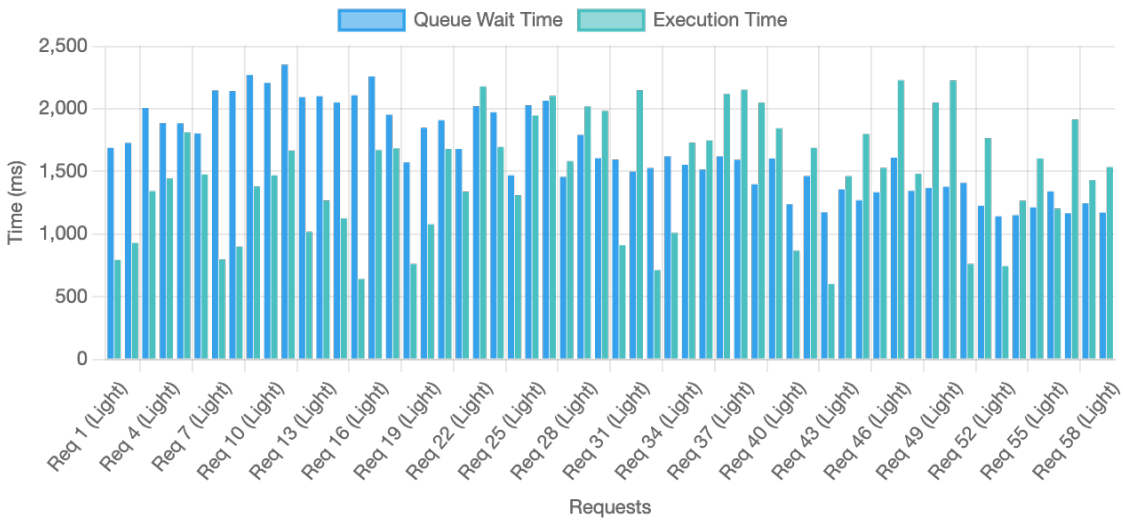



**Figure 7.4:** Impact of Request Queue and Cold Start on Function Request Performance

offs between instance size and cost efficiency. Modifying the *max concurrent requests* per function helps in understanding how this parameter affects an instance's utilisation and the system's scaling behaviour. This dynamic experimentation empowers learners and allows architects to rapidly prototype and evaluate design decisions in a risk-free environment.

### 7.5.3 Virtualisation as a Tool for Insight

The real-time, animated visualisation in Serv-Drishti is a core functional element designed to enhance comprehension and intuition. It leverages them by animating request journeys, seeing requests moving through the system, queuing, and changing function states. This provides an immediate and intuitive understanding of the workflow. The instant visual updates of queue length, function states, and node resource meters provide real-time feedback that allows users to directly correlate parameter changes with system behaviour. This feedback loop facilitates active learning and reinforces conceptual understanding. The visual queue and colour-coded instances immediately draw attention to potential bottlenecks, allowing users to quickly identify problematic configurations or load conditions.

