Performant, Secure and Optimized Microservice-based Distributed Systems

by

Changyuan Lin

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Microservices have gained substantial importance over the past decades and matured into one of the fundamental techniques to build performant, cloud-native, and cost-efficient distributed systems that are scalable and highly available. However, like any other type of distributed system, there are inherited pain points related to performance, cost, and security pertaining to microservice-based distributed systems. For applications based on serverless microservices, there is no analytical or experimental tool to enable what-if analysis between performance and cost in a systematic manner. Moreover, production-ready microservice-based systems, e.g., Internet of Things (IoT), are typically large-scale systems for which we require autonomic management to enforce the quality attributes such as security and performance.

In this thesis, we address the challenges mentioned above by targeting four research objectives. More specifically, we propose a solution to help build performant, secure and optimized microservice-based distributed systems, particularly systems based on serverless microservices and IoT systems, from three aspects, including service level agreement (SLA) adherence, performance/cost modeling and optimization, and autonomic security management.

In the first part of the thesis, we initially formulate the function placement problem in which function containers are placed on virtual machines (VMs) optimally to avoid performance degradation due to resource contention. To solve this problem, we design and evaluate a machine learning-based adaptive function placement algorithm that Function as a Service (FaaS) platforms can leverage to improve the throughput of functions and thus enhance SLA adherence without incurring significant overhead.

The proposed algorithm can predict the performance of the function based on the workload profile of functions and performance metrics of VMs. As verified by experimental evaluation, the proposed adaptive function placement algorithm could improve the throughput of serverless functions by 10.35% - 44.89% with negligible overhead.

For performance and cost modeling of applications based on serverless microservices, we first propose a new construct to formally define a serverless application workflow and then implement analytical models to predict the serverless application's average end-to-end response time and cost. Also, we propose a heuristic algorithm with four greedy strategies to answer two fundamental optimization questions regarding performance and cost. The proposed models and algorithms are extensively evaluated by conducting experimentation on Amazon Web Services (AWS). Our analytical models can predict the performance and cost of serverless applications with more than 98% accuracy. Also, the optimization algorithm can achieve the optimal configurations of serverless applications with 97% accuracy on average.

To address security management for microservice-based distributed IoT systems, we strive to build an autonomic manager that can: 1) Monitor the smart space continuously. 2) Analyze the context. 3) Plan and execute countermeasures to maintain the desired level of security. 4) Reduce liability and risks of security breaches. We follow the microservice architecture pattern and propose a generic ontology named Secure Smart Space Ontology (SSSO) for describing dynamic contextual information in security-enhanced smart spaces. Based on SSSO, we build an autonomic security manager with four layers that continuously monitor the managed spaces, analyze contextual information and events, and automatically plan and implement adaptive security policies. As the evaluation, focusing on a current BlackBerry customer problem, we deploy the proposed autonomic security manager to maintain the security of a smart conference room with 32 IoT devices and 66 services encapsulated as microservices. Also, the high performance of the proposed solution is evaluated on a large-scale deployment with over 1.8 million triples.

Preface

The research presented in this thesis has been conducted in the Performant and Available Computing Systems (PACS) Lab led by Dr. Hamzeh Khazaei. This thesis is based on the following three publications.

1. N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu. "Optimizing serverless computing: introducing an adaptive function placement algorithm," in Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, pp. 203-213. 2019.

2. C. Lin and H. Khazaei, "Modeling and Optimization of Performance and Cost of Serverless Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 615-632, March 2021.

3. C. Lin, H. Khazaei, A. Walenstein, and A. Malton, "Autonomic Security Management for IoT Smart Spaces," in ACM Transactions on Internet of Things 2, 4, Article 27, July 2021.

Also, I contributed to the following two publications during my graduate studies, which are not included in this thesis but are related to microservice-based distributed systems and blockchain-based security management for distributed systems, respectively.

1. C. Lin, S. Nadi, and H. Khazaei, "A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub," in 2020 IEEE International

Conference on Software Maintenance and Evolution (ICSME), pp. 371-381, 2020.

2. L. Bindra, C. Lin, E. Stroulia, and O. Ardakanian, "Decentralized access control for smart buildings using metadata and smart contracts," in 2019 IEEE/ACM 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), pp. 32-38, 2019.

Chapter 3 is based on our conference paper [1], in which I was responsible for building the API and middleware of the FaaS platform for evaluation, collecting and processing the performance metrics, and feature engineering for training the performance model. I also contributed to the analysis of evaluation results and manuscript composition. Nima Mahmoudi was responsible for building the machine learning model, developing the rest of the system, the manuscript composition, and other parts of the work. Dr. Hamzeh Khazaei and Dr. Marin Litoiu were the supervisory authors and were involved with concept formation. Chapter 4 and Chapter 5 of this thesis are based on the publication in IEEE Transactions on Parallel and Distributed Systems [2]. I was responsible for the design, implementation, and evaluation of the models and algorithms and the manuscript composition. Dr. Hamzeh Khazaei was the supervisory author and was involved with concept formation and manuscript edits. Chapter 6 of this thesis is based on our collaboration with BlackBerry, which has been published in ACM Transactions on Internet of Things [3]. I was responsible for the design, implementation, and evaluation of the solution. Dr. Hamzeh Khazaei was the supervisory author and was involved with concept formation and manuscript edits. Dr. Andrew Walenstein and Dr. Andrew Malton gave insight about enterprising computing and offered input on the case study from BlackBerry.

Acknowledgements

First, I would like to express my deepest gratitude to my supervisors, Dr. Hamzeh Khazaei and Dr. Marek Reformat, for providing continuous support, profound inspiration, and invaluable guidance throughout my graduate studies. It has been a great honor and privilege for me to have worked under the supervision of such knowledgeable and kind mentors.

My sincere thanks also go to Dr. Sarah Nadi and Dr. Omid Ardakanian for their inspirational lectures and professional guidance on my research. I would also like to thank Dr. Cor-Paul Bezemer and Dr. Lei Ma for accepting to be my thesis examiners.

I would like to thank BlackBerry for their collaboration throughout the research. Special thanks should also go to Amazon Web Services, Compute Canada, and Cybera for providing cloud computing resources that supported part of the experiments in this thesis.

Thanks to all my friends and colleagues in the Performant and Available Computing Systems (PACS) Lab for their valuable help throughout my graduate years. Finally, I would like to thank my parents for their unconditional love, support, and encouragement throughout my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background

1.1 Distributed Systems

Over the past decades, distributed systems have gained substantial importance and have a significant impact on architecture design and software development. A distributed system is a collection of autonomous and interconnected computing elements working together as a single coherent system to accomplish certain collective tasks [4, 5]. Each computing element, generally referred to as a node, is a hardware device or a program process, which can work independently from and concurrent with other computing elements in the system. Such distributed nodes provide great redundancy, allowing a part of the system may fail without affecting the functionality of the whole system. Distributed systems may easily scale by adding and removing nodes on demand for better performance and cost. Distributed systems can be heterogeneous, in which nodes may use various hardware, software components, operating systems, and communication protocols.

There are numerous use cases for distributed systems. Examples include the World Wide Web, IoT sensor networks, distributed file systems and databases, smart grids, cloud computing, and blockchains. With the increasing popularity of distributed systems, specifically cloud computing, new software design patterns and architectures have emerged, such as high-availability architectures, microservices, and serverless computing, dramatically changing the process of application development. While

distributed systems have the potential for better performance, cost effectiveness, scalability, availability, resilience, parallelism, and fault tolerance, they usually have high complexity, heterogeneity, and stochasticity. As a result, there are many challenging issues swirling around the use of distributed systems, such as failure handling, testing, messaging, consistency guarantees, resource scheduling and orchestration, performance and cost modeling and optimization, and security management [4–11].

1.2 Cloud Computing: From IaaS to FaaS

Cloud computing is one of the most common use cases of distributed systems and has drawn tremendous attention from academia and industry over the past decade. It has become one of the fastest growing industries and research hotspots in recent years, changing the way of using and managing distributed computing resources, transforming the process of software development, and enabling many groundbreaking products. The increasing popularity of cloud computing has triggered intense competition among technology giants, including Amazon, Google, Microsoft, and Alibaba. They have already become mainstream cloud service providers by launching cloud platforms, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, and Alibaba Cloud, to offer diverse cloud services and infrastructures to the public.

Evolved out of distributed computing and grid computing, cloud computing generally utilizes virtualized resources backed by large-scale data centers, including CPU, memory, storage, and network, to offer scalable and practically infinite computing resources with flexible tenancy and pay-as-you-go and near real-time billing manner [12]. In other words, cloud computing systems are usually multi-tenant distributed systems. Due to their high scalability, high availability, fault-tolerance, pay-as-you-go billing model, and low management overhead, leveraging cloud solutions has become a natural way for developers and companies to build, test, and deploy new applications and services. A large number of businesses are also planning to migrate their services to

the public or on-premises private cloud.

Today's cloud computing generally has four paradigms, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and Function as a Service (FaaS).

IaaS is the most basic and flexible cloud computing model. IaaS providers, such as Amazon EC2 and Google Cloud Compute Engine, rent out computing resources together with the storage and network services to customers. While on-demand operating system (OS) instances in the form of virtual machines (VM) are the primary deployment unit in IaaS, dedicated (bare-metal) machines could also be outsourced [13]. Service providers leverage IaaS to deploy their applications in an environment with complete control over infrastructure, OS, middleware, dependencies, programming, data, runtime environment, and applications in an automated way.

PaaS functions at a higher level than IaaS. PaaS providers take responsibility over OS, middleware, and runtime environment, and rent out the ready-to-use platform on which customers can develop and deploy their services. As a result, the PaaS customer can not manage the underlying infrastructure (typically VM and OS) but has full control over their deployed applications and sometimes limited control of storage and networking. AWS Elastic Beanstalk and Google App Engine are two examples of popular PaaS solutions.

SaaS functions at an even higher level than PaaS, providing a ready-to-use packaged application to customers, usually the end-users of the software. SaaS providers manage nearly everything required for running the hosted application and handle software maintenance, such as application updates, security patches, and bug fixes, eliminating the overhead of managing, installing, and using software in the local environment. End-users can rent and use the service online, slashing the expense of distributing and using the software. Some of the most common types of SaaS applications include video conferencing services and cloud storage services.

FaaS is a relatively new type of cloud computing paradigm that allows develop-

ers to execute event-based functions in the cloud. The function is a chunk of code that abstracts a part of an application implementing business logic. FaaS providers take over all the operational responsibilities, such as function deployment, resource management, scaling, and monitoring. Thus, developers can mainly focus on the business logic of functions, expediting the application development. Many cloud service providers have launched FaaS platforms, including AWS Lambda, Google Cloud Functions, and Azure Functions.

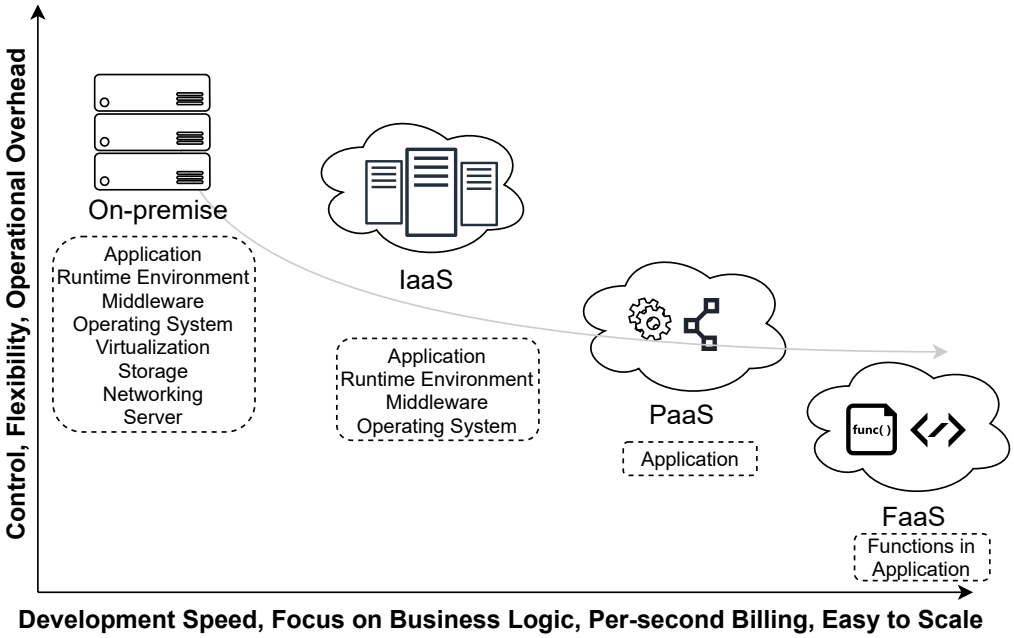


Figure 1.1: Comparison of different cloud computing models.

For application developers, IaaS, PaaS, and FaaS are three primary cloud models that they may leverage to develop and deploy their native cloud applications. Figure 1.1 demonstrates the comparison among different cloud computing models regarding control, flexibility, operational overhead, development speed, focus on business logic, scalability, and billing granularity. Items in the boxes for each paradigm represent the responsibility that developers should take. As shown in Figure 1.1, product providers take the most responsibilities from the underlying hardware to the hosted application when leveraging the on-premise infrastructure. While the onsite

paradigm usually incurs significant operational overhead, as users have full control of every single layer in the computing infrastructure, such infrastructure could ensure data integrity and security in some cases.

By leveraging hypervisor-based virtualization, software-defined networking (SDN), cloud storage (typically block and file storage), and well-defined application programming interface (API), IaaS providers host and rent out infrastructure components in the cloud with per-second to per-hour billing models. Customers outsource on-demand cloud instances, networking, and storage resources with different flavors to deploy the application while managing OS, middleware, and runtime by themselves. As presented in Figure 1.1, while the control over resources is somehow limited compared to the on-premise infrastructure, IaaS gives users a high level of flexibility for customizing the operating system, middleware, and runtime environment for better fit, which could yield high isolation levels, performance, and availability. Besides, by launching/removing VMs and changing instance flavors (e.g., vCPU, memory, and storage), resources can scale horizontally and vertically with lower overhead costs.

PaaS further eliminates the hassle of managing runtime, OS, and underlying infrastructure for developers, allowing them to develop and deploy applications on the platform directly. Most popular PaaS platforms offered by public cloud providers leverage lightweight container-based virtualization, particularly Docker [14], to deploy the application container that packages up the application together with its dependencies. Containers can provide isolation and resource allocation characteristics, and can be deployed and removed in seconds in most cases, enabling per-second billing. By breaking down the application or the system into different components packaged in different containers and adopting microservice architecture, containerization and PaaS platforms could also help developers develop, deploy, refactor, and scale parts of the application easily, increasing the overall resilience and scalability of the system [13]. We discuss containers and microservices further in Section 1.3 and Section 1.4.

FaaS allows developers to build, execute, and manage stateless and event-driven functions that abstract parts of the business logic in the cloud. Like container-based PaaS, FaaS also leverages containers to execute functions, but the containerization, resource allocation, and infrastructure are fully managed by the service provider. FaaS platforms take over all the operational responsibilities such as function deployment, resource management, scaling, and monitoring, allowing developers to focus on writing business logic (functions). The event-triggered container is easy and fast to scale in response to incoming traffic and can scale to zero for cost efficiency. The per-millisecond billing granularity of FaaS can help further reduce the operational overhead [15]. Regarding flexibility, FaaS customers usually have very limited control over resources allocated to containers. FaaS providers usually offer other managed services, such as API gateway, message queue, and workflow coordinator, to help customers build applications with serverless architecture. In other words, FaaS is the core of serverless computing. Section 1.5 further discusses the background of serverless computing.

As illustrated in Figure 1.1, each paradigm represents a different part of cloud computing with different levels of abstraction, virtualization, control, flexibility, and management. Different cloud computing paradigms use different management, deployment, control, and virtualization strategies to address different needs in different scenarios. It is worth acknowledging that these paradigms are co-existent, and there is no evolutionary relationship among them. Therefore, we do not claim that one paradigm is necessarily better than another. However, in recent years, there has been a growing trend in the paradigm of cloud computing shifting from IaaS to FaaS and serverless for developing cloud-native applications because of the reduced complexity, accelerated development speed, improved dependability, auto-scaling, well-maintained service level agreement (SLA), and low overhead on infrastructure management [16, 17].

1.3 Containers

Containerization is widely used in cloud computing [18, 19] and in applications with microservice architecture [20], which is the process of encapsulating source code with OS, environment variables, dependencies, and libraries required to execute the code into a lightweight, isolated, ephemeral, and executable container. As containers share the host kernel and use cgroups and namespaces to achieve lightweight virtualization, containers only isolate system processes and resources and are easy to deploy, scale, and migrate. Compared to traditional VMs, containers generally have considerably lower overhead of runtime management and scaling [21].

Docker is currently one of the most popular containerization solutions [22]. An application encapsulated by Docker is distributed in the form of a Docker image. The Docker image is defined through a Dockerfile, which contains all commands and environment variables required to run the application. When the target application encapsulated in the image is deployed, an instance of the image is created [23] and is referred to as the Docker container. Hence, a Docker image is like a snapshot of the target application and allows creating containers in a reproducible way. One of the advantages of the Docker ecosystem is that it is not necessary to write an entirely new Dockerfile to develop a new Docker image. Similar to inheritance in object-oriented programming, a Docker image can inherit image definitions from another base image by using the *FROM* command. In this case, all properties and files encapsulated in the base image are inherited by the new image.

When building an image, Docker executes statements from the Dockerfile and generates a layer for each instruction. Similar to git commits, each layer contains only a collection of differences from the previous layer. The Docker image is built from a massive pileup of layers. Each Docker image has a unique SHA-256 code as its ID. For ease of versioned image development and management, Docker provides a tagging mechanism where image developers can provide tag names for each version,

which could also help versioning management in services based on containers.

Container orchestration is the automated process of provisioning, deploying, scheduling, managing, networking, and scaling containers. Proper container orchestration is crucial, especially in large-scale production environments, because of the increasing proliferation of containers, large-scale distributed infrastructure, rapid iterations, and DevOps pipelines. There are many container orchestration solutions for managing containers at scale, such as Kubernetes, Docker Swarm, and Red Hat OpenShift. With these orchestration solutions, developers can easily develop, deploy, and manage containerized applications and microservices with high availability and scalability.

1.4 Microservices

Microservices, or microservice architecture, have become tremendously popular over the past decade. Microservices are a modern architectural pattern in which an application is composed of many small distributed services that can be deployed and function independently, communicate through protocols and work together seamlessly. Hence, microservices are a suitable approach to implement a distributed system, and the application based on microservices is naturally a distributed system where each service is a fundamental computing element.

While microservices can be deployed in a variety of ways, containers are an ideal deployment unit. As discussed in Section 1.2 and Section 1.3, compared to IaaS, containers can scale easily with considerably lower latency and footprint in terms of hardware resources. In the microservice architecture, each service is usually encapsulated into and running in a container. There are also techniques offering isolated runtime with even lower overhead by leveraging Unikernels [24] and processes [25]. The small footprint of containers makes them possible to run on hardware with limited resources, such as IoT devices.

As a distributed system, the application based on microservices has inherited advantages in terms of scalability, performance, availability, and fault tolerance, com-

pared with monolithic applications. Furthermore, the DevOps process benefits from microservices. As the microservice architecture decouples the application and each service is independently deployable and scalable, developers can develop, refactor, monitor, and maintain a single part of the application without worrying about the rest part.

Microservices have become the core of developing cloud-native applications, allowing developers to build applications that are scalable and resilient, have a large scale, and meet the needs of rapid iterative development. Besides web-based systems and cloud applications, other distributed systems could also benefit from the microservice architecture, including IoT systems [26, 27]. Microservices and IoT systems already share some ties, such as heterogeneous and distributed hardware, embedded, self-contained and granular services, and technology stacks related to cloud computing. There is a fast growing trend of containers with non-AMD64 architectures (e.g., ARM) that are frequently used by IoT devices, which may suggest microservice-based IoT systems have been gaining popularity [28].

With the advent of serverless computing, serverless functions fully managed by FaaS platforms have become a popular way to host microservices in recent years. We refer to microservices hosted by serverless functions as serverless microservices. A serverless function is a relatively small bit of code that abstracts a small part of an application and performs one event-driven action. Therefore, the serverless paradigm generally enables a more granular level for microservices. An application can be referred to as serverless if it is built with serverless microservices. In this thesis, we use serverless functions and serverless microservices, and serverless applications and applications based on serverless microservices interchangeably. Serverless computing, FaaS, serverless functions, and serverless applications are further discussed in Section 1.5.

1.5 Serverless Computing

1.5.1 FaaS and Serverless Paradigm

As an emerging cloud computing paradigm, serverless computing is yet another approach to build distributed systems and has transformed the way how developers build and manage cloud-native applications. Under FaaS, developers mainly focus on writing source code for each function that abstracts a part of an application implementing business logic. The notion of function in FaaS is similar to a function in functional programming or a method in object-oriented programming, responsible for handling one task required by the application statelessly. The source code of the function and its dependencies are packaged together, and the function is running in an ephemeral isolated environment (e.g., Docker container).

AWS launched a computing service called Lambda in 2014 [29], which could store, package, and deploy functions uploaded by users, handle events and requests (triggers) of functions, monitor the resource usage, and autoscale function containers correspondingly. This was the first time that the function execution was offered as a cloud service. Today, besides AWS Lambda, many FaaS platforms with similar functionalities are provided by public cloud service providers and open-source communities such as Google Cloud Functions, Azure Functions, Apache OpenWhisk, and OpenFaaS.