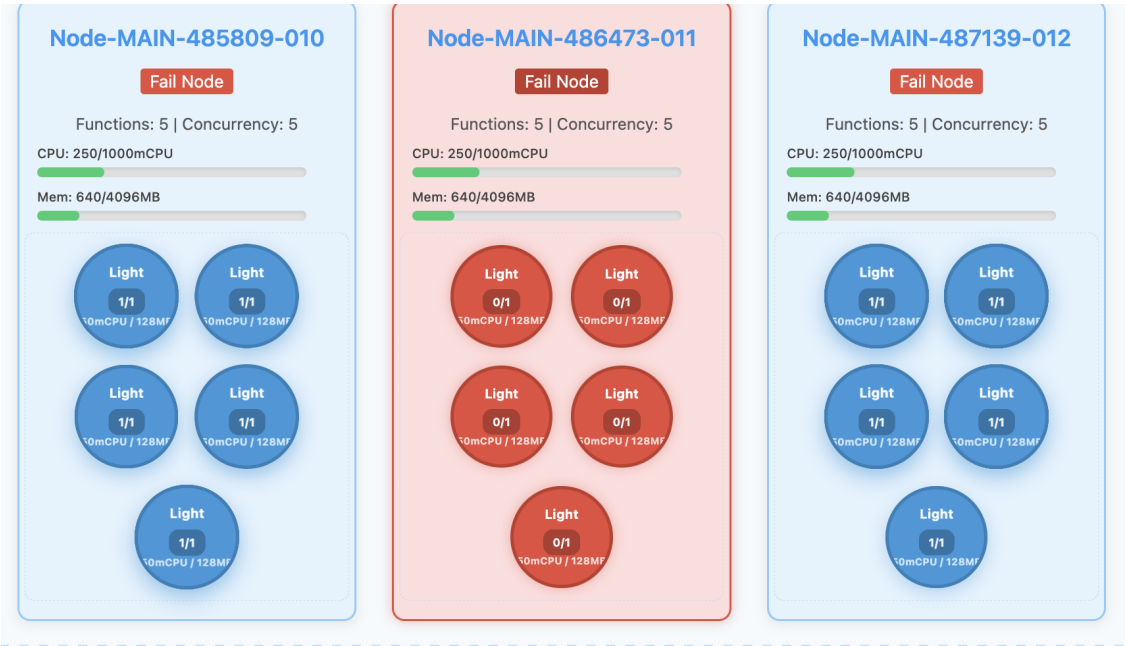



**Figure 7.5:** A Failure Simulation through Fail Node

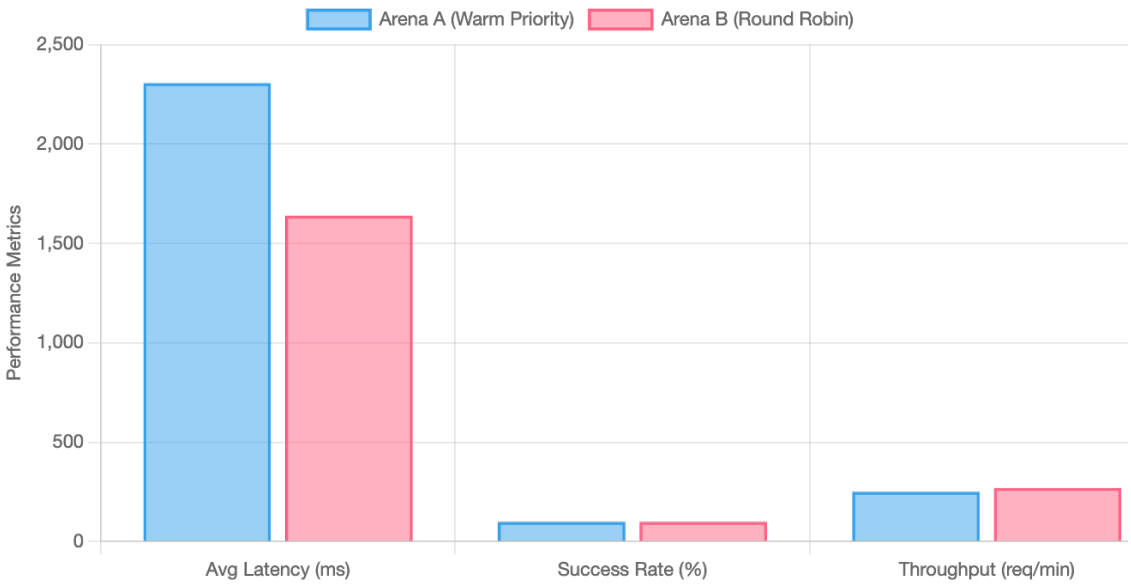



**Figure 7.6:** Battleground Performance Comparison (Routing and Placement Strategy)

## 7.6 Failure Simulation and Robustness

Understanding system resilience is paramount in distributed and cloud-native architectures. Serv-Drishti integrates a robust failure simulation module, enabling users to observe and analyse the system's response to various fault conditions, which provides invaluable insights into designing resilient serverless applications and understanding the importance of failure handling mechanisms. The simulation models several key failure scenarios. Each queued request is assigned a configurable Time-to-Live (TTL). If it is not processed within this limit, it is marked as failed and removed from the queue, which simulates real-world client timeouts and demonstrates the impact of unfulfilled requests due to system congestion or delays. Additionally, each Function Instance has a configurable *maximum execution timeout*. If a request exceeds this duration, all requests on that function are marked as failed, and the instance may be terminated, which highlights the importance of setting appropriate timeouts to prevent long-running executions from consuming excessive resources. Finally, the visualiser provides a "Fail Node" button, Fig. 7.5, allowing users to manually trigger an immediate infrastructure failure. When a node fails, all hosted functions and in-flight requests on those functions are marked as failed, but the system's robustness is demonstrated as the request dispatcher's logic automatically routes new requests to healthy nodes and provisions new function instances on them, if capacity allows. Failure events are clearly marked visually within the simulation and are accurately reflected in the performance graphs, providing a clear and compelling demonstration of failure propagation and the importance of designing for transient failures.

## 7.7 Performance Analysis and Data Export

Beyond its visual and interactive capabilities, Serv-Drishti provides quantitative data for a deeper, more rigorous performance analysis, essential for both academic research and practical architectural design. The platform integrates a robust metrics and analytics engine that provides real-time data and comprehensive reports.

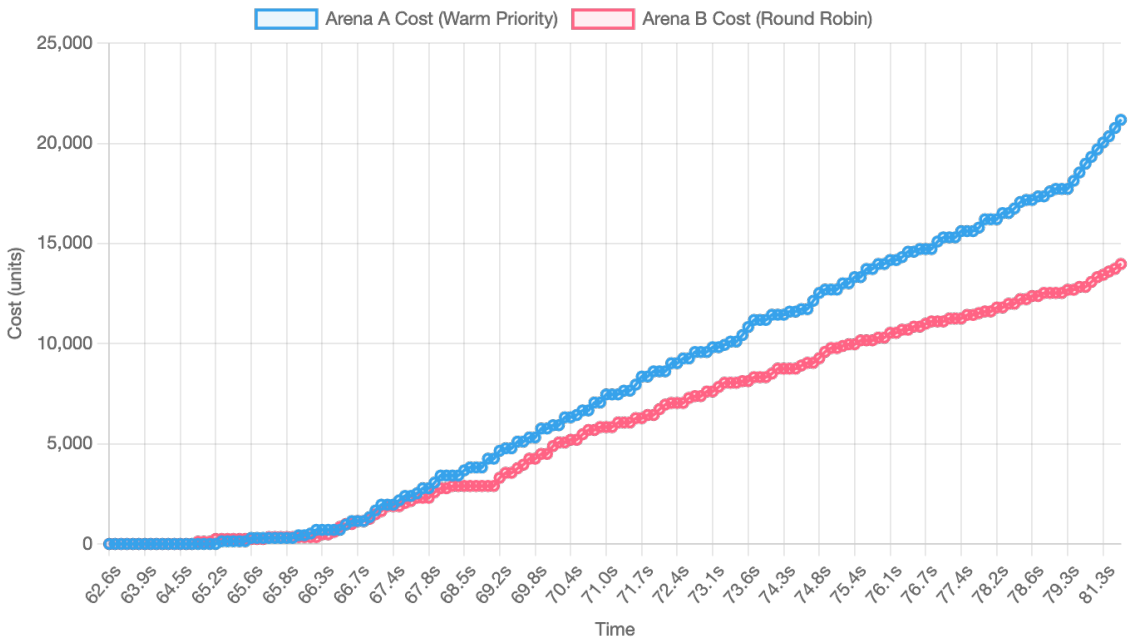



**Figure 7.7:** Cumulative Cost Analysis in Serv-Drishti

## Requests

| ID | Function Type | Node | Queue Wait | Exec Time | Total Latency | Routing | Placement | Status | Simulation |
|---|---|---|---|---|---|---|---|---|---|
| 17574706... | Light | Node-MAIN-625211-017 | 5857.0 ms | 1607.0 ms | 7464.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625868-018 | 6364.0 ms | 726.0 ms | 7090.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625868-018 | 6490.0 ms | 754.0 ms | 7244.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625868-018 | 5847.0 ms | 2064.0 ms | 7911.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625211-017 | 5413.0 ms | 683.0 ms | 6096.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625868-018 | 6172.0 ms | 1643.0 ms | 7815.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625211-017 | 5597.0 ms | 1582.0 ms | 7179.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625211-017 | - ms | 983.0 ms | 983.0 ms | current | best-fit | Success | Main |
| 17574706... | Light | Node-MAIN-625211-017 | - ms | 1083.0 ms | 1083.0 ms | current | best-fit | Success | Main |

**Figure 7.8:** Live Request Metrics Collection

### 7.7.1 Comprehensive Data Collection and Export

The platform offers several features for data observation and export, providing multiple layers of analytical depth. Live Metrics, Fig. 7.8 are continuously captured and presented in detailed tables that provide a real-time snapshot of the simulation's state, including the status and resource usage of active function instances and compute nodes, as well as a history of recently completed or failed requests.

For in-depth analysis, the platform generates a variety of interactive performance charts. A dynamic bar chart, Fig. 7.4, provides real-time insights into the average Queue Wait Time and Execution Time for requests processed within the current observation session, which can be reset for short-term experiments. A cumulative line graph continuously displays aggregate performance data over the entire simulation run, including total successful requests, total failed requests, average end-to-end latency, and average resource utilisation. A key aspect of this analysis is the cost model, which Serv-Drishti calculates for each request using a formula based on execution time and memory consumption: $(executionTimeMs/1000) * memoryMB$, Fig. 7.7. This accumulated cost is tracked and provides a direct link between performance and financial metrics.

For external analysis, the platform offers the capability to export the complete cumulative dataset in standard formats such as CSV. This granular data includes timestamps for various lifecycle events, component IDs, and performance metrics, enabling researchers to perform their own statistical analysis, create custom virtualisations, and validate hypotheses. Charts can also be exported as PNG images.

### 7.7.2 Battleground System for Comparative Analysis

The Battleground System is another critical feature that provides a dedicated environment for comparative analysis. It runs two independent "arenas" side-by-side, each of which can be configured with a different routing or placement algorithm. Synchronised auto-requests allow for direct observation of the performance trade-offs in a single, cohesive view. The battleground generates its own set of charts for direct comparison across key metrics such as average latency, success rate, and throughput, Fig. 7.6. It also features time-series charts that compare queue length, resource utilisation, active function

counts, and cumulative cost between the two arenas. This powerful feature transforms Serv-Drishti from a mere demonstration tool into a versatile platform for quantitative analysis and research, providing verifiable data to back up visual observations.

**Serv-Drishti Request Routing Logic Pseudocode**

```
switch (this.currentRoutingLogic) {
  case 'current':
    selectedFunction = availableFunctions[0];
    break;

  case 'round-robin':
    for (let i = 0; i < availableFunctions.length; i++) {
      const candidateIndex =
        (this.roundRobinIndex + i) % availableFunctions.length;
      const candidate = availableFunctions[candidateIndex];

      if (!candidate.func.isColdStarting &&
          candidate.func.concurrentRequests <
            window.globalConfig.maxConcurrentRequests) {
        selectedFunction = candidate;
        this.roundRobinIndex =
          (candidateIndex + 1) % availableFunctions.length;
        break;
      }
    }
    break;

  case 'least-connections':
    availableFunctions.sort(
      (a, b) => a.func.concurrentRequests - b.func.concurrentRequests
    );
    selectedFunction = availableFunctions[0];
    break;

  default:
```

```
31      selectedFunction = availableFunctions[0];
32  }
33
```

**Code Listing 7.1:** Serv-Drishti Request Routing Logic Pseudocode

## 7.8 Extensibility and Usage

The design of Serv-Drishti is a key consideration in its value as a research and educational platform, as its modular and open-source nature provides significant opportunities for extensibility and community contribution. It is purely implemented in JavaScript, HTML, and CSS, making the tool lightweight where it runs directly in the browser, and is highly accessible. This section outlines how users can leverage the platform's design to implement their own custom logic and details a basic guide for its practical use.

### 7.8.1 Implementation of Custom Logic

The clear separation between the simulation logic and the visualisation layer ensures that new features can be added without overloading the entire system and codebase. Serv-Drishti provides specific interfaces for core behaviours, allowing for the "plug-and-play" integration of custom algorithms.

To implement a new algorithm, a user can create a new function within the appropriate module (e.g., `core/placement-algorithms.js` or `core/simulation.js`), Listing 7.1. This function must adhere to the established interface, taking a defined set of inputs (e.g., a list of nodes, function type) and returning a specific output (e.g., the best node for placement). The simplicity of the tech stack means no external libraries or complex build processes are needed for development. For example, a researcher can implement a novel predictive scaling policy that provisions new function instances based on a predefined threshold of the request rate, which can be derived from the global request logs (`window.allRequestsLog`). This allows for a direct comparative analysis against the default reactive scaling policies. The core simulation logic will automatically

use this new function once it is implemented at the correct position, and its performance can be visualised instantly in the platform's charts and tables.

### 7.8.2 User Guide

The simplicity of the UI and the browser-based implementation make Serv-Drishti highly accessible for both learning and research.

1. Running a Simulation: Users can begin by simply opening the `index.html` file in a web browser, requiring no complex setup or dependencies. The user-friendly, collapsible UI panel allows for real-time adjustments of key simulation parameters, such as the *auto-request rate*, *cold start delay*, *routing strategy* and *placement strategy*. Users can also select a pre-configured demo scenario or manually trigger requests to observe the system's response in real-time. These scenarios are available from the `welcome` tab where appropriate interactive guides are also provided.

2. Observing Results: As the simulation runs, the real-time, animated visualisations provide immediate insights into the request journey, queuing, and function state changes. For quantitative analysis, the platforms live metrics tables and dynamic charts provide a comprehensive view of performance and resource utilisation.

3. Data Export for Analysis: As discussed earlier, Serv-Drishti offers the capability to export all simulation data. A user can download the complete cumulative dataset in formats such as CSV, which includes granular details on each request's lifecycle events, performance metrics, function instance and compute node information. This data can then be used to perform custom statistical analysis and create visualisations beyond the tool's built-in capabilities.

## 7.9 Summary

The Serv-Drishti visualiser is a powerful educational and analytical tool that provides significant transparency into the often-abstracted world of serverless computing. It

serves as a visual and interactive guide by animating the request lifecycle, simulating complex scaling and routing logic, and demonstrating realistic failure scenarios. The platform empowers users to gain a practical and intuitive understanding of serverless platform dynamics. Its highly interactive nature and extensively configurable parameters make it suitable for various learning and experimental contexts, from a university classroom where students grasp fundamental cloud concepts to a professional architecture design session evaluating different deployment strategies. The chapter fills a unique and critical gap in the ecosystem of serverless tools by focusing on interactive, visual demystification for pedagogical and architectural purposes, distinguishing itself from purely programmatic simulators or real-time monitoring solutions.

**Software Availability:** The source code of Serv-Drishti simulation engine and visualiser is accessible on https://github.com/Cloudslab/Serv-Drishti as an open-source tool under the Apache 2.0 license.

# Chapter 8
# Conclusions and Future Directions

*This chapter concludes the thesis by summarising its works and key contributions. Further, it highlights key future directions to continue advancing function resource management and configuration in serverless computing environments.*

## 8.1 Summary of Contributions

The FaaS execution model of serverless computing offers an event-driven, compute service characterised by consumption-based pricing, where no costs are incurred for idle resources. Originally designed for short, stateless, and bursty workloads, FaaS has demonstrated suitability for a diverse range of applications. This includes latency-critical applications and longer-running jobs such as video processing, AI/ML inference tasks, and big-data analytics, which often involve complex function-based workflows. The models core benefit is the elimination of infrastructure management burden that is achieved through the dynamic creation and destruction of function resources, leading to high elasticity and cost efficiency.

Despite these significant advantages, the FaaS paradigm is fundamentally constrained by the complexity of its underlying resource management. Specifically, the critical and interrelated challenges that compromise performance and resource efficiency are unpredictable cold start latency and its direct link to capacity scaling, the sub-optimal reactive autoscaling policies currently used by platforms, and the inherent static resource configuration dilemma faced by developers. The research presented in this thesis directly addresses these constraints by proposing adaptive, learning-based solutions designed to bridge the gap between static allocation and dynamic workload demand.

Chapter 1 introduced the serverless computing paradigm and its event-driven compute model - FaaS, highlighting the critical resource management challenges due to its inherent operational intricacies. Furthermore, we discuss the adverse impact of those challenges on function performance, SLO/QoS satisfaction, a developer's ability to configure them and a CSP's ability to provide the black-box resource management. Further, it aligns these challenges to the identified resource questions and summarised the thesis contributions.

Chapter 2 comprehensively discusses the background on serverless computing and FaaS execution model, reviewing the key characteristics of the model. It investigated and surveyed the existing literature for function resource configuration and management in serverless computing. Furthermore, it proposes a detailed taxonomy for categorising these works and then classify literature using the identified taxonomy.

Chapter 3 proposed the application of Q-learning model to the FaaS environments for reducing the frequent function cold starts. This initialisation overhead is due to the offered autoscaling of resources and the proposed approach mitigates them by analysing the incoming workload, its resource utilisation and failure rate to actively prepare the required number of functions in advance. It models the reduction of cold starts as an optimisation problem aimed at minimising the frequency of them. In doing so, the proposed agent attempts to learn the best policy while maintaining average CPU utilisation and reduced failure rates. A practical serverless test setup was also implemented where all the experiments including the agent training and inference were conducted.

Chapter 4 presents a deep RL-based agent, DRe-Scale, that identifies serverless environments to be partially observable due to their intrinsic multi-tenant nature where side-effects and performance interference exists in addition to the available metrics not capturing these nuances. Moreover, the agent identifies the hysteresis effect and thus models the problem of reactive function autoscaling as model-free POMDP with an objective to learn an optimal scaling policy while maintaining the satisfactory performance and throughput levels. The proposal is developed as an open-source RL agent and interacts with OpenFaaS framework for both training and inference purposes while successful experiments were conducted under this controlled setup.