Figure 1.2 depicts the high-level overview of FaaS platforms. Functions hosted on FaaS platforms can be triggered by various event sources, such as HTTP requests, messages queues, and cron jobs. The API gateway (or any other type of gateway) processes the event, finds the requested function from the function store, and sends it to the container management platform together with its payload. If there is already a container comprising the requested function, the function call will happen in the existent container. If there is not, the platform will first create a new container, inject the function, configure the environment, and then execute the function. Such

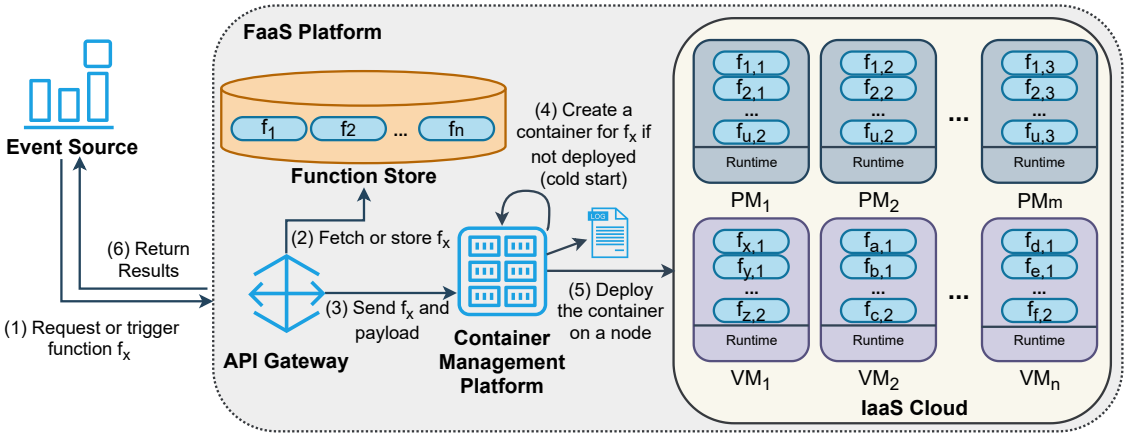


Figure 1.2: The high-level overview of FaaS platforms.

processes are known as warm start and cold start, respectively. The results will be returned through the API gateway after the function execution is completed.

As FaaS platforms take over operational responsibilities, such as function deployment, container orchestration, resource management, and monitoring, besides uploading the source code of functions, users have limited control over resources on FaaS platforms. Taking AWS Lambda as an example, the amount of allocated memory during execution and the concurrency level are only options for tuning the performance of functions. The amount of allocated memory is between 128 MB and 3,008 MB in 64MB increments [30]. Previous researches have proven that computational power and network throughput are in proportion to the amount of allocated memory, and disk performance also increases with larger memory size due to less contention [31, 32]. By reserving and provisioning more instances to host functions, a high concurrency level can decrease fluctuations in the function performance incurred by cold starts (container initialization provisioning delay if no warm instance is available) and reduce the number of throttles under very heavy request loads [33].

FaaS platforms also introduce a new GB-second billing model depending on the allocated memory size, function duration, and the number of invocations. For example, AWS Lambda charges $0.000016667 for every GB-second and $0.20 per 1M

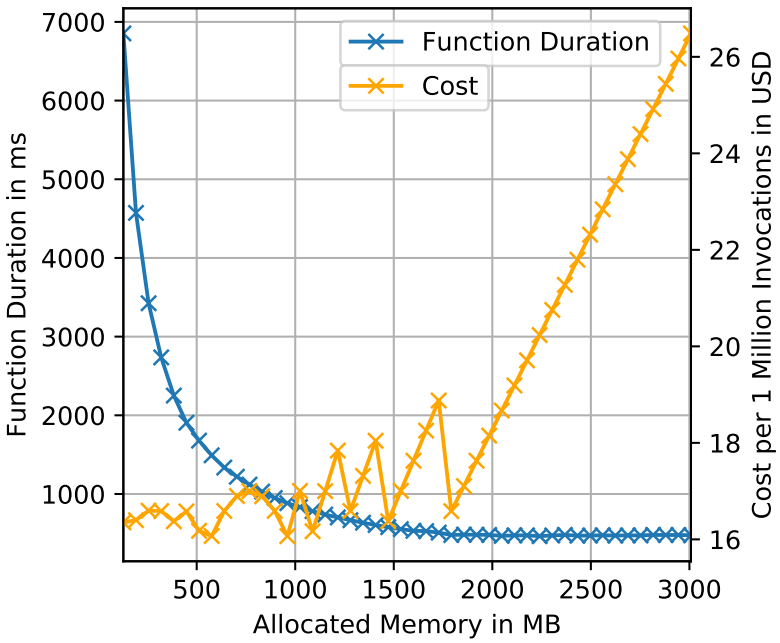


Figure 1.3: The cost and performance with regard to the amount of allocated memory of a CPU-intensive function (hashing) deployed on AWS Lambda.

function invocations. The billed duration is the function duration rounded up to the nearest 100ms and metered at a granularity of 100ms [34]. Due to the rounding and billing granularity, the cost fluctuates erratically as the memory size changes, as our experiment results shown in Figure 1.3, in which the blue line is the performance-memory curve of the function. In other words, the allocated memory size acts like a tuner tuning the performance (function execution duration or function response time) and cost of functions hosted on FaaS platforms. The figure is further discussed in Section 5.1. The new billing model and such irregularities make the modeling and optimization problems non-trivial.

By leveraging FaaS, application developers can decouple the business logic into a group of functions hosted on FaaS platforms without having to manage the underlying infrastructure. Such a way to develop cloud-native applications is the serverless paradigm. As FaaS platforms shift most operational responsibilities to cloud service providers, the overhead of server provisioning, management, monitoring, failover, and

scaling is eliminated, like there are no servers to manage at all, namely serverless. While serverless is a superset containing a series of fully-managed cloud services, FaaS and serverless are often conflated. We use them interchangeably in this thesis.

1.5.2 Serverless Application and Workflow

Similar to microservices, serverless applications are inherently distributed. The serverless application decouples its business logic into a group of serverless functions hosted on FaaS platforms and leverages necessary cloud services such as bucket storage, message queue, and pub/sub messaging service to build a stateless and event-driven software system [17, 35]. In general, the serverless workflow is the orchestration of functions in the serverless application to implement the entire business logic. AWS defines that the serverless workflow describes a process as a series of individual functions and coordinates them [36]. To complete the business logic of the application, interactions among decoupled functions are indispensable, leading to different structures in the workflow graph. As shown in many examples of serverless applications deployed in the production environment, there can be four types of structures in the serverless workflow, including parallel, branch, cycle, and self-loop [37–39].

In most cases, a coordinator is required to chain together components of the application, handle events among functions, and trigger functions in the correct order defined by the business logic. Typically, a message queue, a pub/sub messaging service, an event bus, or a workflow coordinator like AWS Step Functions Express Workflows works as such a coordinator [11, 35, 40]. While serverless functions are stateless by the very design, many serverless function orchestration services can coordinate serverless functions and integrate them into a serverless workflow abstracted as a state machine, making it possible to handle complex tasks. Figure 1.4 illustrates an example of a serverless application composed of seven functions. The two numbers on each function represent the response time and allocated memory, respectively. P and D are the transition probability function and the delay function, which are detailed

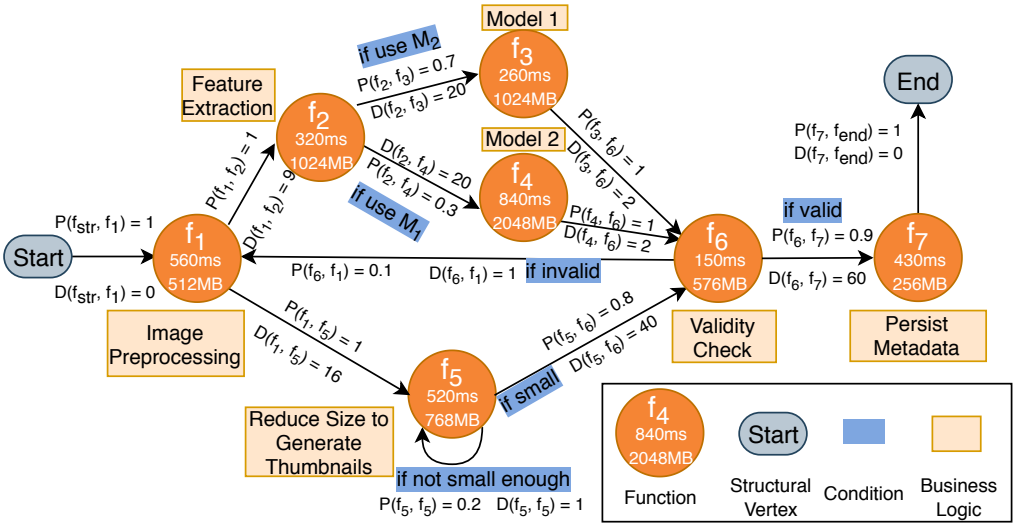


Figure 1.4: Workflow of a serverless image classification application composed of seven FaaS functions.

in Chapter 4.

1.6 Motivation

Performance, cost, and security are key concerns inherent in any type of distributed system, hindering its wide adoption by potential users. Cloud computing, including serverless computing, is one of the most popular use cases of distributed systems. Also, microservice architecture is a widely used approach to build distributed systems. Furthermore, the Internet of Things (IoT) is a cyber-physical ecosystem with distributed interconnected devices and has become an indispensable factor in many areas of daily life. Therefore, in this thesis, our focus is on addressing such key challenges in distributed systems in the form of serverless microservices and IoT systems with microservices.

Specifically, the motivation of this thesis is fourfold.

1. The performance of a single computing node is one of the most crucial factors influencing the performance of distributed systems. This is true especially for multi-tenant distributed systems, including cloud computing and serverless

computing platforms, where the system size is somehow limited or even fixed for customers due to cost considerations. For distributed computing service providers, SLA adherence in terms of performance is one of the critical competitive advantages. For example, the throughput and response time of functions are important aspects of serverless computing, which largely determine the performance of applications based on serverless microservices. However, when it comes to the placement of function containers (or any other type of sandbox), many FaaS providers leverage simple spread and bin packing algorithms to place containers on VMs without considering the type of the workload and their performance implications when placed on the same infrastructure [31]. For instance, deploying too many function containers with CPU-intensive workloads on the same node could degrade the throughput of serverless functions. The existing function placement algorithms could result in insufficient performance isolation and compromise the ability of FaaS platforms to deliver on their performance commitments. Therefore, an adaptive serverless function placement algorithm is required to improve SLA adherence.

2. Performance modeling is essential when designing and implementing distributed systems. Also, for multi-tenant distributed systems, including FaaS platforms, the infrastructure where serverless microservices are usually deployed, the cost is one of the key concerns for their users. As the serverless application is emerging as a new type of distributed system, performance and cost modeling is crucial for developers to design, configure, and test serverless applications properly. However, the high complexity and the opacity of the underlying infrastructure make performance modeling of serverless applications a challenging task. FaaS platforms also introduce new billing models. Many studies have mentioned the lack of performance and cost models for serverless applications as one of the challenges in serverless computing [11, 41, 42]. New performance and cost

models designed for serverless applications are highly demanded.

3. Besides modeling performance and cost of serverless applications, the trade-off analysis between the performance and cost of serverless applications is also essential [42, 43]. Currently, developers have to test the serverless application under numerous configurations and cope with the performance and budget constraints. Such a testing process is usually extremely time-consuming and financially unacceptable. Performance and cost optimization is a non-trivial and necessary step towards guaranteeing the SLA of serverless applications in an economical manner, which is basically finding an acceptable trade-off between performance and cost.

4. Security is one of the major challenges in distributed systems [4–6, 10]. Similarly, while IoT devices can enable intelligent processes, the security of the IoT system has always been an issue hindering its widespread application in critical businesses [44]. Distributed systems built on IoT devices are vulnerable to different types of attack due to high exposure, numerous attack surfaces, limited computational resources, and low reliability. As a distributed system deployed in a real-world environment, security breaches in IoT systems can lead to severe consequences [45]. Therefore, distributed systems built on IoT devices should have fine-grained security management components. Also, security management of IoT systems should require minimum human intervention since IoT systems are usually deployed at a large scale and manually handling numerous possible risks in the environment is practically impossible. Hence, autonomic security management is indispensable for IoT-based distributed systems.

1.7 Objectives

Given the aforementioned motivations, there is a clear gap in research regarding SLA adherence, performance and cost modeling and optimization, and autonomic security

management, which leads us to four research objectives. Specifically, in this thesis, we aim to:

- **Objective 1:** Optimize the placement process of serverless functions to mitigate resource contention and improve throughput without incurring significant overhead for better SLA adherence.
- **Objective 2:** Build performance and cost models that can obtain the end-to-end response time and cost of a serverless application when orchestration and configuration of the application are given.
- **Objective 3:** Design an optimization algorithm for calculating the best performance under a given budget and the minimum cost for a desired performance for a serverless application.
- **Objective 4:** Develop a security management solution that can mitigate threats and enhance security for distributed IoT systems in an autonomic manner.

1.8 Contributions

The main contributions of this work are four-fold, benefiting both microservice-based distributed system users and distributed computing service providers.

1. To tackle the unoptimized function placement problem, we design and evaluate a machine learning-based algorithm to optimize the placement of function containers of serverless applications without incurring significant performance implications for hosted applications and delay for the container orchestration process. With practically negligible additional overhead, the proposed function container placement algorithm improves the throughput by 10% to 36% and reduces the response time by 9% to 32%.
2. To resolve the performance and cost unpredictability of serverless applications, we propose performance and cost models to accurately get the average end-to-

end response time and cost of serverless applications with parallels, branches, cycles, and self-loops in their workflows. The proposed models can precisely give the end-to-end response time of serverless applications with over 98% accuracy and estimate the average cost with an accuracy of over 99%, independent of the complexity of their workflows.

3. To address the trade-off between the performance and cost of serverless applications, we present a heuristic algorithm for optimizing the performance and cost of serverless applications, which can yield the optimal configurations of functions achieving the best performance for a given budget and the minimum cost satisfying a given performance. The proposed optimization algorithm can effectively and efficiently resolve two types of optimization problems with over 97% accuracy.

4. To achieve autonomic security management, we first propose an ontology with four features for describing environments with distributed IoT systems, which can facilitate analysis and reasoning about the current state of the space in a machine-understandable fashion. Then, we implement an autonomic security manager with four layers that can monitor and analyze events and context, and plan and execute adaptive countermeasures with minimum human intervention at a large scale.

All scripts, algorithms, and experimental results proposed and discussed in this thesis are available in the artifact repositories [46–48].

1.9 Thesis Outline

The rest of the thesis is organized as follows: In Chapter 2, we survey and discuss the latest literature related to research objectives. In Chapter 3, we optimize the process of serverless function placement and achieve Objective 1 by proposing and evaluating

an adaptive serverless function placement algorithm. Chapter 4 presents the definition of the serverless workflow and performance and cost models to accurately predict the end-to-end response time and cost of serverless applications, solving Objective 2. In Chapter 5, we address Objective 3 by proposing a heuristic-based performance and cost optimization algorithm for serverless applications, which can solve two optimization problems. In Chapter 6, we accomplish Objective 4 by designing an ontology for formal description of the IoT environments and developing an autonomic security manager with the MAPE-k method. Chapter 7 discusses and concludes the thesis and presents some potential future directions toward optimizing microservice-based distributed systems.

Chapter 2

Literature Review

In this chapter, we review the literature related to four research objectives. Specifically, we survey and discuss the research work in the areas of FaaS and serverless computing (related to Objective 1, 2, and 3), performance modeling for cloud computing (related to Objective 2), workflow scheduling and optimization of distributed applications in the cloud environment (related to Objective 3), and microservice-based IoT systems and ontology-based security management (related to Objective 4).

2.1 Serverless Computing and FaaS platforms

Cloud computing, including serverless computing, has become one of the most popular use cases for distributed systems. Many cloud service providers have launched their FaaS platform, including AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions in recent years. Since then, serverless computing has become an emerging technique and research topic in both industry and academia. There have been efforts made to topics including profiling FaaS platforms [19, 31, 49], developing new serverless applications and platforms [50–52], migrating to serverless [53, 54], and designing new software engineering methodology [55, 56].

Wang et al. [31] evaluated three FaaS platforms, namely AWS Lambda, Azure Functions, and Google Cloud Functions, from various aspects, including underly-

ing architectures, performance isolation, scalability, cold start latency, and resource management. They confirmed the CPU power is in proportion to the amount of allocated memory, and disk performance and network throughput also increase with larger memory size due to less contention. They also observed that the underlying infrastructure of FaaS platforms is heterogeneous and cold starts could incur significant overhead. Besides, this work revealed the lack of performance isolation between function containers and the co-residency of functions, causing high variance (14.1% - 90%) in the CPU utilization rates of functions.

Lee et al. [19] studied the performance and overhead of running concurrent serverless functions, and cold start behaviors on four FaaS platforms, and compared the serverless paradigm with VM-based IaaS solutions. They found that AWS Lambda outperforms other FaaS platforms in terms of CPU power, and network and disk throughput when processing dynamic concurrent invocations. Compared to VMs, serverless computing could save cost for processing short-lived tasks.

Figiela et al. [32] conducted performance tests by deploying a serverless workflow on four FaaS platforms and measured the data transfer delay between components and the container lifespan. Their results again confirmed the CPU performance and network throughput of serverless functions is proportional to the allocated memory size on FaaS platforms, including AWS Lambda and Google Cloud Functions. But overheads, such as network latency, routing, and scheduling, are independent of the memory size. They also again observed that serverless functions are deployed on heterogeneous hardware.

There have been some works on the evaluation and performance profiling of open-source FaaS platforms, including Apache OpenWhisk [57], OpenFaaS [58, 59], and Kubeless [58, 59]. Djemame et al. [57] extensively evaluated the performance of Apache OpenWhisk in terms of effectiveness and efficiency and answered three research questions. Their results demonstrated OpenWhisk performs better than the Docker solution that is mimicking the underlying operation of OpenWhisk in terms

of CPU, memory, and network, suggesting OpenWhisk may orchestrate containers optimally. Mohanty et al. [58] observed that OpenFaaS is the most flexible and extendable FaaS platform, and Kubeless has the most consistent performance because of its simple architecture. Balla et al. [59] compared the performance of functions written in three programming languages on four open-source FaaS platforms. They found that the optimum platform is varying under different workloads.

Jackson et al. [60] investigated the impact of programming languages on the duration and cost of FaaS functions. Jonas et al. [17] discussed inadequate storage, high communication overhead, high cold start latency, and lack of predictable performance and cost of applications. Shahrad et al. [61] found the containerization incurs huge overhead and cold start brings the latency as high as ten times of a small function's execution time. Hellerstein et al. [62] investigated cold start, limited storage, and communication overhead by case studies in distributed serverless computing. Daw et al. [63] made efforts to solve the cascading cold start problem for serverless applications by inferring the most likely execution path in the workflow and provisioning relevant resources in advance.

Several works presented a set of key challenges for serverless architecture. The challenge that many studies have mentioned is the unpredictable performance and cost due to the lack of performance and cost models [11, 17, 41, 42].

Eivy et al. [41] found that the serverless paradigm could cost up to three times as much as conventional VM-based solutions under certain scenarios. The cost modeling of serverless applications can be a very complex and difficult task because of the new billing model with small granularity and multiple variables. They still highly recommended that developers should model the cost of their applications hosted on FaaS platforms to avoid surprise bills before migrating to serverless.

Baldini et al. [11] claimed that the serverless function is potentially suitable to handle bursty, unpredictable, short-lived, and CPU-intensive workloads because it is easy to scale and does not require paying for idle instances. While FaaS platforms

take on more operational responsibilities compared to IaaS, developers have to spend more effort on modularizing applications and designing event-driven patterns. They also stated cost modeling of serverless applications as one of the key challenges that need to be investigated.

Eyk et al. [42] identified several challenges and opportunities for serverless computing, including resource management, performance and cost, data privacy, and compatible APIs. Because of the fine-grained billing model, for serverless users, there is an urgent need for new models and tools to solve the predictability problem of performance and cost and the trade-off between them. They stated that new performance and cost models and analysis on optimizing trade-offs between performance and cost are among the top five obstacles for serverless computing.

In this thesis, we aim to mitigate resource contention and improve performance for better SLA adherence by designing a machine learning-based algorithm to optimally place function containers (Chapter 3). To solve the unpredictable performance and cost issues, we propose performance and cost models to accurately get the end-to-end response time and average cost of serverless applications (Chapter 4). We also present a heuristic algorithm to help developers navigate the trade-off between performance and cost (Chapter 5).

2.2 Performance Modeling for Cloud Computing

While performance modeling is one of the challenging issues for distributed systems due to their high complexity, the problem of modeling the performance of cloud architectures and applications has been extensively investigated over the past decade. Quality of service (QoS) requirements, including response time, availability, reliability, and monetary cost, are the main focus of such models.

The queuing theory has been used to predict such QoS metrics for cloud systems. Yang et al. [64] modeled the cloud center as an $M/M/m/m + r$ queuing system and obtained the probability density function of response time and the average value.

Khazaei et al. [65] obtained the distribution of response time and blocking probability with regards to the number of servers and buffer size for a cloud center using a semi-Markov $M/G/m/m+r$ queue, where r is the buffer size for the incoming job. Using the task blocking probability and mean response delay as indicators, this work has been extended to predict the availability of cloud systems later [66]. Li et al. [67] leveraged analytical models based on $M/M/c$ queues to optimize performance levels of composite service application jobs by tuning configurations and resource allocation decisions. Vilaplana et al. [68] utilized $M/M/1$ and $M/M/m$ queues to model a single entry server and processing nodes on the cloud to get the total response time with regards to the service rate of processing nodes.

Petri net has been proven to be an effective formalism for modeling distributed systems with concurrency and synchronization in the cloud. Chen et al. [69] leveraged a deterministic and stochastic Petri net to illustrate the performance of producer/consumer-based application models in the cloud environment. Alansari et al. [70] leveraged a colored Petri net to model the energy and transmission costs along an execution path and find a suitable migration policy. Rygielski et al. [71] used queuing Petri nets to predict throughput between network nodes with an accuracy of 95% for underutilized links and 74% for heavily utilized links. Cao et al. [72] developed an evaluation model based on Queuing Petri Net to model the throughput of cloud systems with three different architectures. Rista et al. [73] introduced a generalized stochastic Petri net model, composed of the combination of timed and non-timed Petri nets, to predict the throughput and latency of the network in container-based cloud environments.

Machine learning has also been applied to performance modeling for cloud computing. Xu et al. [74] leveraged unified reinforcement learning to enable auto-configuration of VMs to provide service quality assurance with adaptive budget and configurations under varying workload. Their approach could also solve the trade-off between utilization objectives and performance guarantees. Kundu et al. [75] proposed the methodology to predict the performance of the application hosted on VMs based

on the allocated resources and resource contention by using artificial neural network (ANN) and support vector machine (SVM), with the median of the prediction error of 4.36%. Didona et al. [76] studied several hybrid box techniques that leverage analytical modeling and machine learning in synergy to predict the performance of Total Order Broadcast service and NoSQL database service. Witt et al. [77] studied the features that are required to build performance models, such as execution duration, memory usage, and wait times. They extensively surveyed numerous approaches for predictive performance modeling, including statistical analysis and neural networks. Eismann et al. [78] applied machine learning techniques to predict the response time of serverless functions that are input-parameter sensitive and the total cost of the serverless workflow based on input parameters.