Chapter 5 introduces an input-aware function configuration estimator and execu-

tor, named MemFigLess, implemented as a combination of offline profiling and online inference workflow. The MemFigLess workflow addresses the performance and cost variation of a function where it is influenced by workload related factors such as the input (size or type) received by the function for execution. The optimal configuration estimation is modelled as a multi-objective optimisation problem with an objective to select the configuration that ensures a deadline-aware execution while reducing excess resource waste and operational cost. The approach is compared against state-of-the-art proposals to demonstrate its superior results in not just the configuration estimation but also in reducing the excess resource allocation and costs along with online inference capabilities. The experiments were conducted on AWS Lambda platform with the known benchmark serverless applications.

Chapter 6 implements an end-to-end serverless framework, Saarthi, responsible for intelligent request orchestration and function execution accounting for the input-awareness, its version proliferation challenge due to various configurations being created simultaneously and service continuity while proactively optimising the scale of functions based on the historical demand knowledge. The proposed framework is integrated with OpenFaaS and attempts to CSP overhead by reusing allocated resources, reducing resource fragmentation allowing for better infrastructure utilisation and providing proactive scaling for predictive workloads. Saarthi leverages hybrid optimisation approach to tackle version proliferation and models predictive scaling as an ILP-based optimisation problem with multi-objectives to account for operational cost, throughput and cold start trade-off analysis. Additionally, it exploits a redundancy-based fault-tolerance mechanism to provide continuous services. The experiments are conducted under the controlled setting on OpenFaaS and the performance is compared to the baseline community edition of OpenFaaS, a representative of FaaS platforms.

Chapter 7 presents an open-source, interactive tool to simulate and visualise the various resource management aspects of a FaaS platform. This includes not only the function model, but also the underlying infrastructure and provider-managed function scheduling decisions like request routing and function placement. Additionally, it enables a a comprehensive comparison of 'what-if' scenarios where different routing and placement policies can be tested. The tool covers a significant in the research where other

simulation tools involve a steep learning curve and is designed to be extensible where new policies can be added and tested. It is developed using Javascript and allows the user to extract all the service metrics produced during the simulation.

The chapters discussed above presented multiple approaches, techniques and practical frameworks toward optimising function resource configuration and management in FaaS execution model, representing a timely contribution to the state-of-the-art. The outcomes of these works present a strong evidence-based methodology for addressing reactive autoscaling, its cold start overheads and replacing static resource allocation with adaptive, learning-based solutions for the benefit of all the stakeholders involved.

## 8.2 Future Research Directions

Based on the research works presented in this thesis, we propose potential future research directions for the various resource management aspects of FaaS execution model and serverless computing.

### 8.2.1 Dynamic Resource Allocation Factors

Existing FaaS platforms require developers to either select or configure the compute resources for the entire scope and lifecycle of a function. The actual resource needs of a function can fluctuate significantly based on runtime factors such as the input parameters it receives and the observed workload characteristics [33][18]. This dynamism indicates that adaptive resource configuration methods, which could adjust the allocated resources of a function based on the runtime requirements, are promising for enhancing cost and performance efficiency in FaaS. In particular, the effects and feasibility of granular vertical scaling in FaaS deserve thorough empirical investigation. In this sense, ML techniques, such as RL and predictive modelling, could play a crucial role in developing intelligent adaptive resource configuration mechanisms that can learn from past behaviour and anticipate future resource needs based on input parameters and workload patterns. Such techniques could lead to more efficient resource utilisation and improved application performance by ensuring that functions always have the right

amount of resources when they need them. Supplemental to this, the selected language runtime of the function significantly influences its resource requirements, runtime performance, and cost [196]. Additionally, an affinity to a specific resource [197][124] such as CPU architecture or memory of the underlying infrastructure could also affect the configuration decisions in the serverless offerings. To this end, future research should focus on comprehensively analysing the performance characteristics of serverless functions implemented in different programming languages across various serverless platforms. In addition to this, research should explore the impact of resource affinity on serverless function performance. In particular, functions that can benefit from specific CPU features or architectures could be preferentially scheduled on nodes that provide those capabilities. Hence, investigating techniques that allow developers to express resource affinity requirements and enabling serverless platforms to honour these preferences could lead to significant performance improvements.

### 8.2.2 Decoupled Resource Configuration

Another research direction is decoupling of underlying infrastructure where the resources are not bound to memory allocation, for instance. A few of the existing works [20][34][198] have discussed this decoupled allocation and attempted to optimise the single resource configuration for a performance improvement. As the serverless infrastructure grows in heterogeneity, comprising different types of servers, storage, and network devices, management and configuration of them emerge as a unique challenge and opportunity for dynamic resource configuration in serverless computing. As a result, the resource search space explodes with numerous possibilities, presenting a potential to explore the relationship among different resources and their impact on function performance and cost. Additionally, the support for specialised hardware such as Graphics Processing Unit (GPU) and Field-Programmable Gate Array (FPGA) remains an open question that could further have a significant impact on resource configuration decisions for specialised workloads. This opens up a possibility for heterogeneity-aware function management that can intelligently configure functions based on available hardware [199].

### 8.2.3 Resource Configuration for Workflows

Serverless applications are often composed of multiple interconnected functions that form workflows. The dependencies between these functions and the patterns of data flow within the workflow can significantly impact the overall performance and resource efficiency of the application. A possible future study should examine how workflow dependencies and data flow patterns between interconnected functions can be leveraged to optimise the dynamic resource configuration of the overall application. However, the studies on workflow optimisation in serverless [94][86] and data flow aware resource management [108] for serverless provide a strong foundation. This could involve developing scheduling algorithms that consider the execution order and data dependencies between functions, optimising resource allocation for functions based on the volume and frequency of data they exchange, and co-locating dependent functions on the same compute nodes to minimise network latency. Additionally, these workflows could further leverage the serverless resources of different providers, where the resource configuration decisions not only impact the overall workflow performance, but also significantly influence the runtime cost. Moreover, research should identify potential cross-platform challenges in dynamic resource configuration, such as differences in resource models, scaling policies, and monitoring capabilities of different CSPs. Thus, exploring the feasibility of vendor-agnostic approaches to resource configuration and management, perhaps through the use of standardised APIs or abstraction layers, is another important direction for future work. As a result, addressing the proposed research opportunities will lead to more efficient, performant, and cost-effective serverless applications, unlocking the full potential of this promising cloud computing paradigm.

### 8.2.4 Configuration-aware Scheduling

The configuration of function resources such as memory and CPU allocation has a profound impact on the scheduling decisions made by the underlying platform [20][118]. These decisions encompass crucial aspects like function placement on available infrastructure, the potential for co-location with other functions, and the resulting interference that may arise in multi-tenant environments. The current serverless platforms often em-

ploy relatively simple, classic scheduling algorithms that may not fully account for the unique characteristics of serverless workloads, such as their burstiness, concurrency, and very short lifecycle. Future research should delve deeper into how different resource configuration decisions influence function placement strategies. Furthermore, the impact of resource configuration on the co-location of functions needs careful examination. On one hand, placing functions with similar resource demands or communication patterns together might enhance performance through improved locality; on the other hand, co-locating resource-intensive functions could lead to adverse interference. Existing research [82] discusses that resource interference and co-location may lead to resource contention like CPU, memory, and network bandwidth that can significantly downgrade performance and violate SLOs. Therefore, function configuration-aware scheduling algorithms that are also adaptable to predict and mitigate potential co-location issues should be explored in future studies.

## 8.3 Final Remarks

The abstraction of FaaS under serverless computing paradigm has emerged as the defining execution model for modern cloud-native infrastructures, driven by the need for maximum resource abstraction and dynamic elasticity. In striving to deliver instant scalability and cost efficiency, these environments exhibit highly inter-related function execution and platform resource layers, which severely limits the CSP's efficiency in ensuring predictable performance and optimal resource utilisation. Adversely, this limited visibility and the static resource allocation placed upon the developer constrain the opportunity to achieve seamless QoS and maximum economic efficiency without manual intervention.