These works on performance modeling either assumed the cloud system is designed homogeneously or required the parameters relevant to the underlying resources and incoming requests, which are not reasonable enough for the serverless paradigm because of the infrastructure-agnostic and closed-source platform, event-driven and highly decoupled software architecture, auto-scaling resources, and elimination of resource management. While many research works have proven that workflows based on the directed acyclic graphs (DAG) and Petri net are effective for performance and cost modeling of systems in parallel, distributed, and scientific computing. However, they are not favorable for modeling serverless applications, as cycles and loops are not allowed, and cause the state explosion problem without efficient solutions in Petri nets. The disadvantages of Petri nets also lie in their high complexity and limited support of non-functional requirements in cloud computing [79]. Besides, these approaches also focused more on the performance analysis, and most of them did not model the cost. In this thesis, we fill these gaps by proposing a definition of the serverless workflow and performance and cost models that are compatible with the current serverless paradigm (Chapter 4).

2.3 VM Placement and Workflow Scheduling

The function placement issue (Objective 1) resembles the virtual machine placement (on physical machines) and workflow scheduling problems in nature. Also, cloud-based workflow scheduling solutions can shed some light on performance and cost modeling and optimization of serverless applications (Objective 2 and Objective 3). Over the past decade, both topics have been extensively studied.

Calcavecchia et al. [80] proposed a technique named Backward Speculative Placement that optimizes VM placement under a continuous stream of deployment requests by migrating VMs on the most loaded physical machine (PM) to all other PMs. They designed a demand risk measuring the risk of demand, and the proposed algorithm could select an optimum migration achieving the least dissatisfaction risk based on historical workload data. Chowdhury et al. [81] studied and evaluated five different bin packing solutions using real-world workload traces, considering power consumption, SLA violation, performance degradation incurred by migrations, and the number of migrations. Alam et al. [82] presented a multi-objective VM placement algorithm that maximizes the reliability of VM placement, minimizes communication delay, and solves the trade-off between reliability and delay. The proposed model could achieve up to about 90% of the optimal solution and improve the reliability by up to 15%. Ghobaei-Arani et al. [83] proposed a VM placement algorithm based on the best fit decreasing algorithm to reduce energy consumption and SLA violations. Qin et al. [84] leveraged multi-objective reinforcement learning to design a VM placement algorithm that minimizes energy consumption and resource wastage simultaneously.

Besides performance, power consumption is also a key focus while designing VM placement algorithms. Energy consumption is one of the optimization goals in works presented by Chowdhury et al. [81], Ghobaei-Arani et al. [83], and Qin et al. [84]. Khosravi et al. [85] identified the most important factors affecting energy consumption and carbon footprint. Then, they designed a dynamic VM placement algorithm that

could measure the power usage effectiveness and reduce operational costs of data centers while meeting performance guarantees. Ismail et al. [86] focused on VM placement in cloud-IoT computing systems and extensively classified and compared 13 different algorithms in a unified setting. Ibrahim [87] presented PAPSO, a power-aware technique based on particle swarm optimization. PAPSO could minimize the number of active PMs with the major constraint to decrease the number of overloaded servers while satisfying SLAs.

Workflow scheduling is an extensively studied topic in cloud computing, which aims to optimally allocate computing resources to inter-dependent tasks in the workflow with the consideration of constraints, such as SLA and cost. Abrishami et al. [88] presented a Partial Critical Paths algorithm to solve the best cost under the performance constraint problem for QoS-based workflows on the utility grid. Later, they extended the work by considering several cloud features such as the pay-as-you-go and duration-based billing model and applied the algorithm to a workflow instance on IaaS clouds to solve the same problem [89]. Lin et al. [90] proposed a Critical-Greedy algorithm to solve the best performance under the budget constraint problem for scientific applications. Faragardi et al. [91] proposed a Greedy Resource Provisioning with the consideration of heterogeneous cloud resources and the efficiency rate of instances to solve the best performance under the budget constraint problem for workflow applications on IaaS Clouds. Bao et al. [92] designed a Greedy Recursive Critical Path algorithm to find the configuration that achieves the best performance under the budget constraint for microservice-based applications on the cloud.

The research work presented in this thesis (Chapter 4 and Chapter 5) differs from the previous work on workflow scheduling at least in the following aspects: 1) We consider the new features in the serverless computing paradigm such as the memory-dependent performance rules, GB-second billing model, independently operated components, and event-driven architecture when modeling applications based on serverless microservices. 2) The serverless workflow is not confined to DAGs, and cycles

and self-loops are allowed to appear in the workflow. 3) We propose a Probability Refined Critical Path Algorithm to solve the trade-off between performance and cost for serverless workflow-based applications.

2.4 Microservice-based IoT and Ontology-based Security Management

Security and heterogeneity are some of the major challenges in distributed systems like IoT. Proper security management should be a key element when deploying IoT systems in critical businesses. Making improvements in IoT architecture is one of the essential ideas for tackling heterogeneity, improving security, and easing management. Recently, several works focusing on microservice-based IoT architecture have emerged. Following the view of microservice-based IoT systems, different IoT devices can be viewed as independent microservice providers, and we can leverage some microservice patterns to manage the IoT system.

Butzin et al. [26] proposed a microservices approach for the IoT to demonstrate how operating-system-level virtualization and open service gateway could ease service deployment and improve scalability and testability. Lu et al. [27] proposed a secure microservice framework for IoT. They considered the IoT system as a service-oriented system of many microservices and adapted some technical patterns widely used in web-centric systems for IoT systems, such as API, SDN, containers, and access control, to enable high-level development and integrated security policies. Based on such an architecture, Lu et al. [93] extended their work and developed a prototype of autonomous vehicles management system, which could help several vehicles form a physically-local chain, and maintain close proximity while traveling down a road. While most of the papers in this area just covered some basic designs and ideas of an IoT system with microservice architecture, such an implementation illustrated the feasibility of implementing microservice-based IoT systems in the production environment. Sun et al. [94] proposed an open IoT framework based on the microservice

architecture, which has nine components responsible for different functions. Different from directly leveraging microservice patterns, they first analyzed the possible functions required and provided by IoT devices and extracted nine general components. Additionally, they also considered artificial intelligence, big data, and tenant services of IoT systems in the framework design.

An ontology is defined as a collection of high-level primitives which capture and model a knowledge domain [95]. The ontology usually adheres to the Resource Description Framework (RDF) data model [96], which utilizes a human-and machine-readable graph, expressed as a collection of triples, to represent the knowledge. During recent years, several works aiming at using ontology to ease the management of smart spaces have been published. Ontology has been proven to be an effective solution to tackle the heterogeneity and enable interoperability in IoT systems [97] [98]. However, most of the ontologies proposed in this domain are focusing on either the context of smart space or human actions in the smart space. Only a very small proportion of work studies security management in smart spaces. These existing ontologies also fail to follow the changes in the IoT architecture, which is more service-oriented nowadays.

Latfi et al. [99] proposed an ontology to describe the telehealth smart home. Chen et al. [100] designed an ontology-based activity recognition technique in the context of assisted living within smart home environments. Evesti et al. [101] proposed an information security ontology and implemented it into security measures of the password. It could be used for adaptive user authentication where the system is able to dynamically modify user authentication depending on the monitoring authentication related measures [102]. Borgo et al. [103] developed an ontology for collaborative robots in the manufacturing domain, enabling the reconfigurable transportation system to adapt control loops based on context knowledge. Seydoux et al. [104] proposed the IoT-O ontology aiming at tackling interoperability issues in the smart home scenario. Khan et al. [105] proposed a context-based security guideline ontology for describing

vulnerabilities in smart homes. Since their proposed ontology could not describe services and relevant context, it is not suitable for the resource description of systems backed by the microservice architecture.

Korzun et al. [106] proposed the Smart-M3 platform with three key properties, namely multi-device, multi-vendor, and multi-domain described by their previous work [107]. The proposed platform has two types of components: the knowledge processors (KPs) representing information producers and consumers such as devices and users, and semantic information brokers (SIBs) to handle interactions among knowledge processors. The Smart Space Access Protocol was implemented to handle communications between SIB and KPs for interoperability. Based on the Smart-M3 platform, many use cases, including smart home, smart city, and healthcare systems, were developed [108–110]. While we solve some similar problems in Chapter 6, including security and ontology-based reasoning, the underlying nature and method are different. We follow the notion of the microservice-based and event-driven architecture where a central message broker is not required, and the interoperability can be well maintained through the replaceable request model. Also, the existing solutions did not give a solution to autonomic security management. While several works focusing on IoT systems with the microservice architecture have emerged [97, 111, 112], they focused more on the resource description at a large granularity level, such as smart city, and did not consider the security management either.

The existent ontologies fail to describe services, devices, policies, events, and contexts in a security-enhanced smart space backed by IoT systems with the microservice architecture. Besides, an effective solution is required to integrate with the ontology and manage the security of the smart space while minimizing manual work. We fill both gaps by proposing and implementing the Secure Smart Space Ontology and an autonomic security manager in this thesis (Chapter 6).

Chapter 3

Improved SLA Adherence: Adaptive Serverless Function Placement

In this chapter, we formulate the serverless function placement problem and address Objective 1 by proposing and evaluating a machine learning-based algorithm named Smart Spread, which can adaptively select the location for function containers to place, mitigate resource contention, improve the throughput of serverless functions, and thus enhance SLA adherence without incurring significant operational overhead.

3.1 Function Placement Problem

As shown in Figure 1.2, when FaaS scheduling a new function container, the container management platform has to select one VM to deploy the container. Function containers have various types of workloads demanding different resources. Different VMs also have a different level of resource utilization in terms of CPU, memory, network, and disk, which is expected to have a different impact on the performance of functions. If there are too many function containers with the same type of workload deployed on the same node, the performance in terms of response time and throughput would inevitably decrease due to resource contention, compromising the ability of FaaS platforms to deliver on their performance commitments. Therefore, the function placement problem is to find an optimal function placement solution

without incurring prohibitively high complexity and cost, which can place the new function container optimally to try to avoid performance degradation due to resource contention, improve throughput, and thus enhance performance SLA adherence.

While the function placement solution of most public FaaS platforms is usually unknown to outsiders, there are works that examine function placement by conducting experiments. Wang et al. [31] examined how functions are scheduled on FaaS platforms by periodically executing 40 measurement functions. Lloyd et al. [113] studied container placement by performing scaling tests. AWS Lambda appears to treat container placement as a bin packing problem and tries to place a new function container on an existing active VM to maximize memory utilization of VMs. For Google Cloud Functions, the placement strategy is unknown. Azure Functions seems to try to avoid co-locating containers of the same function on the same VM, suggesting a spread algorithm may be adopted.

The Docker Swarm scheduler supports three basic scheduling strategies, namely *spread*, *binpack*, and *random*. The default strategy is *spread*, in which the scheduler assigns the container to the Docker Swarm node with the most available resources. The *binpack* strategy minimizes the number of nodes in use and maximizes resource utilization by placing containers on one node until its resources is depleted. As the name suggests, the *random* strategy randomly selects an available node to place the container. Besides the aforementioned built-in strategies, the Swarm manager also supports filters to tell the scheduler how to further narrow down the list of potential nodes for creating and running a container. The constraints of the filter include OS and health of the node, task slot, the number of running/stopped containers on the node, affinity rules, dependencies, and port numbers. Most open-source and Kubernetes-based FaaS platforms leverage the *spread* strategy for container placement by default while providing advanced filters and constraint-based scheduling strategies, which are more or less similar to strategies and filters in Docker Swarm.

In summary, all these major container orchestration solutions treat containers/-

workloads in a black-box manner, and most of them leverage a rigid *spread* or *binpack* strategy as their best effort to maintain the performance SLA. The insufficient performance isolation stemmed from the black-box container scheduling technique can severely degrade the performance of functions and serverless applications hosted on FaaS platforms. In this work, we take the characteristics of function containers and performance metrics of VMs into account and propose the Smart Spread algorithm to mitigate resource contention and improve the throughput of serverless functions. The proposed algorithm can optimally select the VM leading to the best predicted performance to place the function, benefiting both FaaS users and service providers.

3.2 System Overview

The idea behind the Smart Spread algorithm is to leverage a predictive performance model powered by ANN implemented on TensorFlow in order the select a VM to place the function container leading to the best performance for FaaS users. Figure 3.1 illustrates the system overview of the proposed Smart Spread algorithm.

As shown in Figure 3.1, the system consists of three phases, namely a workload profiling and data collection phase, an offline training phase, and a performance prediction phase. In the workload profiling and data collection phase, the function container is first deployed on a dedicated VM without any workloads in order to obtain its baseline throughput and workload profile. Then, the function container is deployed on a VM with random workloads running on it. The system continuously collects the resource utilization metrics of the VM and throughput of the function and calculates the normalized throughput of the function (illustrated by blue lines). The system will later split the collected data sets into training and test sets and use them for training the predictive performance model in the offline training phase (depicted by red lines). In the performance prediction phase, the algorithm will take the workload profile of the function to place and the VM resource utilization metrics of all available VMs in the VM pool as inputs and then leverage the performance model to predict the

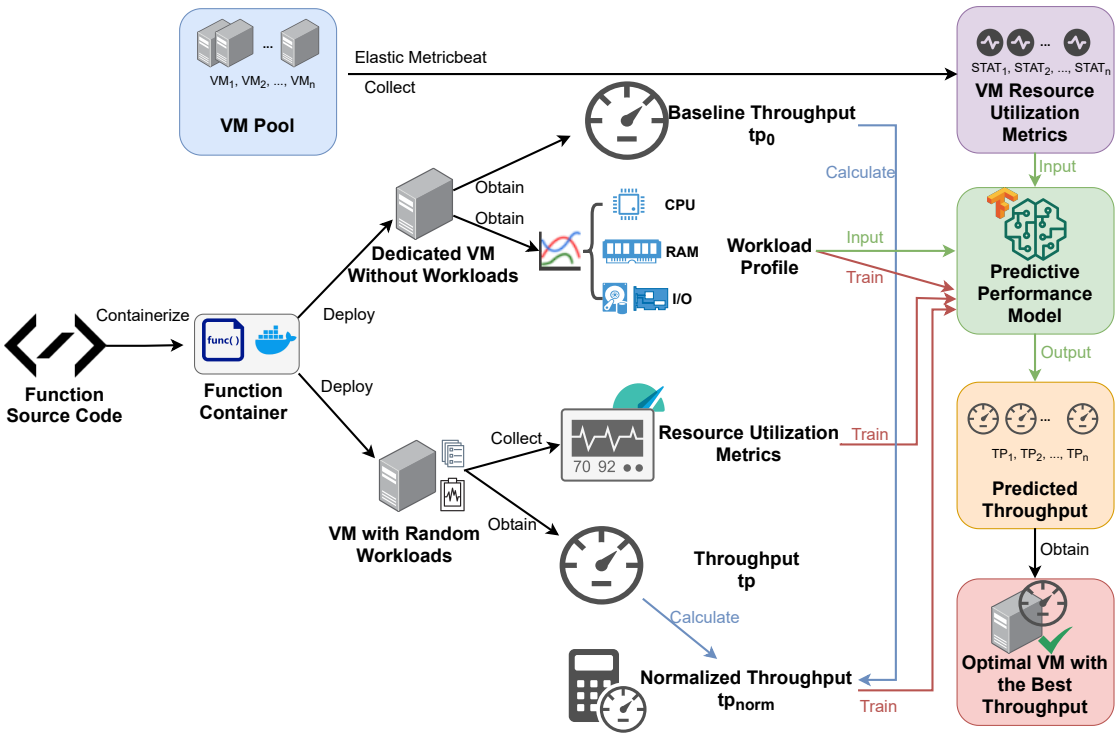


Figure 3.1: System Overview of the proposed Smart Spread algorithm.

normalized throughput of the function on each VM (demonstrated by green lines). The VM yielding the best predicted normalized throughput will be selected by the algorithm to place the function container. Section 3.3, Section 3.4, and Section 3.5 describe three phases in detail.

As building and evaluating such a system require complete control over the FaaS platform for container placement and performance measurement, we develop a serverless computing platform from scratch for evaluating the proposed algorithm. The platform is backed by an OpenStack-based IaaS platform with Docker and Metricbeat preinstalled on each hosted VM. We develop a Flask-based RESTful API and deploy it on every node in order to let the system interact with each node. Through the API, the system can deploy a container with CPU and memory limits on a VM, remove existing containers, and get statistics of containers. RabbitMQ is used as the distributed task queue in the system. The system leverages Elasticsearch and

Metricbeat to collect the performance metrics of VMs. The performance metrics are retrieved through Elasticsearch API by the system. For evaluation and comparison, three built-in placement strategies mentioned in Section 3.1 together with the proposed adaptive function placement algorithm are implemented in the platform.

3.3 Workload Profiling and Data Collection

The workload profiling and data collection phase is responsible for capturing specific characteristics of the workload in the function container and measuring the impacts of random workloads on the performance of the function. When profiling the function, the function container is deployed on a dedicated VM to eliminate the performance degradation incurred by co-located containers. The profiler in the system sends concurrent requests to the function and measures its throughput in RPM (requests per minute). The measured throughput will be used as the baseline throughput for calculating the normalized throughput when collecting performance data under random workloads. At the same time, based on Elasticsearch and Metricbeat, the system monitors and collects the resource utilization metrics of the dedicated VM as the workload profile of the function. We adopted the resource utilization metrics from the previous work presented by Lloyd et al. [114], which studied performance predictors for applications hosted on IaaS clouds. Table 3.1 lists the resource utilization metrics collected by the system.

To study the effect of resource contention caused by other workloads on the performance of functions, we need to collect performance data under random workloads after profiling the workload. The system first deploys a random number of containers that each encapsulates a random workload on a VM and measures the resource utilization metrics of the VM specified in Table 3.1 under such random workloads. Then, the system deploys the function container, sends concurrent requests to the function, and measures its throughput in RPM.

As serverless functions hosted on FaaS platforms have diverse workloads and us-

Table 3.1: Resource utilization metrics.

Statistic	**Description**	**Unit**
CPU time	CPU time	ms
cpu usr	CPU time in user mode	ms
cpu krn	CPU time in kernel mode	ms
cpu idle	CPU idle time	ms
contextsw	Number of context switches	count
cpu io wait	CPU time waiting for I/O completion	ms
cpu sint time	CPU time serving soft interrupts	ms
dsr	Disk sector reads	count
dsreads	Number of completed disk reads	count
readtime	Time spent reading from disk	ms
dsw	Disk sector writes	count
dswrites	Number of completed disk writes	count
writetime	Time spent writing to disk	ms
nbs	Network bytes sent	count
nbr	Network bytes received	count
loadavg	Avg # of processes in last min	count
mem used pct	The % of memory currently used	%

ages, absolute throughput is not a reliable performance indicator. For cloud service providers that handle various workloads, a good performance indicator should be able to handle any type of workload regardless of its type and complexity. Therefore, we need to normalize the throughput measured in this step to eliminate the influence of different types and scales of the workload. Also, normalization can reduce the complexity of data and the difficulty of the problem to be modeled. The normalized throughput tp_{norm} is defined as Equation (3.1).

$$tp_{norm} = \frac{tp}{tp_0} \tag{3.1}$$

where tp_0 is the baseline throughput collected in the workload profiling phase and tp is the throughput measured under random workloads.

Under random workloads, we collect a data set containing 183 data points. Each data point consists of resource utilization metrics specified in Table 3.1 and normalized throughput calculated by Equation (3.1).

3.4 Performance Model Training

With the workload profile and data set collected in the workload profiling and data collection phase, we create a data set for training the predictive performance model, which consists of 183 data points. The data set is divided into a training set with 128 data points and a test set with 55 data points. During the data preprocessing step, the data points are standardized by removing the mean and scaling to unit variance. The predictors of the machine learning-based performance model are resource utilization metrics collected before deploying the new function container and the workload profile of the function. The model predicts the normalized throughput of the function for the given workload profile of a function and resource utilization metrics of a VM.

Due to varied characteristics of workloads, the performance (i.e., throughput and response time) of the serverless function changes predominantly in a non-linear fashion. We use ANN to train the performance model in the offline training phase because of its high agility, fast prediction speed, generality, and flexibility, especially when fitting non-linear functions.

The model is configured as a two-layer neural network. The first and the second layers have ten neurons and five neurons, respectively, with the ReLU activation function. The output layer uses the identity activation function. Mean square error (MSE) is used as the loss function. The optimizer is based on stochastic gradient descent (SGD) with a batch size of 10 and 1,000 epochs of training. Table 3.2 lists the statistics of the trained model. Figure 3.2 demonstrates the performance of the trained model on the test set.

Table 3.2: Statistics of the trained model.

Statistic	**Value**
R^2	0.9309
$RMSE$	0.0584
$Correlation$	0.9705

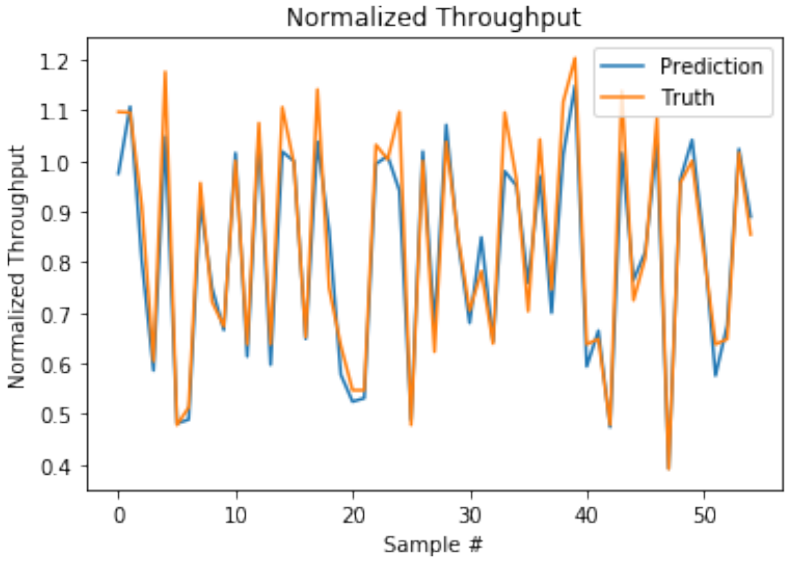


Figure 3.2: The performance of the trained model for predicting the normalized throughput validated by the test set.

3.5 Performance Prediction

The performance prediction phase is implemented by integrating the data collection techniques in the workload profiling and data collection phase and the predictive performance model obtained in the offline training phase. When the system receives a request to trigger a function, the request enters the task queue as a message. When processing the request, the system consumes the message in the task queue and optimally places the new function container on a VM by leveraging the performance model trained in the offline training phase.

When consuming the message, the system first calls the implemented API to re-

Table 3.3: Configuration of the VMs used in the experimental evaluation.

Property	**Value**
vCPU	2
RAM	4 GB
HDD	40 GB
Network	1 Gbps
OS	Ubuntu 18.04 (Cloud Image)

trieve the performance metrics specified in Table 3.1, which are monitored and collected by Metricbeat from all available VMs in the VM pool. Then, after preprocessing the retrieved metrics, the system sends the metrics and the workload profile to the predictive performance model. The model predicts the normalized throughput of the function on each VM and returns predicted values to the system. The system selects the VM with the greatest predicted normalized throughput to place the new function container through the developed API and returns the result to the client. For autoscaling, a threshold-based naive algorithm is leveraged. For each function, the system measures the response time of the request and adds the number of containers if the measured response time is beyond the predefined performance SLA.

3.6 Experimental Evaluation

3.6.1 Experimental Design

We deploy the proposed FaaS platform together with the Smart Spread algorithm on a cluster with four VMs as worker nodes backed by Cybera [115], an OpenStack-based IaaS platform. Table 3.3 summarizes the configuration of the VMs used in the experimental evaluation.

For serverless functions for evaluation, we use three different types of workload, namely CPU intensive, disk I/O intensive, and memory intensive, to simulate possible

Table 3.4: Benchmark programs encapsulated in the container for evaluation.

Application Type	**Benchmark**	**Configuration**
CPU Intensive	Sysbench CPU [117]	max-requests=2500 cpu-max-prime=1000
Disk I/O Intensive	Fileio [117]	max-requests=200 file-test-mode=rndrw
Memory Intensive	OLTP [117, 118]	table-size=10000 table-count=3 max-requests=10

workloads that a FaaS platform may process. Each type of workload is encapsulated into a Docker container, in which a web server listens for incoming requests and executes a benchmark program with a certain type of workload upon receiving a request. Table 3.4 describes the benchmark programs and corresponding configurations used in the evaluation process. For the CPU and disk I/O intensive workloads, we use *debian:latest* as the base image. The base image of the memory intensive workload is *MySQL:5.7*. All base images are obtained from the official repository on Docker Hub [116].

When deploying the container, each container is configured with a memory limit of 512 MB, a CPU limit that is 50% of a CPU core, and a disk I/O limit of 7.15 MB/s, which is about half of the maximum throughput of the hard disk drive attached to the VM. Therefore, a simple calculation shows that at most 7 containers can be deployed on a VM simultaneously. In order to avoid any potential crash due to limited computing resources, we limit the number of co-located containers on a VM to 6, thus limiting the total number of containers deployable in the proposed system to 24.

We test the performance of the hosted function by sending requests to trigger a function with concurrency levels changing with respect to time. A script is developed to simulate that each user in a group makes a request, waits for the response, and makes another request as soon as the response is received for the previous request. Figure 3.3 illustrates the concurrency level of test requests sending to the platform

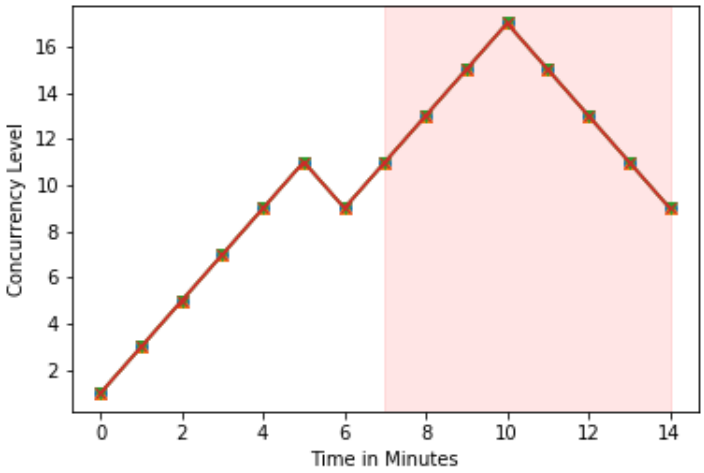


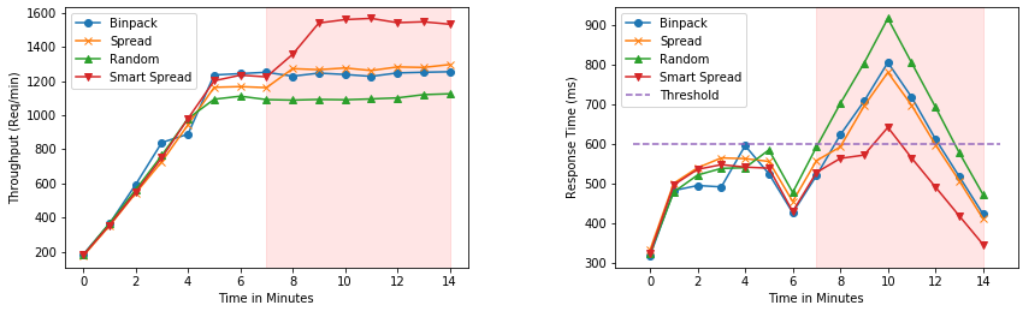
Figure 3.3: Concurrency levels when sending the test requests.

with respect to time. For example, the concurrency level is 1 at time 0, indicating only one client sends requests to the platform. At time 10, the concurrency level is 17, which means there are 17 concurrent clients making requests. Besides, to eliminate the effect of network latency, all requests are made from a VM inside the internal network with negligible latency to the platform. To get the steady-state throughput and response time, we assume a warmup period of 30 seconds during which the results are not recorded and used for evaluation. We perform tests using four function placement algorithms, namely *spread*, *binpack*, *random*, and *Smart Spread*, and obtain and compare the performance of functions in terms of their throughput and response time to evaluate the proposed *Smart Spread* algorithm.

3.6.2 Experimental Result

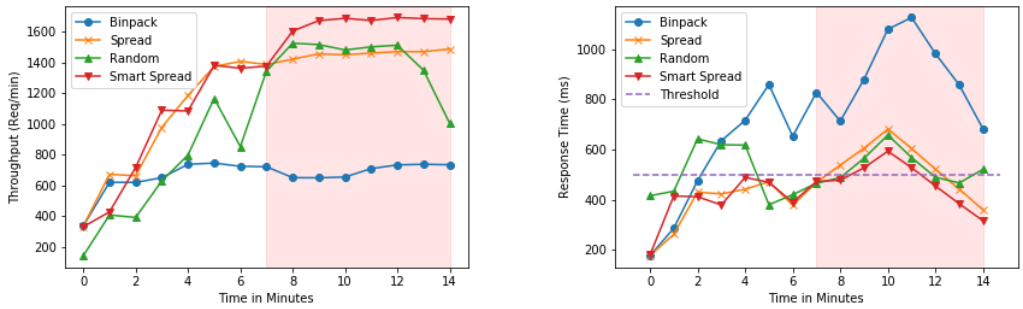
Figure 3.4, Figure 3.5, and Figure 3.6 illustrate the comparison of four function placement algorithms in terms of throughput and response time of functions with CPU, disk, and memory intensive workloads, respectively. To simulate the performance of the proposed function placement algorithm when scheduling a serverless application composed of multiple functions, we aggregate the results of three functions by

summing up the throughput and response time of three functions presented in Figure 3.4, Figure 3.5, and Figure 3.6. Figure 3.7 shows the aggregated results in terms of throughput, response time, and the number of containers. The shaded areas in figures 3.4 to 3.7 represent the performance of functions under heavy load.



(a) Throughput (b) Response time

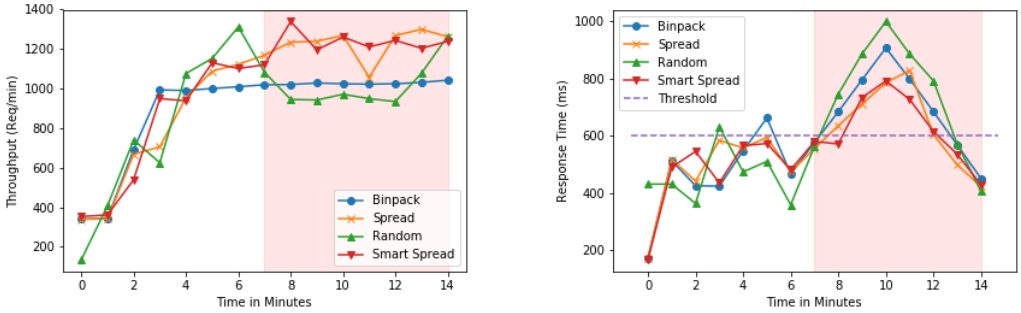
Figure 3.4: Comparison of four function placement algorithms in terms of throughput and response time of the function with a CPU intensive workload.



(a) Throughput (b) Response time

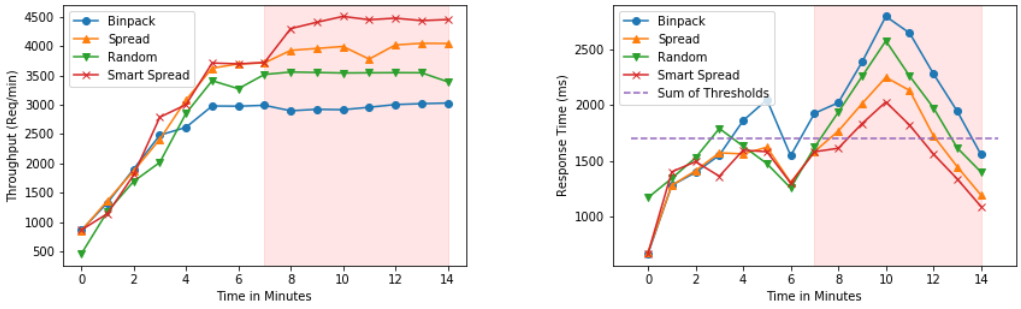
Figure 3.5: Comparison of four function placement algorithms in terms of throughput and response time of the function with a disk I/O intensive workload.

As shown in figures 3.4 to 3.7, the average aggregated throughput achieved by *binpack*, *random*, *spread*, and *smart spread* under heavy load (shaded area) is 2996, 3523, 3934, and 4341, respectively. The proposed adaptive function placement algorithm could improve the throughput by 10.35% - 44.89%. When functions are executed with low concurrency levels, since there are sufficient resources, the throughput of functions increases linearly without considerable differences, regardless of placement

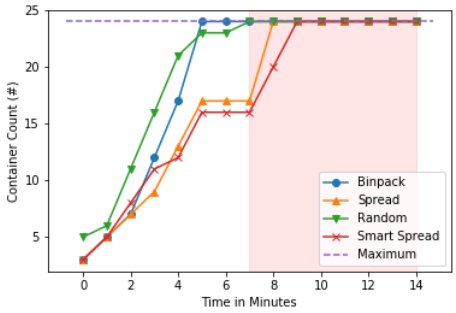


(a) Throughput (b) Response time

Figure 3.6: Comparison of four function placement algorithms in terms of throughput and response time of the function with a memory intensive workload.



(a) Throughput (b) Response time



(c) Container Count

Figure 3.7: Comparison of four function placement algorithms in terms of aggregated throughput, response time, and container count.

algorithms. When testing under heavy load (shaded areas), the proposed Smart Spread algorithm generally produces the highest throughput while using the smallest number of containers among four strategies, suggesting the proposed algorithm could improve performance SLA adherence. As shown in Figure 3.7a, the *binpack* strategy

achieves the worst throughput among three placement algorithms that treat the function in a black-box manner, as it sacrifices performance for lower costs. Conversely, since the *spread* strategy sacrifices cost for better performance, it yields the highest throughput among the three strategies. The performance of the *random* algorithm is better than the *binpack* strategy but worse than the *spread* strategy.

We perform an analysis consisting of 100 iterations of optimal VM selection and obtain the processing time of the system using the proposed algorithm to identify the best VM candidate. Under the experiment configuration with four VMs, the average processing time is 10.16 ms, which is negligible compared to the lifespan of the function container and the billing granularity of FaaS platforms. The result indicates that the proposed function placement algorithm does not incur significant overhead.

3.7 Summary

In this chapter, we addressed Objective 1 by proposing and evaluating an adaptive serverless function placement algorithm named Smart Spread. We used ANN to train the predictive performance model, which can accurately predict the normalized throughput of a function based on the workload profile of the function and performance metrics of VMs. To evaluate the algorithm, we developed a serverless computing platform from scratch and conducted extensive experiments using various workloads. Compared to placement strategies based on black-box techniques, the proposed algorithm can mitigate resource contention, considerably increase the throughput of the serverless function, and thus help FaaS providers improve SLA adherence. The evaluation also validated the low overhead incurred by the proposed algorithm.

Chapter 4

Performance and Cost Modeling for Serverless Applications

As discussed in Section 1.6, performance and cost modeling is essential when designing and implementing distributed systems in order to set and manage performance SLAs and control costs. Also, it is a best practice to test the performance and estimate the cost of the serverless functions and applications before deploying them [41, 119]. However, there is a lack of performance and cost models for applications based on serverless microservices, which has become one of the challenges in serverless computing [11, 41, 42]. In this chapter, we solve this gap by building performance and cost models that can obtain the end-to-end response time and cost of a serverless application when orchestration and configuration of the application are given, i.e., addressing Objective 2. We first construct the serverless workflow of a serverless application by giving the definition of the serverless workflow and four types of structures in the serverless workflow. Then, we propose two analytical models to predict the performance and cost of the serverless application. By real experiments on AWS, the accuracy of the two proposed models is evaluated.

4.1 Definition of the Serverless Workflow

The serverless workflow of a serverless application G_s is defined as a weighted directed graph.

$$G_s = (V, E, P, D, RT, RTTP, M, NI, C) \tag{4.1}$$

where

- V is a finite set of $|V|$ vertices $\{f_1, f_2, ..., f_n\}$, such that $n = |V|$, representing FaaS functions, integrated cloud services, or structural vertices;
- $E \subseteq V \times V$ is a finite set of directed edges. The directed edge from $f_u \in V$ to $f_v \in V$, denoted as $e_i = (f_u, f_v)$, represents the interaction between vertex f_u and f_v defined by the business logic;
- $P : V \times V \rightarrow [0, 1]$ is a transition probability function. $P(f_u, f_v)$ identifies the probability of invoking f_v after finishing the execution of f_u. A transition probability of 0 represents the corresponding edge does not exist;
- $D : V \times V \rightarrow [0, +\infty)$ is a delay function. $D(f_u, f_v)$ identifies the delay from f_u to f_v incurred by the interaction/coordination method;
- $RT : V \rightarrow [0, +\infty)$ is the response time of a function. $RT(f_u)$ is the response time of function f_u;
- $RTTP$: $V \rightarrow \wp([0, +\infty) \times [0, 1])$ is a function representing all possible values of the response time and corresponding probability of interim B-nodes, which are defined in Section 4.3.1 to process branches. $RTTP(f) \triangleq \varnothing$ for any f which is not a B-node;
- $M : V \rightarrow \mathbb{N}$ is a memory function. $M(f_u)$ is the size of the allocated memory of function f_u;
- $NI : V \rightarrow [0, +\infty)$ is a function representing the average number of invocations of each function in V, per execution of the serverless workflow G_s;

– $C : V \to [0, +\infty)$ is a cost function. $C(f_u)$ is the cost per invocation of function f_u;

Besides FaaS functions, vertices can also represent other cloud services, such as a MapReduce service or a database operation service. As those services are similar to functions in terms of execution, we collectively call them functions for brevity. V can also contain several non-functional structural vertices that facilitate developing the workflow and do not incur any delay and cost, such as start node and end node. Typically, the execution of the serverless workflow starts with a particular function as a trigger, and then the following functions will be invoked to complete the business logic [120]. Therefore, we include a start node f_{str} and an end node f_{end} in V, defining the entry point and the endpoint of the workflow, respectively. Figure 1.4 shows an example of a serverless workflow for image classification. Based on the definition of the serverless workflow, we define the following notations:

1. A simple path in the serverless workflow is a finite sequence of distinct vertices and edges $s = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$ such that (i) $f_i \in V$ for all integers $1 \leq i \leq n$, (ii) $e_i \in E$ for all integers $1 \leq i \leq n-1$, and (iii) $e_i = (f_i, f_{i+1})$ for all integers $1 \leq i \leq n-1$.

2. The transition probability of the simple path $s = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$ is defined as Equation (4.2).

$$TPP(s) = \prod_{i=1}^{n-1} P(f_i, f_{i+1}) \tag{4.2}$$

3. The delay (response time) of a simple path $s = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$, denoted as $DLY(s)$, is defined as Equation (4.3), namely the sum of the response time of functions and the delay incurred by edges in this path. In particular, we use $DLY^{-}(s)$ to denote the delay of a simple path without considering the response time of the first and last vertices in the simple path, defined as Equation (4.4).

$$DLY(s) = \sum_{i=1}^{n} RT(f_i) + \sum_{i=1}^{n-1} D(f_i, f_{i+1}) \tag{4.3}$$

$$DLY^{-}(s) = \sum_{i=2}^{n-1} RT(f_i) + \sum_{i=1}^{n-1} D(f_i, f_{i+1}) \tag{4.4}$$

4. $ASP(f_u, f_v)$ denotes all simple paths between vertex f_u and f_v, which is the set of all possible simple paths in the workflow graph, starting from f_u and ending at f_v, with f_u and f_v included.

5. The shortest path length between two vertices f_u and f_v, denoted as $SPL(f_u, f_v)$, is specified as the length of the shortest simple path from f_u to f_v. The length of a simple path is the number of edges in it.

6. $SUB(f_u, f_v)$ denotes the subgraph between $f_u \in V$ and $f_v \in V$, which is derived from G_s by removing all vertices and edges not in any paths in $ASP(f_u, f_v)$.

7. $out(f_u)$ denotes the set of all edges starting from vertex f_u, defined as the following.

$$out(f_u) = \{e \in E : e = (f_u, f) \text{ for some } f \in V\}$$

For convenience, Table 4.1 includes the definitions of notations and parameters used in the performance and cost models.

4.2 Structures in the Serverless Workflow

In this section, we define four types of structures in the serverless workflow, namely parallel, branch, cycle, and self-loop, as shown in Figure 4.1.

4.2.1 Parallel

Let us consider all simple paths between vertices $f_u \in V$ and $f_v \in V$. If there is more than one simple path whose transition probability is 1, we define the subgraph composed of all simple paths with the transition probability of 1 as a ***parallel structure***.

Table 4.1: Definition of notations used in the performance and cost models.

Notation	**Definition**
G_s	the serverless workflow (eq. (4.1)) $G_s = (V, E, P, D, RT, RTTP, M, NI, C)$
f	a FaaS function, cloud service, or structural vertex
$e = (f_u, f_v)$	a directed edge from f_u to f_v
s	a simple path
$TPP(s)$	transition probability of a simple path s
$DLY(s)$	delay of a simple path s
$DLY^{-}(s)$	delay of a simple path s without considering the response time of first and last vertices in it
$ASP(f_u, f_v)$	the set of all simple paths between f_u and f_v
$SPL(f_u, f_v)$	the shortest length of simple paths between f_u and f_v
$SUB(f_u, f_v)$	the subgraph between f_u and f_v
$out(f_u)$	the set of all edges starting from f_u
$ERT(G)$	the end-to-end response time of the workflow G
$\wp$	power set
$\uplus$	disjoint union
$\Downarrow$	restriction of a function
$\mapsto$	map an element in the function domain
$\|A\|$	the cardinal number of a set
G_p, G_b, G_c, G_l	parallel, branch, cycle, self-loop structures, respectively
SP_p	the set of the simple path whose transition probability is 1
SP_b	the set of the simple path whose transition probability is not 1
G_{pr}	probabilistic DAG used in the performance model
G_{dl}	de-looped graph (DAG) used in the cost model
G_{perf}	the graph used in the performance model (eq. (4.5))
G_{cost}	the graph used in the cost model (eq. (4.11))
$SRT(G)$	the RT of a parallel/branch/cycle/self-loop
$EI(G)$	the expected number of iterations of a cycle/self-loop
$DI(G)$	the delay incurred by iterations of a cycle/self-loop
f_B	a B-node used in procedures of processing branches
f_P	a P-node used in procedures of processing parallels
PGC	price per GB-second of FaaS functions
PI	price per invocation of FaaS functions

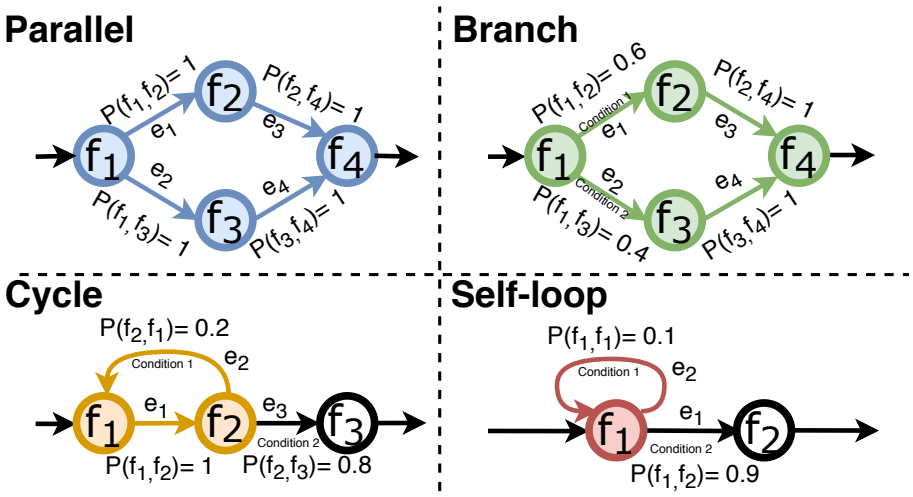


Figure 4.1: Four types of structures in the serverless workflow.