With the increasing adoption of FaaS for complex, latency-sensitive workloads (such as AI/ML inference), improving resource predictability and operational efficiency is critical. In this thesis, we explored the tight coupling of static configuration decisions and dynamic runtime mechanisms in the FaaS execution model to improve its resource management efficiency through adaptive, learning-based approaches. The algorithms, predictive models, and integrated frameworks presented in this thesis achieve better re-

source utilisation and cost efficiency of function configuration, scheduling and autoscaling related aspects while maintaining adequate user QoS, application latency performance and ensuring SLO adherence. Furthermore, the research outcomes of this thesis identify significant opportunities to continue advancing autonomous resource orchestration and predictable QoS in the rapidly evolving serverless ecosystem.

# Bibliography

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, Serverless Computing: Current Trends and Open Problems. Singapore: Springer Singapore, 2017, pp. 1–20.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," 2019.

[3] Precedence Research, "Serverless computing market size to hit USD 92.22 billion by 2034," Precedence Research, Market Research Report, July 2025, report ID: Not specified. [Online]. Available: https://www.precedenceresearch.com/serverless-computing-market

[4] Grand View Research, "Serverless computing market size, share & trends analysis report by service model (function-as-a-service, backend-as-a-service), by deployment, by enterprise size, end-use, by region, and segment forecasts, 2025–2030," Grand View Research, Industry Report, October 2024, report ID: GVR-4-68040-502-6, 120 pages. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/serverless-computing-market-report

[5] IBM. (2025) What is serverless computing? Accessed: 2025-10-30. [Online]. Available: https://www.ibm.com/think/topics/serverless

[6] A. W. Services. (2024) Aws lambda - run code without thinking about servers or clusters. [Online]. Available: https://aws.amazon.com/lambda/

[7] Microsoft. (2024) Azure functions documentation. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/

[8] Google. (2024) Cloud run functions. [Online]. Available: https://cloud.google.com/blog/products/serverless/google-cloud-functions-is-now-cloud-run-functions

[9] OpenFaaS. (2016) Openfaas-serverless function, made simple. [Online]. Available: https://www.openfaas.com/

[10] K. Authors. (2024) Knative is an open-source enterprise-level solution to build serverless and event driven applications. [Online]. Available: https://knative.dev/docs/

[11] OpenWhisk. (2016) Openwhisk - open source serverless cloud platform. [Online]. Available: https://openwhisk.apache.org/

[12] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in Proceedings of the International Conference on Omni-layer Intelligent Systems (COINS). IEEE, 2020, pp. 1–7.

[13] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in Proceedings of the USENIX Annual Technical Conference, 2020, pp. 205–218.

[14] A. Stuyvenberg. (2023) Understanding aws lambda proactive initialization. Accessed: November 19, 2024. [Online]. Available: https://aaronstuyvenberg.com/posts/understanding-proactive-initialization

[15] A. W. Services. (2025) Lambda managed instances. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-managed-instances.html

[16] Y. Cui, "aws lambda compare coldstart time with different languages, memory and code sizes," 2017. [Online]. Available: https://theburningmonk.com/2017/06/aws-lambda-compare-coldstart-time-with-different-languages-memory-and-code-sizes/

[17] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and C. Das, "Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms," in Proceedings of the 13th Symposium on Cloud Computing, New York, NY, USA, 2022, p. 257272.

[18] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "Ofc: An opportunistic caching system for faas platforms," in Proceedings of the 16th European Conference on Computer Systems, New York, NY, USA, 2021, p. 228244.

[19] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 10631075.

[20] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in Proceedings of the Eighteenth European Conference on Computer Systems, New York, NY, USA, 2023, p. 381397.

[21] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, NY, USA, 2021.

[22] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," Commun. ACM, vol. 64, no. 5, p. 7684, apr 2021.

[23] S. Kounev, N. Herbst, C. L. Abad, A. Iosup, I. Foster, P. Shenoy, O. Rana, and A. A. Chien, "Serverless computing: What it is, and what it is not?" Commun. ACM, vol. 66, no. 9, p. 8092, aug 2023.

[24] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al., "Cloud programming simplified: A berkeley view on serverless computing," arXiv preprint arXiv:1902.03383, 2019.

[25] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," arXiv preprint arXiv:1903.12221, 2019.

[26] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 134–143.

[27] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," ACM Computing Surveys (CSUR), 2019.

[28] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.

[29] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, 2018, pp. 181–188.

[30] K. Solaiman and M. A. Adnan, "Wlec: A not so cold architecture to mitigate cold start problem in serverless computing," in Proceedings of the IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2020, pp. 144–153.

[31] S. N. A. Jawaddi and A. Ismail, "Autoscaling in serverless computing: Taxonomy and openchallenges," 2023, pre-print on webpage https://doi.org/10.21203/rs.3.rs-2897886/v1.

[32] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, and S. Kounev, "Why is it not solved yet? challenges for production-ready autoscaling," in Proceedings of the ACM/SPEC on International Conference on Performance Engineering, 2022, pp. 105–115.

[33] A. Moghimi, J. Hattori, A. Li, M. Ben Chikha, and M. Shahrad, "Parrotfish: Parametric regression for optimizing serverless functions," in Proceedings of the 2023 ACM Symposium on Cloud Computing, New York, NY, USA, 2023, p. 177192.

[34] P. Sinha, K. Kaffes, and N. J. Yadwadkar, "Online learning for right-sizing serverless functions," in Proceedings of the Architecture and System Support for Transformer Models (ASSYST @ISCA 2023), 2023.

[35] D. Tomaras, M. Tsenos, and V. Kalogeraki, "Prediction-driven resource provisioning for serverless container runtimes," in Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), 2023, pp. 1–6.

[36] A. W. Services. (2023) Welcome to serverless land. Last accessed on 01/02/2023. [Online]. Available: https://serverlessland.com/

[37] ——. (2023) Serverlessvideo: Connect with users around the world! Last accessed on 27/11/2023. [Online]. Available: https://video.serverlessland.com/

[38] ——. (2020) Serverless case study - netflix. Last accessed on 01/02/2023. [Online]. Available: https://dashbird.io/blog/serverless-case-study-netflix/

[39] CapitalOne. (2023) Capital one saves developer time and reduces costs by going serverless on aws. Last accessed on 27/11/2023. [Online]. Available: https://aws.amazon.com/solutions/case-studies/capital-one-lambda-ecs-case-study/

[40] E. Johnson, "Deploying ml models with serverless templates," https://aws.amazon.com/blogs/compute/deploying-machine-learning-models-with-serverless-templates/, 2021.

[41] A. Sojasingarayar, "Build and deploy llm application in aws," https://medium.com/@abonia/build-and-deploy-llm-application-in-aws-cca46c662749, 2024.

[42] A. W. Services. (2024) Lambda quotas. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[43] Microsoft. (2024) Azure functions consumption plan hosting. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/consumption-plan

[44] A. Bhattacharya and T. Wen, "Understanding and remediating cold starts: An aws lambda perspective,"

2025. [Online]. Available: https://aws.amazon.com/blogs/compute/understanding-and-remediating-cold-starts-an-aws-lambda-perspective/

[45] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "Slam: Slo-aware memory optimization for serverless applications," in Proceedings of the 15th International Conference on Cloud Computing, 2022, pp. 30–39.