Namely, we define the subgraph $G_p = (V_p, E_p)$ as a parallel structure, such that $|SP_p| > 1$, where

$$V_p = \{f \in V : \text{f is in s for some s} \in SP_p\}$$

$$E_p = \{e \in E : e \text{ is in } s \text{ for some s} \in SP_p\}$$

$$SP_p = \{s \in ASP\,(f_u, f_v) : TPP\,(s) = 1\}$$

As the parallel shown in Figure 4.1, the vertices and edges of the parallel structure are depicted in blue color. There are two simple paths with the transition probability of 1 between f_1 and f_4, namely $f_1e_1f_2e_3f_4$ and $f_1e_2f_3e_4f_4$. Functions f_2 and f_3 are processed in parallel, indicating that f_1 leverages interactions to invoke f_2 and f_3 at the same time after finishing its execution, and f_4 starts execution only after both f_2 and f_3 are completed.

4.2.2 Branch

Let us consider all simple paths between vertices $f_u \in V$ and $f_v \in V$. If there is more than one simple path whose transition probability is less than 1, but can sum up to 1 in total, we define the subgraph composed of those simple paths with the transition probability not equal to 1 as a ***branch structure***.

Namely, we define the subgraph $G_b = (V_b, E_b)$ as a branch structure, such that $|SP_b| > 1$, where

$$V_b = \{f \in V : \text{f is in s for some s} \in SP_b\}$$

$$E_b = \{e \in E : e \text{ is in } s \text{ for some s} \in SP_b\}$$

$$SP_b = \{s \in ASP\,(f_u, f_v) : TPP\,(s) \neq 1\}$$

$$\sum_{s \in SP_b} TPP\,(s) = 1$$

As the branch shown in Figure 4.1, the vertices and edges of the parallel structure are depicted in green color. There are two simple paths between f_1 and f_4 whose transition probabilities are not equal to 1, but can add up to 1. Hence, vertices and edges in simple paths $f_1e_1f_2e_3f_4$ and $f_1e_2f_3e_4f_4$ form a branch structure. After completing f_1, the workflow continues with only one path in the branch, depending on the satisfied condition.

4.2.3 Cycle

Considering all simple paths between vertices $f_u \in V$ and $f_v \in V$, we define the subgraph $G_c = (V_c, E_c)$ as a ***cycle structure*** between vertices f_u and f_v, where

$$V_c = \{f \in V : \text{f is in s for some s} \in ASP\,(f_u, f_v)\}$$

$$E_c = \{e \in E : \text{e is in s for some s} \in ASP\,(f_u, f_v)\} \uplus \{(f_v, f_u)\}$$

such that

- $0 < P(f_v, f_u) < 1$, invoking f_u again after completing f_v is a possible event;
- $\sum_{s \in ASP(f_u, f_v)} TPP\,(s) = 1$, transition probabilities of all simple paths between f_u and f_v sum up to 1;
- $SUB(f_u, f_v)$ is a DAG, the subgraph between f_u and f_v does not have any loops and cycles;

- $SPL\left(f_{str}, f_u\right) < SPL(f_{str}, f_v)$, compared to f_v, f_u is closer to the entry point;
- $\sum_{(f_v, f_i) \in out(f_v)} P\left(f_v, f_i\right) = 1$, the transition probabilities of all edges starting from f_v add up to 1.

As the cycle shown in Figure 4.1, vertices f_1 and f_2 and edges e_1 and e_2 form a cycle, depicted in orange color. After completing f_2, depending on the satisfied condition, the workflow will either invoke f_3, or enter the cycle by invoking f_1.

4.2.4 Self-loop

A self-loop is a special case of a cycle with only one vertex and one edge. Considering a function f_u connected by an edge (f_u, f_u) to itself, we define the subgraph $G_l = (V_l, E_l) = (\{f_u\}, \{(f_u, f_u)\})$ as a ***self-loop structure***, such that

- $0 < P(f_u, f_u) < 1$, invoking f_u again after completing f_u is a possible event;
- $\sum_{(f_u, f_i) \in out(f_v)} P\left(f_u, f_i\right) = 1$, the transition probabilities of all edges starting from f_u add up to 1.

As the self-loop depicted in red color shown in Figure 4.1, after accomplishing f_1, the workflow will either invoke f_2, or enter the self-loop by invoking f_1 again.

4.3 Performance Modeling

We propose a performance model to get the end-to-end response time of the serverless workflow. As M, NI, and C defined in G_s are relevant to cost instead of performance, we do not consider them in the performance model for brevity. Specifically, we only consider the following graph with part of elements in G_s, defined as

$$G_{perf} = (V, E, P, D, RT, RTTP) \tag{4.5}$$

With different methods for different structures, the performance model trims the graph of G_{perf} by removing, adding, and modifying vertices, edges and elements, and

converts G_{perf} into a probabilistic DAG, denoted as G_{pr}. We define the probabilistic DAG as

$$G_{pr} = (V, E, P, D, RT, RTTP) \tag{4.6}$$

such that

- G_{pr} is a DAG without any cycles and loops;
- $\sum_{s \in ASP(f_{str}, f_{end})} TPP(s) = 1$, the transition probabilities of all simple paths between the start node and the end node in G_{pr} can sum up to 1.

The end-to-end response time of the serverless workflow G_{perf}, denoted as $ERT(G_{perf})$, is defined as Equation (4.7).

$$ERT(G_{perf}) = \sum_{s \in ASP(f_{str}, f_{end})} TPP(s)\, DLY(s) \tag{4.7}$$

where $ASP(f_{str}, f_{end})$ is the set of all simple paths between the start node and end node in G_{pr}, which is the probabilistic DAG converted from G_{perf} by the proposed performance model.

To convert a serverless workflow into a probabilistic DAG, the model needs to remove cycles and self-loops from the workflow and trim parallel paths. In the following subsections, we describe how the performance model processes four types of structures in the serverless workflow and converts the workflow graph into a probabilistic DAG.

Algorithm 1 gives the pseudo-code of the performance model. Figure 4.2 illustrates the step-by-step changes of the workflow graph when the performance model works on the serverless workflow shown in Figure 1.4. In Figure 4.2, the interim B-node and P-node (B_1, B_2, P_1, and P_2) are depicted in bold. The value of $RTTP$ of the B-node retains the response time and probability of each path in the branch. The number on each edge represents the transition probability, and the transition delay is considered as zero for brevity.

Algorithm 1: Performance modeling algorithm

Input: a serverless workflow $G_s = (V_s, E_s)$

Output: the average end-to-end response time of the serverless application

$G' \leftarrow G_s$;
while G' *is not a probabilistic DAG (using eq.* (4.6)*)* **do**
    loop_list $\leftarrow$ find_self_loops(G'); ▷ `section 4.2.4`
    for *each self-loop* G_i *in loop_list* **do**
        Process self-loop G_i; ▷ `Section 4.3.4`
    end
    cycle_list $\leftarrow$ find_cycles(G'); ▷ `section 4.2.3`
    for *each cycle* G_i *in self_loop_list* **do**
        Process cycle G_i; ▷ `Section 4.3.3`
    end
    parallel_list $\leftarrow$ find_parallels(G'); ▷ `section 4.2.1`
    for *each parallel* G_i *in parallel_list* **do**
        Process parallel G_i; ▷ `Section 4.3.2`
    end
    branch_list $\leftarrow$ find_branches(G'); ▷ `section 4.2.2`
    for *each branch* $G_i = (V_i, E_i)$ *in branch_list* **do**
        Process branch G_i; ▷ `Section 4.3.1`
    end
end
$ERT \leftarrow \sum_{s \in ASP_{G'}(f_{str}, f_{end})} TPP(s) \cdot DLY(s)$; ▷ `eq. (4.7)`
return ERT

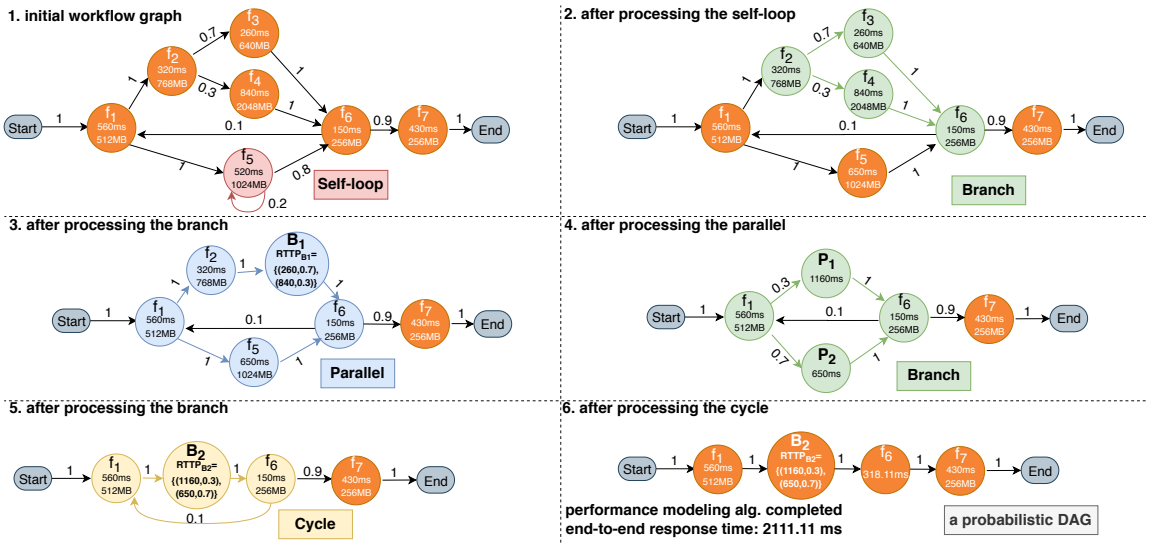


Figure 4.2: Steps of the performance modeling algorithm solving the end-to-end response time of the serverless application.

4.3.1 Process Branches

The workflow selects only one path in the branch depending on the satisfied condition. Each condition has a probability specified by the transition probability function. Hence, the probability of executing each path in the branch is the transition probability of the path. The main idea of processing the branch is to simplify the workflow by replacing branch paths with an interim B-node while retaining the information, including response time and probability of each branch path.

Consider a branch $G_b = (V_b, E_b)$ between vertices f_u and f_v, let SRT denote the response time of a structure, we can obtain the branch structure's expected value of the response time by Equation (4.8).

$$SRT\left(G_b\right) = \sum_{s \in ASP(f_u, f_v)} TPP\left(s\right) DLY\left(s\right) \tag{4.8}$$

After calculating the expected value of the delay, the performance model trims G_s by first removing all vertices in V_b except f_u and f_v from V, namely, we get a smaller set of vertices, denoted as V^*, defined as

$$V^* = V \backslash \left(V_b \backslash \left\{f_u, f_v\right\}\right)$$

Correspondingly, we remove all elements relevant to removed vertices by restricting functions as follows:

$$E^{*} = E \Downarrow V^{*}, P^{*} = P \Downarrow V^{*}, D^{*} = D \Downarrow V^{*}$$

$$RT^{*} = RT \Downarrow V^{*}, RTTP^{*} = RTTP \Downarrow V^{*}$$

Then, we add an interim vertex called "B-node", denoted as f_B, to V, and connect f_u, f_B, and f_v in sequence by adding two edges (f_u, f_B) and (f_B, f_v) to E. We extend functions in the graph tuple as follows:

$$V' = V^{*} \uplus \{f_B\}$$

$$E' = E^{*} \uplus \{(f_u, f_B), (f_B, f_v)\}$$

$$P' = P^{*}[(f_u, f_B) \mapsto 1, (f_B, f_v) \mapsto 1]$$

$$D = D^{*}[(f_u, f_B) \mapsto 0, (f_B, f_v) \mapsto 0]$$

$$RT' = RT^{*}[f_B \mapsto SRT(G_b) - RT(f_u) - RT(f_v)]$$

The above procedures simplify the workflow graph, as multiple vertices in the branch are replaced by an interim B-node, whose response time is the weighted average response time of branch paths. However, simply using the weighted average value would compromise the accuracy of the performance model. An example is when a branch is in parallel to other paths, as shown in Figure 4.2. For instance, let us consider a branch with two paths whose response time is 1290 ms and 1870 ms, with the transition probability of 0.7 and 0.3, respectively. There is another path in parallel to these two branches, whose response time is 1360 ms. In this case, the response time of these three paths combined should be either 1360 ms or 1870 ms, and the probability of each case is equal to 0.7 and 0.3, respectively. However, if the weighted average value is used as the branch's delay, the response time of these three paths would be a deterministic value of 1464 ms. Therefore, we retain the information, including response time and probability of each branch path using $RTTP$, and leverage Algorithm 2 to accurately model the response time in such cases.

Specifically, we define $RTTP$ as a function $RTTP : V \rightarrow \wp([0, +\infty) \times [0, 1])$, which represents all possible values of the response time and corresponding probability of all original paths replaced by interim B-nodes. When processing branches and adding B-nodes, we have

$$RTTP' = RTTP^* [f_B \mapsto \mathrm{B}]$$

where

$$\mathrm{B} = \left\{ \left(\mathrm{DLY}^{-} (\mathrm{s}), \mathrm{TPP} (\mathrm{s}) \right) | \forall \mathrm{s} \in \mathrm{ASP} (\mathrm{f_u}, \mathrm{f_v}) \right\}$$

After processing branches, the workflow graph is updated as

$$G_{perf} \leftarrow G' = (V', E', P', D', RT', RTTP')$$

4.3.2 Process Parallels

For a given parallel structure $G_p = (V_p, E_p)$ between vertices f_u and f_v, the workflow executes all parallel paths after completing f_u and invokes f_v only after finishing the executions of all paths. Hence, as shown in Equation (4.9), the response time of a parallel structure is the longest delay of parallel paths in it. Therefore, the main idea of processing the parallel is to retain the parallel path with the longest delay and prune other paths.

$$SRT (G_p) = \max \{ DLY (s) | \forall s \in ASP (f_u, f_v) \} \tag{4.9}$$

For a parallel structure without any interim B-nodes in any paths, the delay of each path is deterministic. We can directly use Equation (4.9) to get the delay of the parallel structure, which is also deterministic. However, if there are B-nodes in any paths, the delay of the parallel structure may have a probability distribution instead of being a fixed value. Since the response time of the B-node, recorded by $RTTP$, varies based on probabilities under different conditions, the delay of the path with B-nodes is subject to a probability distribution, making the delay of the parallel

structure probabilistic. In this case, the performance model leverages Algorithm 2 to get a set of tuples, denoted as $RTTP\_List$, which has all possible values of response time and corresponding probabilities of the parallel structure.

After calculating the delay using Equation (4.9) or deriving $RTTP\_List$, similar to the trimming step in Section 4.3.1, the performance model removes all vertices in V_p except f_u and f_v from V, and then restricts functions. We have

$$\begin{aligned}
&V^* = V \backslash (V_p \backslash \{f_u, f_v\}) \\
&E^* = E \Downarrow V^*, P^* = P \Downarrow V^*, D^* = D \Downarrow V^* \\
&RT^* = RT \Downarrow V^*, RTTP^* = RTTP \Downarrow V^*
\end{aligned}$$

Then, we add the interim vertex called "P-node" to V. If the parallel structure does not have any B-nodes, we only add one P-node, denoted as f_P and let

$$\begin{aligned}
&V' = V^* \uplus \{f_P\} \\
&E' = E^* \uplus \{(f_u, f_P), (f_P, f_v)\} \\
&P' = P^* [(f_u, f_P) \mapsto 1, (f_P, f_v) \mapsto 1] \\
&D' = D^* [(f_u, f_P) \mapsto 0, (f_P, f_v) \mapsto 0] \\
&RT' = RT^* [f_P \mapsto SRT(G_p) - RT(f_u) - RT(f_v)]
\end{aligned}$$

If V_p contains B-nodes, by using Algorithm 2, we have $RTTP\_List$ that has M tuples of the response time and probability as

$$RTTP\_List = \{(rt_1, pr_1), (rt_2, pr_2), ..., (rt_M, pr_M)\}$$

Then, M P-nodes will be added as follows:

$$\begin{aligned}
&V' = V^* \uplus_{i=1}^{M} \{f_{P_i}\} \\
&E' = E^* \uplus_{i=1}^{M} \{(f_u, f_{P_i}), (f_{P_i}, f_v)\} \\
&\text{for all } 1 \leq i \leq M\text{:} \\
&P' = P^* [(f_u, f_{p_i}) \mapsto pr_i, (f_{p_i}, f_v) \mapsto 1] \\
&D' = D^* [(f_u, f_{p_i}) \mapsto 0, (f_{p_i}, f_v) \mapsto 0] \\
&RT' = RT^* [f_{p_i} \mapsto rt_i]
\end{aligned}$$

After processing parallels, the workflow graph is updated as

$$G_{perf} \leftarrow G' = (V', E', P', D', RT', RTTP')$$

4.3.3 Process Cycles

For processing cycles, the main idea is to remove the cycle and add the delay incurred by cycle iterations to the last vertex's response time in the cycle. For a given cycle $G_c = (V_c, E_c)$ between vertices f_u and f_v, the expected value of the number of cycle G_c iterations, denoted as $EI(G_c)$, namely the average number of times the workflow enters the cycle after completing f_v, can be expressed as Equation (4.10).

$$\begin{aligned} EI(G_c) &= \sum_{n=1}^{\infty} [1 - P(f_v, f_u)] [P(f_v, f_u)]^{n-1} (n-1) \\ &= \frac{P(f_v, f_u)}{1 - P(f_v, f_u)} \end{aligned} \tag{4.10}$$

By multiplying the expected number of cycle iterations and the time required for each iteration, we can calculate the expected delay incurred by cycle iterations.

$$DI(G_c) = \left(\sum_{s \in ASP(f_u, f_v)} TPP(s) DLY(s) + D(f_v, f_u) \right) EI(G_c)$$

In terms of the response time of the cycle structure, the delay incurred by cycle iterations is equivalent to increasing the response time of f_v by the same amount of time. Therefore, the performance model first removes the edge (f_v, f_u) as

$$E' = E \backslash \{(f_v, f_u)\}$$

Then, the model modifies the transition probability, edge delay, and vertex response time as follows:

$$\begin{aligned} P^* &= P\left[e \mapsto \frac{P(e)}{1 - P(f_v, f_u)}\right], \text{ for all } e \in out(f_v) \\ D' &= D[(f_v, f_u) \mapsto 0] \\ P' &= P^*[(f_v, f_u) \mapsto 0] \\ RT' &= RT[f_v \mapsto RT(f_v) + DI(G_c)] \end{aligned}$$

Algorithm 2: Get $RTTP\_List$

Input: a parallel $G_p = (V_p, E_p)$ between $f_u \in V_p$ and $f_v \in V_p$ with n paths

Output: a set of tuples $RTTP\_List$ which has all possible values of response time and corresponding probability of G_P

for *path* $s_i \in ASP(f_u, f_v)$ **do**
  $RTTP\_List_i \leftarrow []$; ▷ `results list`
  if s_i *has* $m \geq 1$ *B-nodes* $\{f_{B_1}, f_{B_2}, ..., f_{B_m}\}$ **then**
    $\text{RT\_wo\_B} \leftarrow DLY^{-}(s_i) - \sum_{k=1}^{m} RT(f_{B_k})$
    ▷ `the delay of the path without B-nodes`
    $cmb \leftarrow RTTP(f_{B_1}) \times ... \times RTTP(f_{B_m})$;
    ▷ `all combinations of the RT and probability of all B-nodes by the Cartesian product`
    for *each combo* C_j *in* cmb **do**
      $RTC_j \leftarrow \sum_{(rt_k, tp_k) \in C_j} rt_k + RT\_wo\_B$;
      ▷ `a possible response time of` s_i
      $TPC_j \leftarrow \prod_{(rt_k, tp_k) \in C_j} tp_k$;
      ▷ `the corresponding probability`
      append (RTC_j, TPC_j) to $RTTP\_List_i$;
    end
  else ▷ `the path that does not have any B-nodes`
    $RTC \leftarrow DLY^{-}(s_i)$; ▷ `RT is deterministic`
    append $(RTC, 1)$ to $RTTP\_List_i$; ▷ `probability is 1`
  end
end
$rttp\_comb \leftarrow RTTP\_List_1 \times ... \times RTTP\_List_n$;
$RTTP\_List \leftarrow []$;
for *each combo* C_j *in* $rttp\_comb$ **do**
  $RTC_j \leftarrow \max\{rt_k | \forall (rt_k, tp_k) \in C_j\}$; ▷ `a possible RT of` G_p`, use maximum value due to parallelism`
  $TPC_j \leftarrow \prod_{(rt_k, tp_k) \in C_j} tp_k$; ▷ `the corresponding prob.`
  append (RTC_j, TPC_j) to $RTTP\_List$;
end
return $RTTP\_List$

After processing cycles, the workflow graph is updated as

$$G_{perf} \leftarrow G' = (V, E', P', D', RT', RTTP)$$

4.3.4 Process Self-loops

The procedure to process self-loops is similar to that used to process cycles. Given a self-loop $G_l = (\{f_u\}, \{(f_u, f_u)\})$, similarly using Equation (4.10), we can calculate the expected number of self-loop iterations as $EI(G_l) = \frac{P(f_u,f_u)}{1-P(f_u,f_u)}$, and the delay incurred by self-loop iterations as $DI(G_l) = EI(G_l)(RT(f_u) + D(f_u, f_u))$. Then, the performance model updates the graph as

$$G_{perf} \leftarrow G' = (V, E', P', D', RT', RTTP)$$

where

$$\begin{aligned}
&P^* = P\left[e \mapsto \frac{P(e)}{1 - P(f_u, f_u)}\right], \text{ for all } e \in out(f_u) \\
&D' = D\left[(f_u, f_u) \mapsto 0\right] \\
&P' = P^*\left[(f_u, f_u) \mapsto 0\right] \\
&E' = E \backslash \{(f_u, f_u)\} \\
&RT' = RT\left[f_u \mapsto RT(f_u) + DI(G_l)\right]
\end{aligned}$$

4.4 Cost Modeling

In this section, we introduce a cost model to get the average cost of the serverless workflow. Since FaaS platforms leverage a GB-second billing model depending on the allocated memory size, rounded-up function duration, and the number of invocations, we consider the following graph with part of elements in G_s, defined as

$$G_{\cos t} = (V, E, P, RT, M, NI, C) \tag{4.11}$$

All vertices representing FaaS functions have an amount of allocated memory, identified by M. The allocated memory of 0 represents such a vertex might be a

structural node without any cost, like start and end nodes, or other cloud services to which the GB-second billing model is not applicable. The rounded-up function duration can be directly calculated using the response time of each function defined by RT. For a FaaS function $f_u \in V$, its average cost per application execution can be calculated as Equation (4.12), where PGS is the price per GB-second and PPI is the price per invocation (the cost of handling the invocation request). For vertices representing cloud services with other pricing models, the model obtains their cost from users' input.

$$C\left(f_u\right) = NI\left(f_u\right)\left(\left\lceil \frac{RT\left(f_u\right)}{100} \right\rceil \cdot M\left(f_u\right) \cdot PGS + PPI\right) \tag{4.12}$$

Each vertex $f_u \in V$ has an average number of invocations, denoted as $NI(f_u)$, which depends on the structure of the serverless workflow. Branches can reduce the number of invocations of vertices in them since the transition probability of paths in branches is less than 1. Conversely, cycles and self-loops can lead to more than one invocation of functions in them, and the number of invocations depends on the expected value of the number of cycle/self-loop iterations, calculated as Equation (4.10). Hence, for a given cycle $G_c = (V_c, E_c)$ between vertices f_u and f_v, we have the expected value of the number of invocations of each vertex $f_i \in V_c$ as

$$NI\left(f_i\right) = 1 + EI(G_c) = \frac{1}{1 - P(f_v, f_u)}, \forall f_i \in V_c \tag{4.13}$$

Similarly, for a self-loop $G_l = (\{f_u\}, \{(f_u, f_u)\})$, we have

$$NI\left(f_u\right) = 1 + EI(G_l) = \frac{1}{1 - P(f_u, f_u)} \tag{4.14}$$

The cost model first leverages Equation (4.13) and Equation (4.14) to calculate the average number of invocations of vertices in cycles and self-loops. Then, similar to procedures of processing cycles and self-loops in the performance model, the cost model updates the edge and transition probability to remove cycles and self-loops from the graph. After removing all cycles and self-loops, considering the impact of

parallels and branches, the cost model updates NI of each vertex based on the sum of transition probabilities of all simple paths from the start node to it. By following these steps, we convert G_{cost} into a de-looped graph used for cost modeling, denoted as G_{dl}. The average cost of the serverless workflow can be calculated by $\sum_{f \in V} C(f)$. Algorithm 3 gives the pseudo-code of the cost model. An example of using the cost model for a serverless application is discussed in Section 4.5.

Algorithm 3: Cost modeling algorithm

Input: a workflow graph G_{cost}
Output: the average cost for each execution of the serverless application

$G' \leftarrow G_{cost}$;
for all $f \in V$, let $NI(f) \leftarrow 1$; ▷ `initialization`
while G' *is not a DAG* **do**
  loop_list ← find_self_loops(G'); ▷ `section 4.2.4`
  for *self-loop* $G_l = (f_u, (f_u, f_u))$ *in loop_list* **do**
    $NI(f_u) \leftarrow \frac{1}{1-P(f_u,f_u)}$; ▷ `eq. (4.14)`
    $E \leftarrow E \backslash \{(f_u, f_u)\}$; ▷ `remove the loop edge`
    for *each edge e in out*(f_u) **do**
      $P \leftarrow P\left[e \mapsto \frac{P(e)}{1-P(f_u,f_u)}\right]$; ▷ `updat prob.`
    end
  end
  cycle_list ← find_cycles(G');
  for *each cycle* $G_i = (V_i, E_i)$ *in cycle_list* **do**
    For each $f \in V_i$ update $NI(f)$ using eq. (4.13);
    Remove the cycle edge;
    Update transition probabilities of outgoing edges; ▷ `similar to steps for self-loops`
  end
end
for *each vertex* $f \in V$ **do**
  $tp\_sum \leftarrow \sum_{s \in ASP_{G'}(f_{str}, f)} TPP(s)$;
  if $tp\_sum < 1$ **then**
    $NI \leftarrow NI[f \mapsto NI(f) \cdot \text{tp\_sum}]$;
  end
  $cost\_sum \leftarrow cost\_sum + C(f)$; ▷ `eq. (4.12)`
end
return *cost_sum*

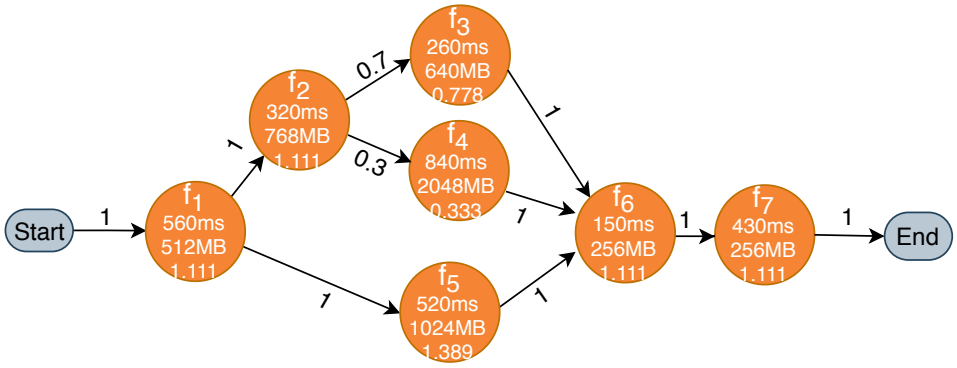


Figure 4.3: The de-looped workflow of the serverless application.

4.5 Example and Analysis

The implementation of the performance model is presented in Algorithm 1. Based on definitions of structures in the serverless workflow, as mentioned in Section 4.1, the algorithm identifies structures in the workflow and leverages different procedures to process structures, defined in Section 4.3, to trim the workflow graph to a probabilistic DAG and obtain the end-to-end response time of the application. Figure 4.2 illustrates the step-by-step changes of the workflow graph when the performance model works on the serverless workflow shown in Figure 1.4. The average end-to-end response time of the application is 2111.11 ms.

Algorithm 3 describes the implementation of the cost model. Taking the serverless workflow showing in Figure 1.4 as an example, after completing the first while loop, the algorithm trims the workflow to a de-looped graph shown in Figure 4.3, in which the three numbers on each function represent the response time, allocated memory and the average number of invocations, respectively. The number on each edge represents the transition probability. Then, in the for loop, the algorithm updates the average number of invocations for each function again and calculates the cost of the applications. The average cost of the application is $41.82 per 1 million executions.

Let us consider the worst-case scenario and analyze the time complexities of the

performance and cost models. Under the definition of the serverless workflow mentioned in Section 4.1, the most complex workflow is composed of as many cycles and self-loops as possible, since the cycle and self-loop structures require the least number of vertices, compared to the parallel and branch structures. Figure 4.4 illustrates the workflow under the worst-case scenario, where the workflow is composed of n functions, $(n+2)$ vertices, $\frac{n(n+3)+2}{2}$ edges, $\frac{n(n-1)}{2}$ cycles, and n self-loops. The time complexities for detecting self-loops and cycles are $O(|E|)$ and $O(C(|V|+|E|))$, respectively, where C is the number of cycles. The performance model and cost model can process cycles and self-loops in $O(n^2)$ and $O(n)$ time, respectively. Hence, the time complexities for the performance and cost modeling under the worst-case scenario are $O(n^6)$ and $O(n^5)$, respectively, where n is the number of functions in the workflow. We empirically analyze the AWS and Azure official repositories [121, 122]. The average number of functions in serverless applications in this repository, orchestrated by Amazon Step Functions or Microsoft Azure Functions, is less than 5. We do not find any serverless application in the repositories resembling the worst-case topology. The most common typologies appear to be sequential, paralleled, and branched. Our proposed models can calculate the performance and cost of a worst-case topology application with 27 functions and 406 edges in less than a second on a laptop with a 2.70GHz Intel Core i7-3740QM processor and 16 GB of memory. This shows the applicability of the models for now and the foreseeable future.

4.6 Experimental Evaluation

We implement the performance and cost models using Python 3.8 and validate them by conducting experiments on five serverless applications deployed on AWS.

4.6.1 Experimental Design

To evaluate performance and cost models, we design five serverless applications composed of a various number of FaaS functions with mixed types of workload. We

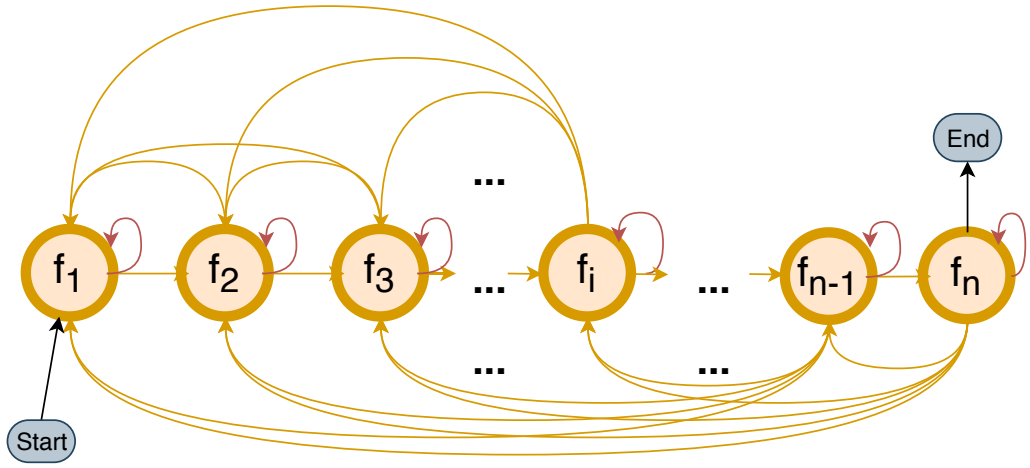


Figure 4.4: The workflow of the worst-case scenario.

deploy functions on AWS Lambda and employ AWS Step Functions as the serverless application coordinator. AWS Step Functions is a serverless workflow coordination service that combines multiple Lambda functions and other serverless services offered by AWS into responsive serverless applications [40].

We design FaaS functions with three different types of workload, namely CPU intensive, disk I/O intensive, and network I/O intensive. The CPU-intensive workloads include string hashing, floating-point arithmetic, and recursive calculation. The disk-intensive workload is to write and read several files to the hard disk drive. The network-intensive workload is designed to download and upload a number of files from and to the AWS S3 bucket. We develop sixteen functions with different input sizes and types of workload and host them on AWS Lambda.

By leveraging AWS Step Functions, we develop five serverless applications using those sixteen functions, which are App8, App10, App12, App14, and App16. The numeric suffix of the application name represents the number of functions in the application. From App8 to App16, we increase the number of functions from eight to sixteen, as well as the number of structures (parallels, branches, cycles, and self-loops) in the application workflow. As a result, the complexity of the workflow grows ac-

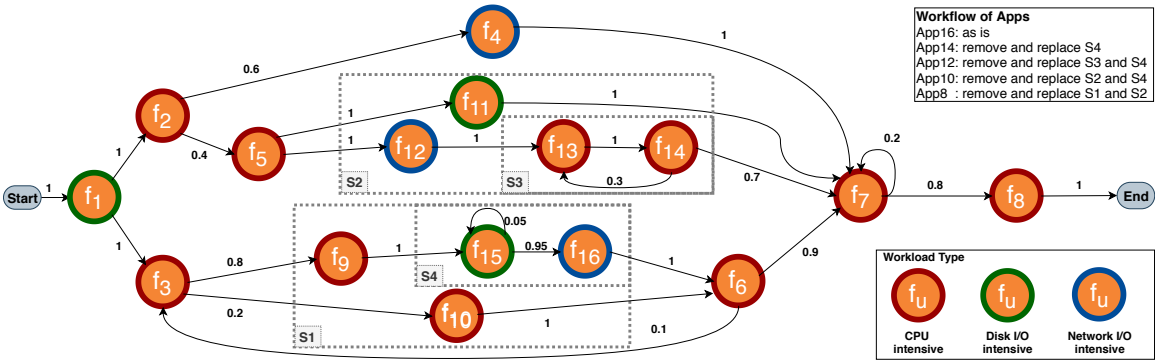


Figure 4.5: Workflow of serverless applications used in the experimental evaluation of performance and cost models.

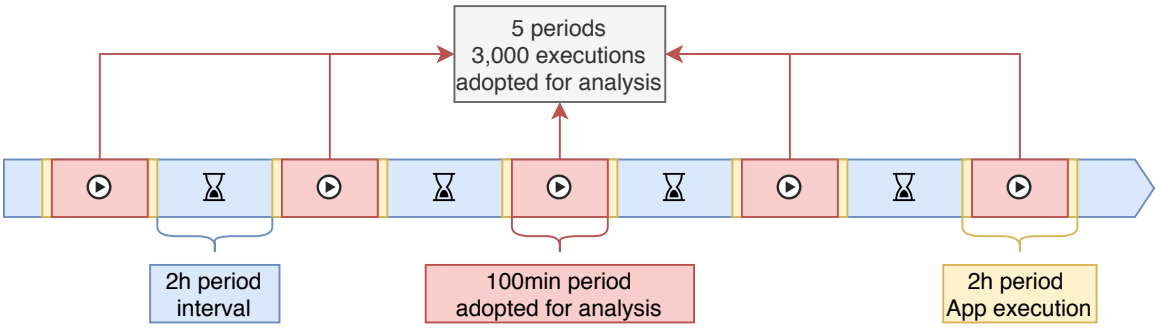


Figure 4.6: The experimental design for evaluating performance and cost models.

cordingly. Besides, each application has a combination of all three types of workloads, making them truly representative of actual serverless applications on the cloud. The workflow of five serverless applications is shown together in Figure 4.5. The number on each edge represents the transition probability. The workflow of App16 is as shown. The workflow of the other four applications can be obtained by removing and replacing all edges and functions in the corresponding box(es) with an edge whose transition probability is 1. The transition delay is defined using the delay model of AWS Step Functions. The applications are composed of functions with three different types of workload depicted by three colors.

We deploy sixteen functions on AWS Lambda and obtain their average duration under a feasible memory configuration by invoking each of them 720 times. Five aforementioned serverless applications are developed and deployed on AWS Step Functions.

For repeatability and rigorousness, we follow the methodological principles proposed by Papadopoulos et al. [123] and execute repeated experiments and long runs. As the timeline shown in Figure 4.6, We execute each application for five periods of two hours with a two-hour interval between two consequent periods. During each period, each application is executed continuously with a 10-second interval between two consequent invocations. For each execution period, we discard executions in the first and last 10 minutes to avoid any transient fluctuations in performance and only adopt invocations between them. By doing so, for each application, logs of 3,000 invocations are ready for analysis.

4.6.2 Experimental Result

By giving the workflows and the average duration and allocated memory size of functions as inputs, we leverage the proposed performance and cost models to obtain the average end-to-end response time and cost of five serverless applications. By analyzing logs of 3,000 invocations for each application, we compare the results of performance and cost models with the duration and billing logs reported by AWS.

Figure 4.7 and Figure 4.8 illustrate the experimental evaluation result of the performance and cost models, where the box plot shows the maximal value, 25%, 50%, 75% percentiles, and the minimum value of response time and cost. The notch shows the 95% confidence interval for the median of response time and cost. The average accuracy for performance modeling is 98.75%, while the average accuracy for cost modeling is 99.97%.

As the number of functions in the application increases from 8 to 16, the workflow becomes more complex in terms of structures. As is evident from the figures, both the average end-to-end response time and cost derived by the proposed performance and cost models are very close to the real values reported by AWS. Regardless of the complexity of the workflow, the accuracy of the predicted average end-to-end response time and cost is over 97.5%. Such results indicate the high accuracy of the proposed

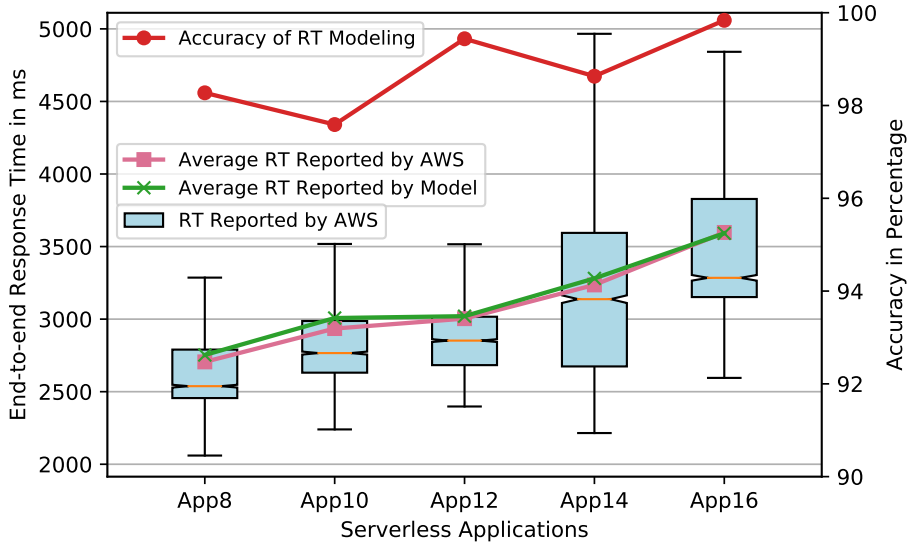


Figure 4.7: Experimental evaluation result of the performance model.

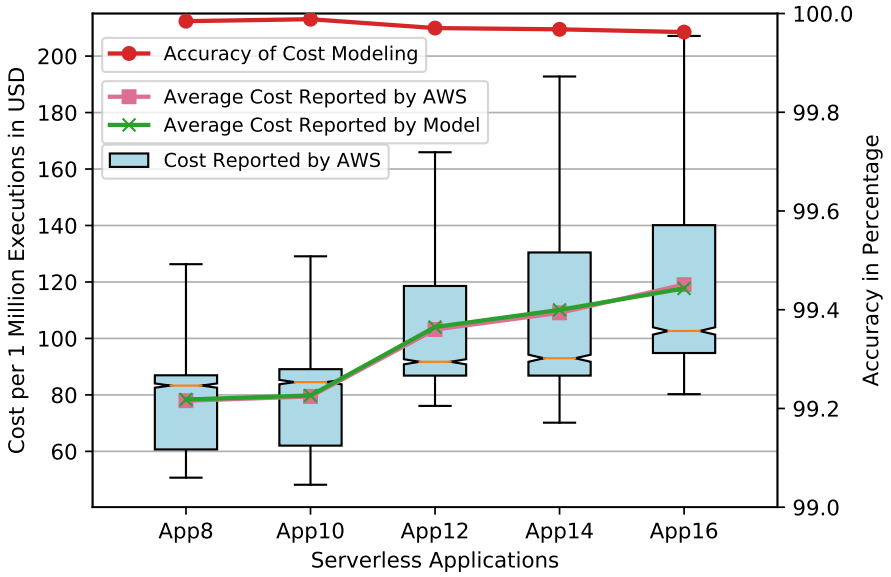


Figure 4.8: Experimental evaluation result of the cost model.

performance and cost models.

4.7 Summary

In this chapter, we achieved Objective 2 by proposing and evaluating two analytical models of serverless applications. For a given orchestration and performance and memory configuration of a serverless application, the proposed models can obtain the average end-to-end response time and cost of the application. To evaluate the models, we developed serverless applications with eight to sixteen functions and deployed them on AWS. The accuracy of predicted average end-to-end response time and cost is 98.75% and 99.97%, respectively. The evaluation results indicate the high accuracy of the proposed performance and cost models.

Chapter 5

Performance and Cost Optimization for Serverless Applications

In this chapter, to accomplish Objective 3, we propose a heuristic algorithm named Probability Refined Critical Path (PRCP) Algorithm to optimize the performance and cost of serverless applications. As mentioned in Chapter 1, the response time of the function varies with the allocated memory, and so does the cost. Therefore, developers can tune the performance and cost of the serverless application by changing the allocated memory size of functions in the application. More specifically, as described by the motivation, very practical problems in serverless computing are to get the best performance under a limited budget or satisfy the performance constraint by the minimum cost. Therefore, Objective 3 leads to two performance and cost optimization problems. In this chapter, we first introduce the performance profile, then define the optimization problem, and present and evaluate the PRCP algorithm.

For consistency and brevity, we continue to use the definitions, notations, and performance and cost models presented in Chapter 4. Besides notations defined in Table 4.1, Table 5.1 lists all other notations used in this chapter.

Table 5.1: Definition of notations used in Chapter 5.

Notation	Definition
$W(s)$	weight of a simple path s
BCR	Benefit-cost ratio
TH	BCR threshold
$\beta(f_u)$	the slope of the performance-memory curve of f_u

5.1 Performance Profile of Serverless Functions

On FaaS platforms, increasing the memory of a function can improve the CPU and IO performance, but it does not necessarily reduce the response time of the function significantly, especially when the performance becomes insensitive to the memory, i.e., the amount is large enough. As shown in Figure 1.3, when the allocated memory is greater than 1792 MB, the response time remains almost the same, and at this time, the continued increase in memory size incurs additional cost due to the rounding and granularity of billed duration. Similarly, albeit a larger allocated memory size leads to a higher price per second of billed duration, the cost may decrease with larger memory, especially before the performance becomes insensitive. For the performance-memory curve of functions, as the allocated memory size increases, response time decreases first and then levels out since the performance becomes insensitive. However, because of the rounding and billing granularity, there are large fluctuations in the cost-memory curve of functions.

Therefore, as mentioned in the best practices for working with AWS Lambda Functions, to find an optimal allocated memory size satisfying both the trade-off between performance and price and the required memory size for the function execution, a performance profiling phase is highly recommended to test the performance of the FaaS functions [119]. In the performance profiling phase, the function is invoked using the payload with the average input size multiple times under different allocated memory sizes, and the average duration of invocations is logged. By doing so, we can

acquire a set of viable memory sizes and a series of function response times under different memory sizes. Figure 2 demonstrates the performance-memory curve of a function obtained in the performance profiling phase. Explicitly, we let $MOpt$ denote a memory option function mapping the function to its viable memory options, defined as Equation (5.1).

$$MOpt : V \to \wp(\mathbb{N}) \tag{5.1}$$

The performance profile of a function f_u, which describes the response time of the function under different allocated memory sizes, is defined as Equation (5.2). $PF(f_u, mem_v)$ is the response time of f_u with the allocated memory size of mem_v, where $f_u \in V$ and $mem_v \in MOpt(f_u)$.

$$PF : \biguplus_{f \in V} \{\{f\} \times MOpt(f)\} \to [0, +\infty) \tag{5.2}$$

Considering the performance profile of functions, in this section, we extend the definition of the serverless workflow defined in Chapter 4 as

$$G_s = (V, E, P, D, RT, RTTP, M, NI, C, MOpt, PF)$$

where $MOpt$ and PF are defined as Equation (5.1) and Equation (5.2), respectively. Let $MOpt(f) \triangleq \varnothing$ for all f is not a FaaS function.