[46] Y. Cui, "How does language, memory and package size affect cold starts of aws lambda?" 2023. [Online]. Available: https://www.pluralsight.com/resources/blog/cloud/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda

[47] A. W. Services. (2025) Amazon elastic compute cloud (amazon ec2). Accessed: 2025-12-07. [Online]. Available: https://aws.amazon.com/ec2/

[48] ——. (2025) Aws elastic beanstalk. Accessed: 2025-08-07. [Online]. Available: https://aws.amazon.com/elasticbeanstalk/

[49] Salesforce. (2025) Salesforce: The customer company. Accessed: August 20, 2025. [Online]. Available: https://www.salesforce.com/au/

[50] J. Wood and C. Munns. (2023, Dec.) Aws re:invent 2023 - best practices for serverless developers. [Online]. Available: https://www.youtube.com/watch?v=sdCA0Y7QDrM

[51] Amazon Web Services. (2025) Amazon s3 - cloud object storage. Accessed: August 20, 2025. [Online]. Available: https://aws.amazon.com/s3/

[52] A. W. Services. (2025) Amazon dynamodb. [Online]. Available: https://aws.amazon.com/dynamodb/

[53] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," IEEE Transactions on Services Computing, vol. 16, no. 2, pp. 1522–1539, 2023.

[54] Apache Software Foundation. (2025) Apache kafka. Accessed: August 20, 2025. [Online]. Available: https://kafka.apache.org/

[55] A. W. Services. (2024) Aws lambda pricing. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[56] ——. (2024) Aws step functions. [Online]. Available: https://aws.amazon.com/step-functions/

[57] Microsoft. (2025) Azure functions consumption plan. Accessed: 2025-03-15. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/consumption-plan

[58] Microsoft. (2024) Azure durable functions documentation. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/

[59] Google. (2024) Cpu allocation (services). [Online]. Available: https://cloud.google.com/run/docs/configuring/cpu-allocation

[60] ——. (2024) Workflows. [Online]. Available: https://cloud.google.com/workflows

[61] Kubernetes, "Understand kubernetes," https://kubernetes.io/docs/home/.

[62] Prometheus: Monitoring System and Time Series Database, Prometheus Authors, 2025, accessed: August 2025. [Online]. Available: https://prometheus.io/

[63] Grafana: Open Source Analytics and Monitoring, Grafana Labs, 2025, accessed: August 2025. [Online]. Available: https://grafana.com/

[64] Fluent Bit: Open Source and Multi-platform Log Processor and Forwarder, Fluent Bit Authors, 2025, accessed: August 2025. [Online]. Available: https://fluentbit.io/

[65] Elasticsearch: Distributed RESTful Search and Analytics Engine, Elastic NV, 2025, accessed: August 2025. [Online]. Available: https://www.elastic.co/elasticsearch/

[66] PostgreSQL: The World's Most Advanced Open Source Database, PostgreSQL Global Development Group, 2025, accessed: August 2025. [Online]. Available: https://www.postgresql.org/

[67] (2025) Amazon elastic kubernetes service (eks). Amazon Web Services. Accessed: August 2025. [Online]. Available: https://aws.amazon.com/eks/

[68] (2025) Azure kubernetes service (aks). Microsoft Azure. Accessed: August 2025. [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/

[69] (2025) Google kubernetes engine (gke). Google Cloud Platform. Accessed: August 2025. [Online]. Available: https://cloud.google.com/kubernetes-engine

[70] OpenWhisk. (2016) Openwhisk system details. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/reference.md

[71] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: lightweight virtualization for serverless applications," in Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation, USA, 2020, p. 419434.

[72] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, 2021, pp. 797–803.

[73] L. Schuler, S. Jamil, and N. K"uhl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in Proceedings of the IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, 2021, pp. 804–811.

[74] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms," Simulation Modelling Practice and Theory, vol. 116, p. 102461, 2022.

[75] H.-D. Phung and Y. Kim, "A prediction based autoscaling in serverless computing," in Proceedings of the 13th International Conference on Information and Communication Technology Convergence (ICTC). IEEE, 2022, pp. 763–766.

[76] Z. Zhang, T. Wang, A. Li, and W. Zhang, "Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning," in Proceedings of the 46th IEEE Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2022, pp. 866–871.

[77] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," IEEE Transactions on Network and Service Management, vol. 19, no. 4, pp. 3995–4009, 2022.

[78] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications," in Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). IEEE, 2022, pp. 674–679.

[79] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, 2022, pp. 180–189.

[80] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," IEEE Internet of Things Journal, 2022.

[81] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in Proceedings of the 25th IEEE International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2019, pp. 9–16.

[82] Y. Tang and J. Yang, "Lambdata: Optimizing serverless computing by making data intents explicit," in Proceedings of the 13th IEEE International Conference on Cloud Computing (CLOUD), 2020, pp. 294–303.

[83] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless:

Predicting the optimal size of serverless functions," in Proceedings of the 22nd International Middleware Conference, New York, NY, USA, 2021, p. 248259.

[84] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in Proceedings of the ACM/SPEC International Conference on Performance Engineering, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265276.

[85] Z. Wang, S. Zhang, J. Cheng, Z. Wu, Z. Cao, and Y. Cui, "Edge-assisted adaptive configuration for serverless-based video analytics," in Proceedings of the 43rd International Conference on Distributed Computing Systems (ICDCS), 2023, pp. 248–258.

[86] M. Zhang, Y. Zhu, J. Liu, F. Wang, and F. Wang, "Charmseeker: Automated pipeline configuration for serverless video processing," IEEE/ACM Transactions on Networking, vol. 30, no. 6, pp. 2730–2743, 2022.

[87] M. Ghorbian and M. Ghobaei-Arani, "Function offloading approaches in serverless computing: A survey," Computers and Electrical Engineering, vol. 120, p. 109832, 2024.

[88] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 615–632, 2020.

[89] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," ACM Comput. Surv., vol. 54, no. 11s, sep 2022.

[90] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," IEEE Transactions on Software Engineering, vol. 48, no. 10, pp. 4152–4166, 2022.

[91] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "Wisefuse: Workload characterization and dag transformation for

serverless workflows," Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 6, no. 2, pp. 1–28, 2022.

[92] Datadog. (2020) The state of serverless, 2020. Last accessed on 01/02/2023. [Online]. Available: https://www.datadoghq.com/state-of-serverless-2020/

[93] A. Raza, N. Akhtar, V. Isahagian, I. Matta, and L. Huang, "Configuration and placement of serverless applications using statistical learning," IEEE Transactions on Network and Service Management, vol. 20, no. 2, pp. 1065–1077, 2023.

[94] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in 2018 IEEE/ACM Symposium on Edge Computing (SEC), 2018, pp. 300–312.

[95] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in Proceedings of the ACM Symposium on Cloud Computing, New York, NY, USA, 2021, p. 117.

[96] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Astrea: Auto-serverless analytics towards cost-efficiency and qos-awareness," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 12, pp. 3833–3849, 2022.

[97] W. Wang, Q. Wu, Z. Zhang, J. Zeng, X. Zhang, and M. Zhou, "A probabilistic modeling and evolutionary optimization approach for serverless workflow configuration," Software: Practice and Experience, 2023.

[98] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, 2022, pp. 303–320.

[99] H. Wu, J. Deng, H. Fan, S. Ibrahim, S. Wu, and H. Jin, "Qos-aware and cost-efficient dynamic resource allocation for serverless ml workflows," in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), 2023, pp. 886–896.