5.2 Problem Statement

Considering a serverless workflow G_s with n functions, where $V = \{f_1, f_2, ..., f_{n-1}, f_n\}$, we define π as a memory configuration of the workflow, such that

$$\pi \in MOpt(f_1) \times MOpt(f_2) \times ... \times MOpt(f_n)$$

$\pi(f)$ denotes the size of the allocated memory of the function f in this configuration. Let ERT^π denote the end-to-end response time of G_s obtained by the performance model and $C^\pi(f_u)$ denote the cost of the function f_u obtained by the cost model, under the memory configuration π, such that (i) $M(f_i) = \pi(f_i)$, for all $1 \leq i \leq n$,

(ii) $RT(f_i) = PF(f_i, \pi(i))$, for all $1 \leq i \leq n$. For a given budget limit BC, and a performance constraint PC, we define the following two optimization problems.

5.2.1 Best Performance under Budget Constraint

Find a memory configuration π that achieves the minimum average end-to-end response time of the application with the average cost less than or equal to the budget BC. We call such a problem the Best Performance under Budget Constraint (BPBC) problem.

$$\begin{aligned} &\arg\min_{\pi} \quad ERT^{\pi}(G_s) \\ &subject\ to \quad \sum_{i=1}^{n} C^{\pi}(f_i) \leq BC \end{aligned} \tag{5.3}$$

5.2.2 Best Cost under Performance Constraint

Find a memory configuration π that achieves the minimum average cost of the application with the average end-to-end response time less than or equal to the performance constraint PC. We call such a problem the Best Cost under Performance Constraint (BCPC) problem.

$$\begin{aligned} &\arg\min_{\pi} \quad \sum_{i=1}^{n} C^{\pi}(f_i) \\ &subject\ to \quad ERT^{\pi}(G_s) \leq PC \end{aligned} \tag{5.4}$$

5.3 Problem Complexity Analysis

We prove that BPBC and BCPC problems in the serverless computing paradigm are fundamentally more complex variants of the multiple-choice knapsack problem (MCKP). MCKP is formulated as follows. Given n sets $\mathbb{N}_1, \mathbb{N}_2, ..., \mathbb{N}_n$ of items, where each item in each set has a profit and a weight, by selecting exactly one item from each set, the optimization problem is to find a selection combination $\varsigma \in \mathbb{N}_1 \times \mathbb{N}_2 \times ... \times \mathbb{N}_n$ such that ς maximizes the total profit while the total weight within the capacity. MCKP has been applied to many optimization problems, including resource

allocation and workflow optimization in parallel computing and microservice-based applications[90, 92].

In BPBC and BCPC problems, the n functions, viable allocated memory sizes of each function $MOpt$, and the memory configuration π are equivalent to n sets, a number of items in each set, and selection combination ς in MCKP, respectively. For the BPBC problem, BC corresponds to the knapsack capacity constraint, $-PF\left(f, M_k\right)$ can be viewed as the profit of the function $f \in V$ with the allocated memory size of $M_k \in MOpt(f)$, and corresponding cost $C(f)$ is equivalent to the weight. Instead of simply accumulating values, the total profit in the BPBC problem represents the end-to-end response time of the application, should be derived by the performance modeling algorithm defined in Algorithm 1. Similarly, for the BCPC problem, PC corresponds to the knapsack capacity constraint, the cost $C(f)$ of the function f under the memory $M_k \in MOpt(f)$ can be viewed as the profit of the function, the response time $PF\left(f, M_k\right)$ can be deemed as the weight.

Compared to MCKP, the higher complexity of BPBC and BCPC problems lies in calculating the total profit and total weight, which leverages the polynomial-time performance and cost models defined in Chapter 4. Besides, for a given memory configuration and corresponding response time, we can check whether the average end-to-end response time and total cost under such a configuration satisfy the constraints in polynomial time. Hence, BPBC and BCPC problems in the serverless computing paradigm are fundamentally more complex variants of MCKP.

As MCKP has proven to be an NP-complete problem without any solutions in polynomial time, unless $P = NP$ [124], we have to resort to a heuristic algorithm to solve BPBC and BCPC problems.

5.4 Probability Refined Critical Path Algorithm

In this section, we propose a heuristic algorithm based on the critical path method to solve BPBC and BCPC problems.

5.4.1 Critical Path Method

Critical path method (CPM) is a heuristic approach for scheduling problems, which optimizes the scheduling scheme by identifying and rescheduling the path with the longest execution time [125]. CPM has proven to be an effective solution to scheduling problems in many areas, including the scientific workflow system [90], distributed computing framework [88], IaaS paradigm [89], and CaaS paradigm [92]. However, previous studies have largely applied CPM to DAG workflows. As mentioned in Section 1.5.2, there can be cycles and loops in the serverless workflow, to which the traditional CPM is not applicable.

5.4.2 Algorithm Design

We propose a Probability Refined Critical Path Algorithm (PRCP) to solve BPBC and BCPC optimization problems for non-DAG serverless workflows. To work with non-DAG workflow topology, PRCP refines the transition probability of edges and simple paths as well as the weight of simple paths based on the transition probability and leverages a weight-based definition of the critical path. PRCP recursively optimizes the memory of functions on the critical path using a greedy manner and obtains the best memory configurations satisfying the constraint.

Based on the definition of the serverless workflow, due to transition probabilities and structures in the workflow, the path with the longest delay may not be the critical path in terms of the response time and cost. Therefore, we need to re-define the critical path to take the transition probability of the path and iterations incurred by cycles and loops into account.

For a simple path $s_k = f_1 e_1 f_2 e_2 ... e_{n-1} f_n$, we define the weight of s_k as Equation (5.5).

$$W(s_k) = \sum_{i=1}^{n} RT(f_i) NI(f_i) TPP(s_k) \tag{5.5}$$

Given a set of M simple paths, we define the path with i^{th} greatest weight as the

i^{th} critical path, such that $1 \leq i \leq M$. We denote $FindCriticalPath(G, i)$ as a procedure that returns the i^{th} critical path in G.

Given a function f_u and a new memory size $mem_v \in MOpt(f_u)$, by assigning the new allocated memory size mem_v to f_u, we define the change of end-to-end response time and the change of cost, as $\Delta ERT\left(f_u, mem_v\right)$ and $\Delta C\left(f_u, mem_v\right)$, respectively, where

$$\Delta ERT\left(f_u, mem_v\right) = ERT^{\gamma}\left(G_s\right) - ERT_{curr}$$
$$\Delta C\left(f_u, mem_v\right) = \sum_{i=1}^{n} C^{\gamma}\left(f_i\right) - C_{curr}$$

such that (i) ERT_{curr} and C_{curr} are the end-to-end response time and cost under the previous configuration π; (ii) $\gamma\left(f_u\right) = mem_v$; (iii) $\gamma\left(f_i\right) = \pi\left(f_i\right)$ for all $1 \leq i \leq n$ and $i \neq u$.

Algorithm 4 demonstrates the pseudocode of the PRCP algorithm solving the BPBC problem. The input are the budget constraint BC and a serverless workflow G_s with n functions, where $V = \{f_1, f_2, ..., f_{n-1}, f_n\}$. PRCP first initializes the workflow by employing the minimum memory configuration π_{min}, such that $\pi_{\min}\left(f_i\right) = \min\left(MOpt\left(f_i\right)\right)$ for all $1 \leq i \leq n$. The workflow has the largest end-to-end response time under the minimum memory configuration. PRCP recursively finds the critical path and calculates ΔERT and ΔC under all possible allocated memory size of all functions in the critical path. Suppose there are functions and memory size options that can reduce the cost without increasing the end-to-end response time of the workflow, namely $\Delta C < 0$ and $\Delta ERT \leq 0$, PRCP selects the function and memory size resulting in the largest cost decrease. If not, PRCP chooses the function and memory size achieving the largest end-to-end response time decrease within the budget constraint. If there is no feasible memory size option within the t^{th} critical path, where t is initialized as 1, PRCP will find the $(t+1)^{th}$ critical path in the next iteration, and so on. The iteration ends when there is no feasible memory size option in the least critical path with the current surplus.

Algorithm 5 provides the pseudocode of the PRCP algorithm solving the BCPC

Algorithm 4: PRCP-BPBC

Input: Budget constraint BC, serverless workflow $G_s = (V, E, P, D, RT, RTTP, M, NI, C, MOpt, PF)$

Output: memory configuration π satisfying eq. (5.3)

$M \leftarrow M\left[f \mapsto \min\left(MOpt\left(f\right)\right)\right], \forall f \in V$; ▷ `use the minimum memory configuration` π_{min}

$G_{dl} \leftarrow (V, E', P', RT, M, NI, C, MOpt, PF)$; ▷ `Get the de-looped graph using steps in Section 4.4`

$NSP \leftarrow |ASP_{G_{dl}}\left(f_{str}, f_{end}\right)|$; ▷ `number of simple paths in` G_{dl}

$\pi_{curr} \leftarrow \pi_{min}$; ▷ `current memory configuration`

$C_{curr} \leftarrow \sum_{i=1}^{n} C^{\pi_{min}}(f_i)$; ▷ `current cost`

$ERT_{curr} \leftarrow ERT^{\pi_{min}}\left(G_s\right)$; ▷ `current end-to-end RT`

$t \leftarrow 1$;

while $BC - C_{curr} > 0$ *and* $t \leq NSP$ **do**

  $s_{cp} \leftarrow FindCriticalPath\left(G_{dl}, t\right)$; ▷ `find` t^{th} `critical path in` G_{dl}

  for *all functions* f_i *in* s_{cp} **do**

    for *all selectable memory* $mem_j \in MOpt(f_i)$ **do**

      Calculate $\Delta ERT\left(f_i, mem_j\right)$ and $\Delta C\left(f_i, mem_j\right)$;

    end

  end

  if $\exists mem_v \in MOpt\left(f_u\right)$*:* $\Delta ERT\left(f_u, mem_v\right) \leq 0$ *and* $\Delta C\left(f_u, mem_v\right) < 0$ **then**

    $f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta C\left(f_u, mem_v\right)$;

  else

    $\boxed{f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta ERT\left(f_u, mem_v\right)}$ s.t. $\Delta C\left(f_u, mem_v\right) \leq BC - C_{curr}$ and $\Delta ERT\left(f_u, mem_v\right) < 0$. If there are multiple functions and memory values that achieve the same maximum ΔERT, select f_u and mem_v that leads to the smallest ΔC;

  end

  if f_{tmp} *and* mem_{tmp} *exist* **then**

    $M \leftarrow M\left[f_{tmp} \mapsto mem_{tmp}\right]$;

    $RT \leftarrow RT\left[f_{tmp} \mapsto PF\left(f_{tmp}, mem_{tmp}\right)\right]$;

    $\pi_{curr} \leftarrow \pi_{curr}\left(f_{tmp}\right) \triangleq mem_{tmp}$;

    $C_{curr} \leftarrow C_{curr} + \Delta C\left(f_{tmp}, mem_{tmp}\right)$;

    $ERT_{curr} \leftarrow ERT_{curr} + \Delta ERT\left(f_{tmp}, mem_{tmp}\right)$;

  else

    $t \leftarrow t + 1$;

  end

end

return π_{curr}

problem. For the BCPC problem, the PRCP algorithm first initializes the workflow by employing the maximum memory configuration π_{max}, such that $\pi_{\max}(f_i) = \max(MOpt(f_i))$ for all $1 \leq i \leq n$. The workflow has the shortest end-to-end response time under the maximum memory configuration. PRCP recursively finds the critical path, but starts with the path with the smallest weight, then calculates ΔERT and ΔC under all possible allocated memory size of all functions in that path. Suppose there are functions and memory size options that can reduce the end-to-end response time of the workflow without incurring additional cost, namely $\Delta ERT < 0$ and $\Delta C \leq 0$, PRCP selects the function and memory size leading to the largest end-to-end response time decrease. If not, PRCP chooses the function and memory size resulting in the largest cost decrease within the performance constraint. If there is no feasible memory size option within the t^{th} critical path, where t starts with the number of simple paths in G_{dl}, PRCP will find the $(t-1)^{th}$ critical path in the next iteration, and so on. The iteration ends when there is no feasible memory size option even in the most critical path with the current surplus in terms of the performance.

5.4.3 Benefit/Cost Ratio Greedy Strategies

PRCP algorithm is essentially a greedy heuristics for BPBC and BCPC problems. The steps with a box in Algorithm 4 and Algorithm 5 are greedy strategies. For instance, in the BPBC problem, the greedy strategy is to find the f_u and $mem_v \in MOpt(f_u)$ in each critical path iteration, which can lead to the greatest end-to-end response time reduction. However, the local optimization achieved by such a strategy might compromise the global optimization. As mentioned in Section 5.1 and shown in Figure 1.3, after the performance becomes insensitive to the memory, the performance gain of the function brought by the increase in memory is insignificant, and due to rounding and billing granularity, the cost may increase significantly.

To avoid bad optimization solutions, we introduce the benefit/cost ratio (BCR) into greedy strategies. Instead of arbitrarily maximizing the end-to-end response

Algorithm 5: PRCP-BCPC

Input: Performance constraint PC, serverless workflow $G_s = (V, E, P, D, RT, RTTP, M, NI, C, MOpt, PF)$

Output: memory configuration π satisfying eq. (5.3)

$M \leftarrow M\left[f \mapsto \max\left(MOpt\left(f\right)\right)\right], \forall f \in V$; ▷ `use the maximum memory configuration` π_{max}

$G_{dl} \leftarrow (V, E', P', RT, M, NI, C, MOpt, PF)$; ▷ `Get the de-looped graph using steps in Section 4.4`

$NSP \leftarrow |ASP_{G_{dl}}\left(f_{str}, f_{end}\right)|$; ▷ `number of simple paths in` G_{dl}

$\pi_{curr} \leftarrow \pi_{max}$; ▷ `current memory configuration`

$C_{curr} \leftarrow \sum_{i=1}^{n} C^{\pi_{max}}(f_i)$; ▷ `current cost`

$ERT_{curr} \leftarrow ERT^{\pi_{max}}\left(G_s\right)$; ▷ `current end-to-end RT`

$t \leftarrow$ NSP;

while $PC - ERT_{curr} > 0$ *and* $t \geq 1$ **do**

  $s_{cp} \leftarrow FindCriticalPath\left(G_{dl}, t\right)$; ▷ `find` t^{th} `critical path in` G_{dl}

  for *all functions* f_i *in* s_{cp} **do**

    for *all selectable memory* $mem_j \in MOpt(f_i)$ **do**

      Calculate $\Delta ERT\left(f_i, mem_j\right)$ and $\Delta C\left(f_i, mem_j\right)$;

    end

  end

  if $\exists mem_v \in MOpt\left(f_u\right)$*:* $\Delta C\left(f_u, mem_v\right) \leq 0$ *and* $\Delta ERT\left(f_u, mem_v\right) < 0$ **then**

    $f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta ERT\left(f_u, mem_v\right)$;

  else

    $\boxed{f_{tmp}, mem_{tmp} \leftarrow \underset{f_u, mem_v}{\arg\min}\, \Delta C\left(f_u, mem_v\right)}$ s.t. $\Delta ERT\left(f_u, mem_v\right) \leq PC - ERT_{curr}$ and $\Delta C\left(f_u, mem_v\right) < 0$. If there are multiple functions and memory values that achieve the same minimum ΔC, select f_u and mem_v that leads to the smallest ΔERT;

  end

  if f_{tmp} *and* mem_{tmp} *exist* **then**

    $M \leftarrow M\left[f_{tmp} \mapsto mem_{tmp}\right]$;

    $RT \leftarrow RT\left[f_{tmp} \mapsto PF\left(f_{tmp}, mem_{tmp}\right)\right]$;

    $\pi_{curr} \leftarrow \pi_{curr}\left(f_{tmp}\right) \triangleq mem_{tmp}$;

    $C_{curr} \leftarrow C_{curr} + \Delta C\left(f_{tmp}, mem_{tmp}\right)$;

    $ERT_{curr} \leftarrow ERT_{curr} + \Delta ERT\left(f_{tmp}, mem_{tmp}\right)$;

  else

    $t \leftarrow t - 1$;

  end

end

return π_{curr}

time reduction, the strategy is to find the configuration achieving the optimal BCR. In other words, the idea is to find the configuration leading to the optimal benefit for its cost. We propose three BCR greedy strategies for each optimization problem and integrate them into the PRCP algorithm. For the BPBC problem, strategies are MAX, ERT/C, and RT/M. For the BCPC problem, strategies are MAX, C/ERT, and M/RT.

In the MAX strategy for the BPBC problem, the reduction in end-to-end response time is the benefit, and the cost is the increased cost of the workflow incurred by the configuration. Namely, for the function f_u and a selectable memory size $mem_v \in MOpt(f_u)$, BCR is defined as

$$BCR\left(f_u, mem_v\right) = \left| \frac{\Delta ERT\left(f_u, mem_v\right)}{\Delta C\left(f_u, mem_v\right)} \right| \tag{5.6}$$

The MAX strategy finds the f_u and $mem_v \in MOpt(f_u)$ in each critical path iteration leading to the maximum BCR, which is defined as Equation (5.6).

The ERT/C strategy has the same definitions of benefit and cost as the MAX strategy. A BCR threshold, denoted as TH, is leveraged. The ERT/C strategy keeps a record of the BCR of the configuration in the previous critical path iteration, denoted as BCR_{pre}. Instead of simply maximizing BCR, the ERT/C strategy finds f_u and $mem_v \in MOpt(f_u)$ that results in the maximum BCR such that $BCR \geq BCP_{pre} \cdot TH$.

In the RT/M strategy, we introduce a BCR threshold TH and consider the two-point slope of the performance-memory curve as the BCR, namely

$$BCR\left(f_u, mem_v\right) = \frac{PF\left(f_u, mem_{v+1}\right) - PF\left(f_u, mem_v\right)}{mem_{v+1} - mem_v} \tag{5.7}$$

where $mem_{v+1} \in MOpt(f_u)$ is the adjacent selectable memory size such that $mem_{v+1} > mem_v$. For each function $f_i \in V$, the RT/M strategy algorithm first calculates the slope of the performance-memory curve by the least squares regression, denoted as $\beta\left(f_i\right)$. Then, the algorithm removes all memory options from the viable memory

Table 5.2: Summary of BCR Greedy Strategies.

Strategy	**BCR Def.**	**Maximize**	**Subject To**
		BPBC Problem	
W/O BCR	N/A	ERT Reduction	None
MAX	eq. (5.6)	BCR	None
ERT/C	eq. (5.6)	BCR	$BCR \geq BCR_{pre} \cdot TH$
RT/M	eq. (5.7)	ERT Reduction	$\mathrm{BCR}\left(f_i, mem_j\right) \geq \beta\left(f_i\right) \cdot TH$
		BCPC Problem	
W/O BCR	N/A	Cost Reduction	None
MAX	eq. (5.6) $^{-1}$	BCR	None
C/ERT	eq. (5.6) $^{-1}$	BCR	$BCR \geq BCR_{pre} \cdot TH$
M/RT	eq. (5.7) $^{-1}$	Cost Reduction	$\mathrm{BCR}\left(f_i, mem_j\right) \geq \beta\left(f_i\right) \cdot TH$

options whose BCR is smaller than $\beta\left(f_i\right) \cdot TH$. After this step, the memory option function satisfies $\beta\left(f_i\right) \cdot TH$ for all $1 \leq i \leq n$ and all $mem_j \in MOpt(f_i)$. In each critical path iteration, the algorithm finds the f_u and $mem_v \in MOpt(f_u)$ resulting in the greatest end-to-end response time reduction.

Correspondingly, MAX, C/ERT, and M/RT are three BCR greedy strategies for the BCPC problem, where the BCR is defined as the multiplicative inverse of the BCR defined for the BPBC problem. In the M/RT strategy, the algorithm calculates the slope of the memory-performance curve. Table 5.2 gives the summary of the original PRPC algorithm and BCR greedy strategies. It contains BCR, optimization goal in each iteration, and the conditions to which the optimization is subject. Besides the s.t. conditions mentioned in Table 5.2, as specified in the pseudocode of the PRPC algorithm, all strategies should also satisfy $\Delta C\left(f_u, mem_v\right) \leq BC - C_{curr}$ and $\Delta ERT\left(f_u, mem_v\right) < 0$ for the BPBC problem, and $\Delta ERT\left(f_u, mem_v\right) \leq PC - ERT_{curr}$ and $\Delta C\left(f_u, mem_v\right) < 0$ for the BCPC problem.

5.5 Experimental Evaluation

5.5.1 Experimental Design

To evaluate the proposed performance and cost optimization solution, namely the PRCP algorithm, we develop a serverless application named App6. As the name suggests, App6 is composed of 6 functions with three types of workload (CPU-intensive, disk-intensive, network-intensive). For generality, App6 is designed to have all four types of structures, namely the parallel, branch, cycle, and self-loop.

As described in Section 5.1, to optimize the performance and cost of the serverless application, a performance profiling phase is required to get the feasible memory configuration $MOpt$ and performance profile PF of serverless functions in the application. We deploy functions on AWS Lambda, which allows the allocated memory to vary between 128 MB and 3,008 MB in 64MB increments, resulting in 46 possible choices. We stipulate that all 46 memory sizes are feasible choices for all functions. For each function, we obtain its performance profile by the performance profiling phase, during which the function is invoked 100 times under each feasible memory size, and the duration is logged.