[100] A. P. Jegannathan, R. Saha, and S. K. Addya, "A time series forecasting approach to minimize cold start time in cloud-serverless platform," in Proceedings of the IEEE International black sea conference on communications and networking (BlackSeaCom). IEEE, 2022, pp. 325–330.

[101] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in Proceedings of the 21st International Middleware Conference, 2020, pp. 1–13.

[102] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li et al., "Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing," in 2022 USENIX annual technical conference (USENIX ATC 22), 2022, pp. 69–84.

[103] K. Mahajan, S. Mahajan, V. Misra, and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in Proceedings of the 15th International Conference on emerging Networking Experiments and Technologies, 2019, pp. 37–39.

[104] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," IEEE Transactions on Cloud Computing, vol. 10, no. 4, pp. 2834–2847, 2022.

[105] T. Zubko, A. Jindal, M. Chadha, and M. Gerndt, "Maff: Self-adaptive memory optimization for serverless functions," in Service-Oriented and Cloud Computing, F. Montesi, G. A. Papadopoulos, and W. Zimmermann, Eds., 2022, pp. 137–154.

[106] M. Li, J. Zhang, J. Lin, Z. Chen, and X. Zheng, "Fireface: Leveraging internal function features for configuration of functions on serverless edge platforms," Sensors, vol. 23, no. 18, 2023.

[107] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in Proceedings of the 22nd International Middleware Conference, New York, NY, USA, 2021, p. 6478.

[108] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in Proceedings of the IEEE Conference on Computer Communications, 2022, pp. 1868–1877.

[109] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt, "Estimating the capacities of function-as-a-service functions," in Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2022.

[110] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD). IEEE, 2019, pp. 502–504.

[111] J. Manner and G. Wirtz, "Resource scaling strategies for open-source faas platforms compared to commercial cloud offerings," in Proceedings of the 15th IEEE International Conference on Cloud Computing (CLOUD). IEEE, 2022, pp. 40–48.

[112] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in Proceedings of the ACM Symposium on Cloud Computing, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 153167.

[113] L. Zhu, G. Giotis, V. Tountopoulos, and G. Casale, "Rdof: Deployment optimization for function as a service," in Proceedings of the 14th International Conference on Cloud Computing (CLOUD), 2021, pp. 508–514.

[114] Z. Li, H. Yu, and G. Fan, "Time-cost efficient memory configuration for serverless workflow applications," Concurrency and Computation: Practice and Experience, vol. 34, no. 27, p. e7308, 2022.

[115] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in Proceedings of the ACM symposium on cloud computing, 2019, pp. 158–164.

[116] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated fine-grained cpu cap control in serverless computing platform," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 10, pp. 2289–2301, 2020.

[117] Anaconda. (2024) Dask.distributed: Lightweight library for distributed computing in python. [Online]. Available: https://distributed.dask.org/en/stable/

[118] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems, 2020, pp. 1–10.

[119] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in Proceedings of the 5th International Workshop on Serverless Computing, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1924.

[120] Q. Liu, Y. Yang, D. Du, Y. Xia, P. Zhang, J. Feng, J. R. Larus, and H. Chen, "Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with jiagu," in Proceedings of the USENIX Annual Technical Conference, 2024, pp. 1–17.

[121] M. Yu, T. Cao, W. Wang, and R. Chen, "Pheromone: Restructuring serverless computing with data-centric function orchestration," IEEE/ACM Transactions on Networking, pp. 1–15, 2024.

[122] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 114.

[123] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in Proceedings of the IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, 2020, pp. 129–138.

[124] R. Cordingly, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with cpu time accounting," in 2022 IEEE International Conference on Cloud Engineering (IC2E), 2022, pp. 104–115.

[125] Y. Sfakianakis, M. Marazakis, C. Kozanitis, and A. Bilas, "Latest: Vertical elasticity for millisecond serverless execution," in 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2022, pp. 879–885.

[126] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," Proc. VLDB Endow., vol. 13, no. 11, pp. 2438–2452, 2020.

[127] D. Ustiugov, V. T. Ravi, R. Bruno, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in Proceedings of the ACM Symposium on Cloud Computing (SoCC). Association for Computing Machinery, 2021.

[128] Kubeless, "Kubeless-kubernetes native serverless," https://kubeless.io/docs/, 2019.

[129] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in Proceedings of the USENIX Annual Technical Conference (USENIX ATC 20), 2020, pp. 205–218.

[130] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," arXiv preprint arXiv:1812.03651, 2018.

[131] M. Shilkov, "Comparison of cold starts in serverless functions across aws, azure, and gcp," https://mikhail.io/serverless/coldstarts/big3/, January, 2021.

[132] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 442–450.

[133] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in Proceedings of the USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 133–146.

[134] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017, pp. 405–410.

[135] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2017, pp. 162–169.

[136] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in Proceedings of the IEEE Conference on Network Softwarization (NetSoft). IEEE, 2019, pp. 351–359.

[137] S. K. Mohanty, G. Premsankar, M. Di Francesco et al., "An evaluation of open source serverless computing frameworks." in Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 115–120, 2018.

[138] R. S. Sutton, "Reinforcement learning: Past, present and future," in Proceedings of the Asia-Pacific Conference on Simulated Evolution and Learning. Springer, 1998, pp. 195–197.

[139] M. Ashraf, "Reinforcement learning demystified: Markov decision processes," Towards Data Science, 2018.

[140] A. Galstyan, K. Czajkowski, and K. Lerman, "Resource allocation in the grid using reinforcement learning," in Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004., vol. 1. IEEE Computer Society, 2004, pp. 1314–1315.

[141] A. Jmeter, "Apache jmeter-getting started," https://jmeter.apache.org/usermanual/index.html.

[142] S. Zychlinski, "Qrash course: Reinforcement learning 101 deep q networks in 10 minutes," https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677, 10 January, 2019.

[143] Google. (2024) Cloud functions. [Online]. Available: https://cloud.google.com/functions

[144] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, "Reinforcement learning-based application autoscaling in the cloud: A survey," Engineering Applications of Artificial Intelligence, vol. 102, p. 104288, 2021.

[145] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–33, 2018.

[146] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," Sensors, vol. 20, no. 16, p. 4621, 2020.

[147] T. Ni, B. Eysenbach, and R. Salakhutdinov, "Recurrent model-free rl can be a strong baseline for many pomdps," arXiv preprint arXiv:2110.05038, 2021.

[148] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in Proceedings of the aaai fall symposium series, 2015.

[149] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," Journal of Machine Learning Research, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[150] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulo, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierr, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023.

[151] AWS, "Lambda function scaling - aws lambda," https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html, 2023, last accessed on 01/02/2023.

[152] G. C. Functions, "Configure minimum instances - google cloud functions," https://cloud.google.com/functions/docs/configuring/min-instances, 2023, last accessed on 01/02/2023.

[153] C. Gold, "Fx trading via recurrent reinforcement learning," in Proceedings of the IEEE International Conference on Computational Intelligence for Financial Engineering, 2003. Proceedings. IEEE, 2003, pp. 363–370.

[154] P. Hu, L. Pan, Y. Chen, Z. Fang, and L. Huang, "Effective multi-user delay-constrained scheduling with deep recurrent reinforcement learning," in Proceedings of the 23rd International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing, 2022, pp. 1–10.

[155] MicroK8s, "Microk8s-the lightweight kubernetes," https://microk8s.io/, 2023.

[156] PyTorch, "Recurrent dqn: Training recurrent policies," https://pytorch.org/tutorials/intermediate/dqn_with_rnn_tutorial.html, 2023, last accessed on 30/10/2023.

[157] A. Juliani, "Deeprl-agents: A set of deep reinforcement learning agents implemented in tensorflow." https://github.com/awjuliani/DeepRL-Agents/tree/master, 2023, last accessed on 30/10/2023.

[158] X. Li, L. Li, J. Gao, X. He, J. Chen, L. Deng, and J. He, "Recurrent reinforcement learning: a hybrid approach," arXiv preprint arXiv:1509.03044, 2015.

[159] B. Bakker, "Reinforcement learning with long short-term memory," Advances in neural information processing systems, vol. 14, 2001.

[160] Q. Cong and W. Lang, "Double deep recurrent reinforcement learning for centralized dynamic multichannel access," Wireless Communications and Mobile Computing, vol. 2021, pp. 1–10, 2021.

[161] R. Kozlica, S. Wegenkittl, and S. Hiränder, "Deep q-learning versus proximal policy optimization: Performance comparison in a material sorting task," in Proceednigs of the 32nd International Symposium on Industrial Electronics (ISIE). IEEE, 2023, pp. 1–6.

[162] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," CoRR, vol. abs/1707.06347, 2017.

[163] ——, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017.

[164] J. Dogan, "Hey: Http load generator, apachebench (ab) replacement," https://github.com/rakyll/hey, 2023, last accessed on 01/02/2023.

[165] A. W. Services, "Aws compute optimizer," https://aws.amazon.com/compute-optimizer/, 2023.

[166] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in Proceedings of 12th International Conference on Cloud Computing, 2019, pp. 502–504.

[167] AWS, "Amazon cloudwatch," https://docs.aws.amazon.com/cloudwatch/, 2024.

[168] A. Casalboni, "Profiling functions with aws lambda power tuning," 2023. [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning

[169] L. Breiman, "Random forests," Machine learning, vol. 45, pp. 5–32, 2001.

[170] R. B. Uriarte, S. Tsaftaris, and F. Tiezzi, "Service clustering for autonomic clouds using random forest," in Proceedings of 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 515–524.

[171] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. A. M. Elfadel, "Proactive random-forest autoscaler for microservice resource allocation," IEEE Access, vol. 11, pp. 2570–2585, 2023.

[172] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof, "On-line random forests," in 2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops, 2009, pp. 1393–1400.

[173] M. Golec, D. Chowdhury, S. Jaglan, S. S. Gill, and S. Uhlig, "Aiblock: Blockchain based lightweight framework for serverless computing using ai," in Proceedings of 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2022, pp. 886–892.

[174] P. Ngatchou, A. Zarei, and A. El-Sharkawi, "Pareto multi objective optimization," in Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems, 2005, pp. 84–91.

[175] AWS, "Serverless on aws," https://aws.amazon.com/serverless/, 2024.

[176] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[177] T. Pusztai and S. Nastic, "Chunkfunc: Dynamic slo-aware configuration of serverless functions," IEEE Transactions on Parallel and Distributed Systems, vol. 36, no. 6, pp. 1237–1252, 2025.

[178] A. W. Services. (2025) Aws lambda service level agreement. Last accessed on 30/08/2025. [Online]. Available: https://aws.amazon.com/lambda/sla/

[179] A. Ellis. (2022) Rethinking auto-scaling for openfaas. Last accessed on 01/08/2025. [Online]. Available: https://www.openfaas.com/blog/autoscaling-functions/

[180] ——. (2025) Introducing queue based scaling for functions. Last accessed on 01/08/2025. [Online]. Available: https://www.openfaas.com/blog/queue-based-scaling/

[181] O. Author(s). (2024) Comparison of openfaas editions. Last accessed on 01/08/2025. [Online]. Available: https://docs.openfaas.com/openfaas-pro/comparison/

[182] M. Pandey and Y. W. Kwon, "Optimizing memory allocation in a serverless architecture through function scheduling," in IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops, 2023, pp. 275–277.

[183] O. Sedefoğlu and H. Sözer, "Cost minimization for deploying serverless functions," in Proceedings of the 36th Annual ACM Symposium on Applied Computing, New York, NY, USA, 2021, p. 8385.

[184] J. S. Roy and S. A. M. et al., "PuLP: A Python Linear Programming API," https://github.com/coin-or/pulp, 2025, version 3.2.2, accessed September 20, 2025.

[185] S. Agarwal, M. A. Rodriguez, and R. Buyya, "Input-based ensemble-learning method for dynamic memory configuration of serverless computing functions," in Proceedings of the 17th International Conference on Utility and Cloud Computing, USA, 2024, p. 346355.

[186] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in Proceedings of the 21st International Middleware Conference, New York, NY, USA, 2020, p. 280295.

[187] O. Alqaryoutia and N. Siyamb, "Serverless computing and scheduling tasks on cloud: A," American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS), vol. 40, no. 1, pp. 235–247, 2018.

[188] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," ACM Computing Surveys, vol. 57, no. 3, pp. 1–36, 2024.

[189] A. Mampage and R. Buyya, "Cloudsimsc: A toolkit for modeling and simulation of serverless computing environments," in Proceedings of the International Conference on High Performance Computing and Communications, 2023, pp. 550–557.

[190] N. Mahmoudi and H. Khazaei, "Performance modeling of metric-based serverless computing platforms," IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1899–1910, 2023.

[191] R. Andreoli, J. Zhao, T. Cucinotta, and R. Buyya, "Cloudsim 7g: An integrated toolkit for modeling and simulation of future generation cloud computing environments," Software: Practice and Experience, vol. 55, no. 6, pp. 1041–1058, 2025.

[192] Y. Semenov and O. Sukhoroslov, "Dslab faas: Fast and accurate simulation of faas clouds," Physics of Particles and Nuclei, vol. 55, no. 3, pp. 485–488, 2024.

[193] P. Raith, T. Rausch, A. Furutanpey, and S. Dustdar, "faas-sim: A trace-driven simulation framework for serverless edge computing platforms," Software: Practice and experience, vol. 53, no. 12, pp. 2327–2361, 2023.

[194] F. Mastenbroek, G. Andreadis, S. Jounaid, W. Lai, J. Burley, J. Bosch, E. van Eyk, L. Versluis, V. van Beek, and A. Iosup, "Opendc 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters," in Proceedings of the 21st International Symposium on Cluster, Cloud and Internet Computing, 2021, pp. 455–464.

[195] H. Cao, J. Zhang, L. Chen, S. Li, and G. Shi, "Servlesssimpro: A comprehensive serverless simulation platform," Future Generation Computer Systems, vol. 163, p. 107558, 2025.

[196] R. Cordingly, H. Yu, V. Hoang, D. Perez, D. Foster, Z. Sadeghi, R. Hatchett, and W. J. Lloyd, "Implications of programming language selection for serverless data processing pipelines," in 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). IEEE, 2020, pp. 704–711.

[197] X. Chen, L.-H. Hung, R. Cordingly, and W. Lloyd, "X86 vs. arm64: An investigation of factors influencing serverless performance," in Proceedings of the 9th

International Workshop on Serverless Computing, ser. WoSC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 712.

[198] Z. Guo, Z. Blanco, M. Shahrad, Z. Wei, B. Dong, J. Li, I. Pota, H. Xu, and Y. Zhang, "Decomposing and executing serverless applications as resource graphs," arXiv preprint arXiv:2206.13444, p. 8, 2022.

[199] X. Zhao, S. Yang, J. Wang, L. Diao, L. Qu, and C. Wu, "Fapes: Enabling efficient elastic scaling for serverless machine learning platforms," in Proceedings of the 2024 ACM Symposium on Cloud Computing, 2024, pp. 443–459.