In order to measure the accuracy of solutions given by the PRCP algorithm, an exhaustive search is necessary to obtain the performance and cost of App6 under all possible memory configurations. However, 6 functions with 46 possible memory choices lead to 9.47 billion states, making the exhaustive search computationally unfeasible. Therefore, we trim the number of memory choices while retaining the trend of the response time-memory size curve by sampling the performance profile. After sampling, the viable memory size of each function varies between 128 MB and 3,008 MB in 192 MB increments, 16 choices left. Using the proposed performance and cost models, we exhaustively obtain the average end-to-end response time and cost of App6 under 16,777,216 different memory configurations. As evidenced in Section 4.6, the performance and cost models can accurately give the average end-to-end response

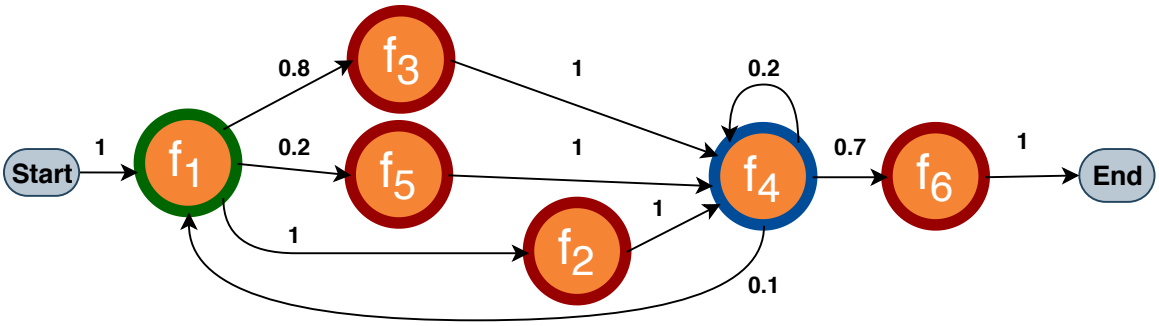


Figure 5.1: The workflow of App6.

time and cost of the serverless application. Therefore, despite the fact that we do not perform the test on AWS Step Functions to get the performance and cost of App6 under 16,777,216 configurations, which is financially and practically unfeasible, the average end-to-end response time and cost derived from proposed models can be regarded as actual values.

We choose the series of 100 equidistant values between the minimum cost and the maximum cost as the budget constraints and execute the PRCP algorithm to solve the BPBC problem with four types of greedy strategies. Similarly, we use the series of 100 equidistant values between the minimum and the maximum end-to-end response time as the performance constraints and solve the BCPC problem. We compare the best performance and the best cost given by the PRCP algorithm with the actual value derived by the exhaustive search under each constraint value.

5.5.2 Experimental Result

Figure 5.1 shows the workflow of App6, in which the legend is the same as Figure 4.5. There are 1 parallel, 1 branch, 1 cycle, and 1 self-loop in App6. The performance profile of functions in App6 is illustrated as Figure 5.2. The allocated memory size varies between 128 MB and 3,008 MB in 64MB increments, resulting in 46 feasible memory choices.

Figure 5.3 depicts the best performance for the BPBC problem achieved by the

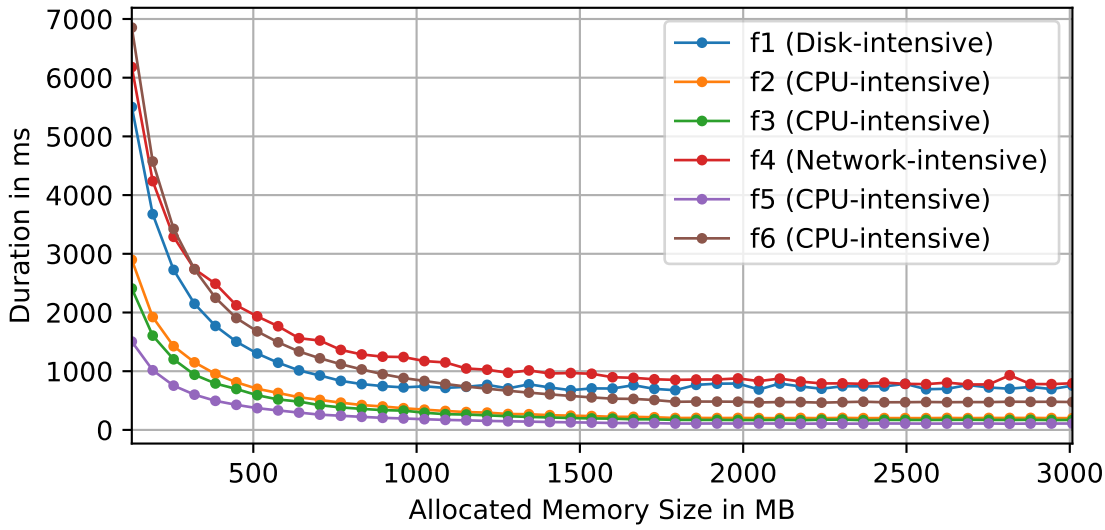


Figure 5.2: The performance profile of 6 functions in App6.

PRCP algorithm with four greedy strategies. The BCR threshold is 0.2. The budget constraints are 100 equidistant values between \$58.86 (minimum cost) and \$163.90 (maximum cost). The average accuracy of the best answer is 97.40%. The best answer is the minimum average response time among solutions. Considering the best performance among solutions given by four strategies, compared to the ideal value, the accuracy of the algorithm is calculated. For the 100 budget constraints, the average accuracy of the PRCP algorithm is 97.40%. Figure 5.4 illustrates the best cost achieved by the PRCP algorithm solving the BCPC problem. The BCR threshold is 0.2. The performance constraints are 100 equidistant values between 2748.24 ms (minimum ERT) and 25433.08 ms (maximum ERT). The average accuracy of the best answer is 99.63%. The best answer is the minimum average cost among solutions. For the 100 performance constraints, the average accuracy of the PRCP algorithm is 99.63%. As is evident from Figure 5.3 and Figure 5.4, the accuracy of different greedy strategies varies with different budget and performance constraints. Therefore, the best method is to employ all four greedy strategies in the PRCP algorithm and select the best solution that is closest to the target performance or cost constraint.

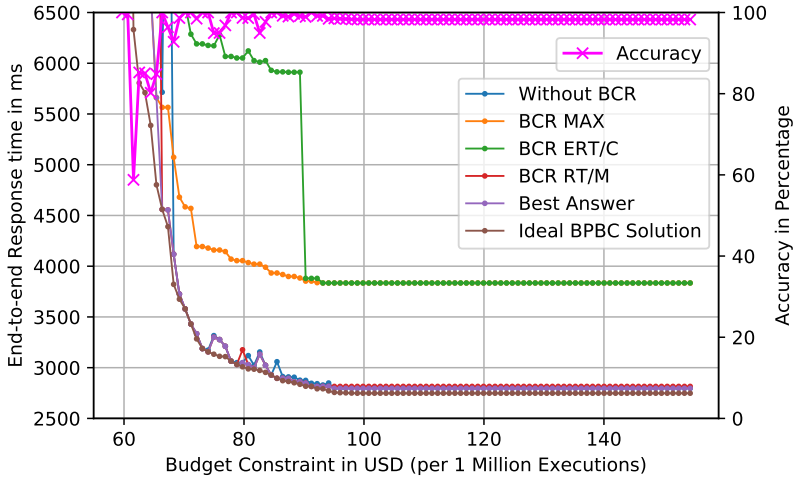


Figure 5.3: The result of the PRCP algorithm solving the BPBC problem.

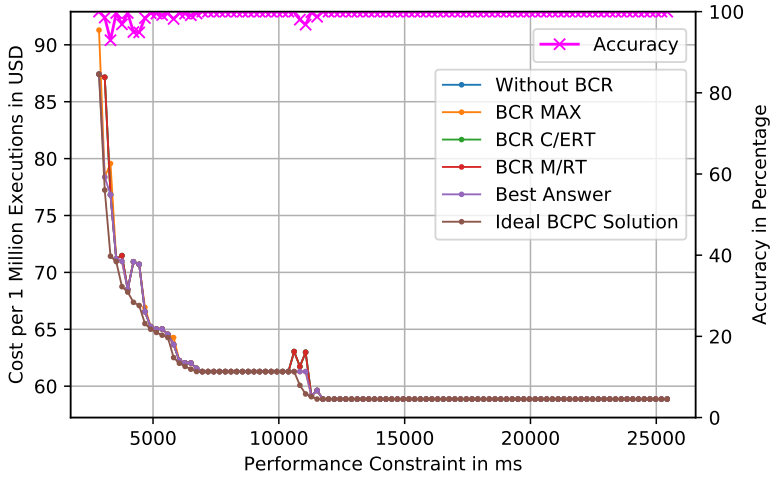


Figure 5.4: The result of the PRCP algorithm solving the BCPC problem.

5.6 Summary

In this chapter, we achieved Objective 3 by presenting a heuristic-based performance and cost optimization algorithm for serverless applications. For a given serverless application orchestration and performance profile of functions, the PRCP algorithm can obtain the memory configuration to solve two optimization problems: the best

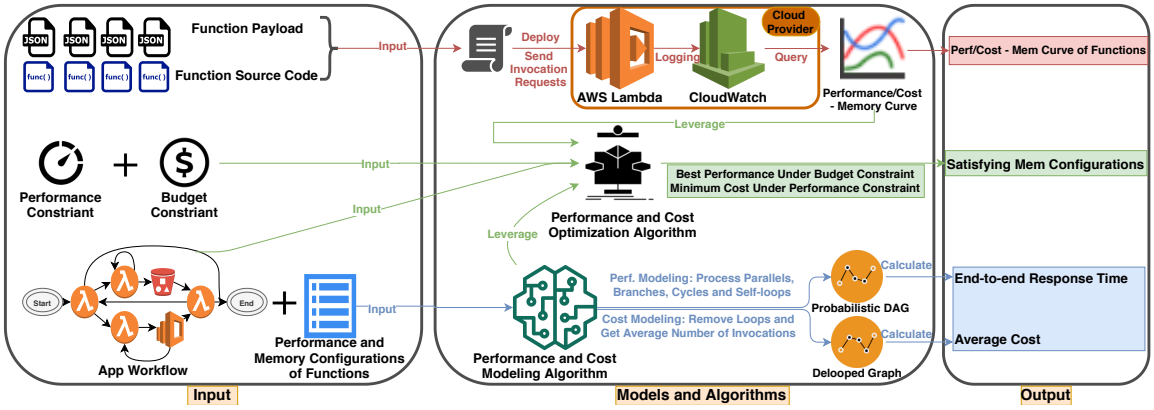


Figure 5.5: Overview of the proposed approach for modeling and optimization of performance and cost of serverless applications in chapter 4 and Chapter 5.

performance under budget constraint and the best cost under performance constraint. To avoid bad optimization solutions, we introduce four greedy strategies based on the benefit/cost ratio into the PRCP algorithm. The proposed algorithm was evaluated by real experiments on AWS. It can achieve the optimal configurations of serverless applications with over 97% accuracy on average.

As there is a tight relationship between performance and cost models and optimization algorithms, in Figure 5.5, we present an overview of the proposed approach in Chapter 4 and Chapter 5. The proposed approach can accomplish three types of tasks (depicted by three different colors): (i) Deploy user-provided function source code and payload to FaaS platform, invoke functions and query the execution log, and give the performance-memory curve of functions as shown in Figure 1.3 as the output. (ii) Take the workflow orchestration of the serverless application and memory configuration and performance (response time) of functions as inputs, process inputs using the performance and cost models, and give the application end-to-end response time and the average cost under the given configuration as the output. (iii) For user-defined performance and cost constraints of a workflow, leverage the proposed performance cost models and optimization algorithms with the performance-memory curve of functions, and give the satisfying memory configuration that achieves the

best performance under a budget constraint or the minimum cost under a performance constraint as the output. The cloud provider part can be replaced with any FaaS platform. We use AWS for experimental evaluation in Chapter 4 and Chapter 5.

Chapter 6

Autonomic Security Management for IoT Smart Spaces

In this chapter, we address Objective 4 by presenting a solution to autonomic security management for IoT smart spaces. We focus on the microservice-based IoT systems where each IoT device provides certain services, and the interactions between users and smart spaces could be viewed as dynamically enabling and disabling services provided by different devices. Due to the heterogeneity of context, we propose a generic ontology named Secure Smart Space Ontology (SSSO) for describing dynamic contextual information in security-enhanced smart spaces. Based on SSSO, we develop a MAPE-k engine that can monitor and analyze the context, and plan and execute countermeasures to achieve autonomic security control. As the evaluation, we conduct a case study on a current BlackBerry customer problem.

6.1 Background and Problem Formulation

Coupled with the accelerating development of computing and telecommunication technology is the fact that more and more interconnected computing devices, or IoT devices, are utilized by enterprises to facilitate business growth. These devices can be mobile such as smartphones and smart bands, or stationary, like smart doors and smart boards. In any case, these devices are equipped with sensors, software, and micro-controllers, that use the underlying network to transfer collected data and

control points [126]. Embedded sensors and smart devices have turned the environments around us into smart spaces that could automatically evolve depending on the need of users and adapt to the new conditions. While smart spaces are beneficial and desired in many aspects, they could be compromised and expose privacy, security or render the whole environment a hostile space in which regular tasks cannot be accomplished anymore. Also, the extraordinary heterogeneity of devices and protocols of IoT devices presents several daunting challenges to developers and managers in a commercial scenario. They need to handle an incredible diversity of hardware, software, and protocols to enable interactions between different systems. When a company wants to deploy new devices, developers have to redevelop some software embedded in previous systems to ensure compatibility with new devices, making the delivery time very long. They also need to cope with complex relationships developed among different devices and correctly map them. Security is always one of the essential concerns for enterprises. How to ensure the safety of connected devices used in critical businesses and avoid disclosure of sensitive information is the vital issue [127]. There is a trade-off between convenience and security. IoT devices can enable automation and intelligent process in enterprises, such as automated door access control and private messaging service. However, IoT devices are also vulnerable to different types of attacks due to high exposure, limited computational resources, and low reliability. In fact, because of the high cost of development, deployment, and maintenance of IoT devices, plus security concerns, the widespread use of IoT solutions in enterprises is just the aspiration, not the reality [44].

In order to tackle the crux of popularizing and applying IoT solutions in enterprises, IoT systems must be able to meet the enterprise-level security bar without incurring prohibitive costs of development, deployment, and management. Systems built on IoT devices must be reliable and scalable and have fine-grained security management components. Fundamental changes should be made in system architecture and security management policies at different levels, including hardware, network,

and software, to address business pain points. The following example highlights the security challenges in a smart space.

Let us consider a smart factory embedded with smart devices. There is a smart camera to monitor the workshop, a smart door lock for entrance control, and several sensors to collect machine process data. It is 1:00 am, and one of the sensors finds that the temperature of the fluid in one pipeline is abnormally high and sends the warning data to the production management system. The production management system automatically shuts down one of the machines in the workshop and sends an alert to a maintenance specialist. Private messaging service on his smartphone informs the maintenance specialist, then he immediately opens his laptop, logs in to the web interface provided by the production management system, and sees real-time process data. After locating the problem, he grants entry permissions to several on-site technicians and asks them to fix the machine. The smart door lock authenticates those technicians by their smart bands. The specialist helps them remotely with smart cameras. In minutes, the team solves the problem, and the machine is running again. This application scenario is very straightforward and comprehensible. However, handling security risks in this scenario is a nightmare for developers and factory managers. For example, there is a risk of an intruder accessing the workshop using a stolen smart band. Developers determine to use two-factor authentication, detect intrusion by identifying the mismatched face using the smart camera, and then take appropriate action. They must redevelop the software of smart cameras to be able to recognize those technicians' faces. If the workshop manager purchases new IoT devices used for authentication, developers must re-design the whole authentication control loop and develop features to support new authentication methods.

What they do is only to tackle one possible security risk. If a sensor in the workshop malfunctions, if the smart door locker manufacturer adds fingerprint recognition, or if an interloper uses a hacked account to access machine process data, developers have to repeatedly analyze the situation, develop new software, and deploy the new system

for risk mitigation. Since such a procedure is mainly based on manual work, the possibility of human error is very high. Moreover, it is conceivable that there are many other security risks to be discovered by trusted developers and managers. That is to say: We can not always have on-site professionals monitoring the whole environment and finding all possible risks. Such a workflow is not suitable for IoT management in commercial scenarios. Hence, autonomic security management is indispensable for IoT systems, which could make IoT systems more secure and reliable. The costs of developing, deploying, and managing such systems would also be considerably reduced.

6.2 MAPE-K

The core of autonomic security management for IoT is that IoT systems can self-detect various types of vulnerabilities, automatically analyze situations, and autonomically learn and implement appropriate security policies. In this case, manual work will be minimized. Therefore, the chance of human failure is less, and security policies will be more concrete. Since IoT systems can be viewed as distributed computing systems, we can leverage the autonomic computing paradigm, introduced by IBM [128], to enable self-managed security by designing and implementing a Monitor-Analyze-Plan-Execute-Knowledge (MAPE-k) loop plus knowledge base, namely the MAPE-k method. Hereafter, we use "MAPE-k engine" and "autonomic security manager" interchangeably. Figure 6.1 shows the high-level architecture of autonomic security management for IoT with the MAPE-k loop. In the following, we will describe this architecture in more detail.

Monitoring is the first step of the MAPE-k method, and the monitored objects include but are not limited to the working status of connected devices, changes in context, explicit user requests, and data streams. Obviously, the challenge here is to develop applications for heterogeneous IoT devices and collect data from them. Considering the IoT system as a microservice-based application composed of multiple

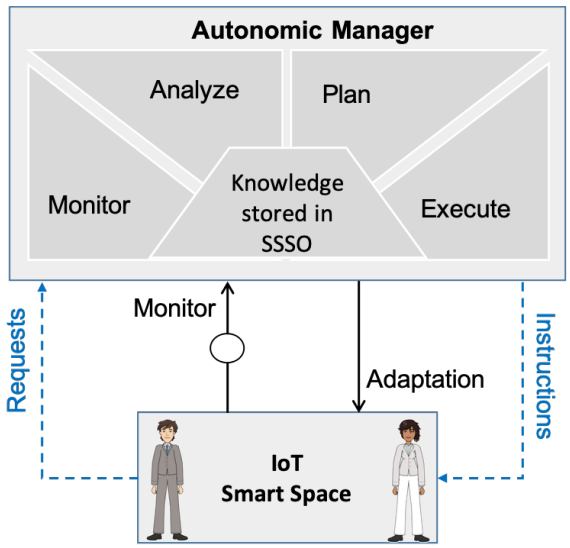


Figure 6.1: The MAPE-k loop and autonomic security management for IoT smart spaces.

microservices, we can adapt some technical patterns widely used in microservice-based web applications and use them in IoT systems, such as API, SDN, containers, access control, to enable high-level development and integrated security policies [27]. Following the view of microservice-based IoT systems, we can assume the IoT system has the ability to sense changes and report anomalies through different microservices. We can also abstract the autonomic manager as a running microservice in the system. Once the anomaly or change is reported to the manager, it will automatically assess the security issue and plan how to mitigate against potential threats by either automatically adapting security policies and implementing them or asking people in the smart space to do explicit actions. All relevant data and actions are stored in the knowledge to help the system reason about threats and make appropriate responses to them. Microservice-based architecture can also offer more options for threat mitigation. For example, when a sensor malfunctions, by unified service discovery, the IoT system can smoothly find the available alternative device which provides the same service.

6.3 Secure Smart Space Ontology

6.3.1 Ontology Design

The ontology usually adheres to the RDF data model expressed as a collection of triples [96]. The underlying structure of the RDF data model can be abstracted as a graph that represents the entities and relationships developed among them. Web Ontology Language (OWL), proposed by World Wide Web Consortium (W3C), is an extension of RDF. Additionally, OWL provides a collection of standard relationships used for RDF [129]. The three components of the OWL ontology are *Classes*, *Properties*, and *Individuals*. Classes provide a basic abstraction of common concepts of things and group things with similar features together. Properties, as the name suggests, describe the relationship between two entities. There are two types of properties: object property used to link two non-data-value entities and data property used to link a non-data-value entity to a data value entity. In our case, we used object properties to describe relationships among IoT devices, and used data properties to describe the points of devices/services (e.g., sensor points, communication endpoints) and detailed information of individuals described by the ontology (e.g., name, description, metadata in JSON). Individuals are class members (like instances in object-oriented programming), and an individual can belong to multiple classes. The statement to describe the relationship between two individuals is usually written as "*individualA property individualB*", and can be easily converted into the RDF triple like (individualA, property, individualB).

Certainly, we adhere to the practice of OWL to design the Secure Smart Space Ontology (SSSO) shown in Figure 6.2, in which each box is a top-level class in SSSO. The solid lines represent object properties. Examples of relationships among classes are described by dotted lines. We use the Protege OWL tool [130] to design and validate the ontology that we propose in this paper. The SSSO consists of five top-level classes, namely *Service Class*, *Equipment Class*, *User Class*, *Policy Class*, and

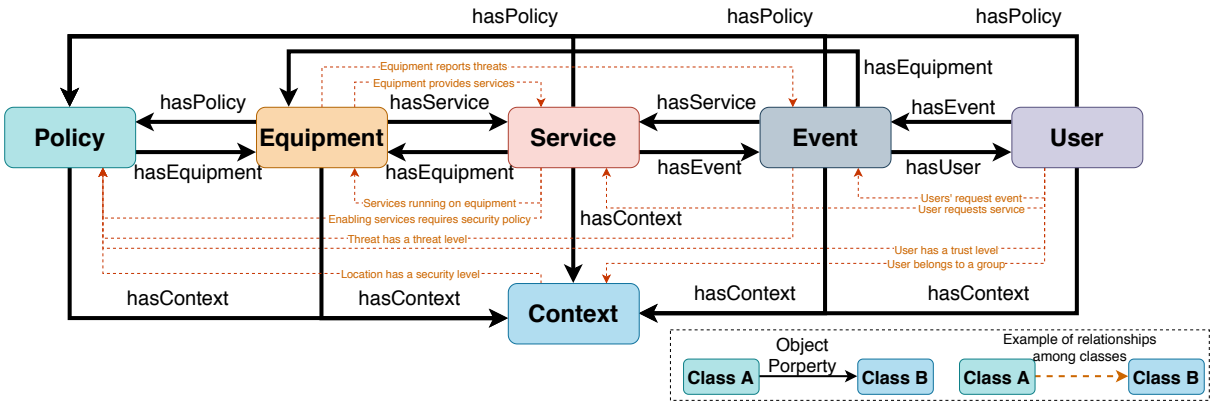


Figure 6.2: The overview of Smart Secure Space Ontology (SSSO).

Context Class. Figure 6.3 depicts the class hierarchy of SSSO. Now, we discuss each class in more detail.

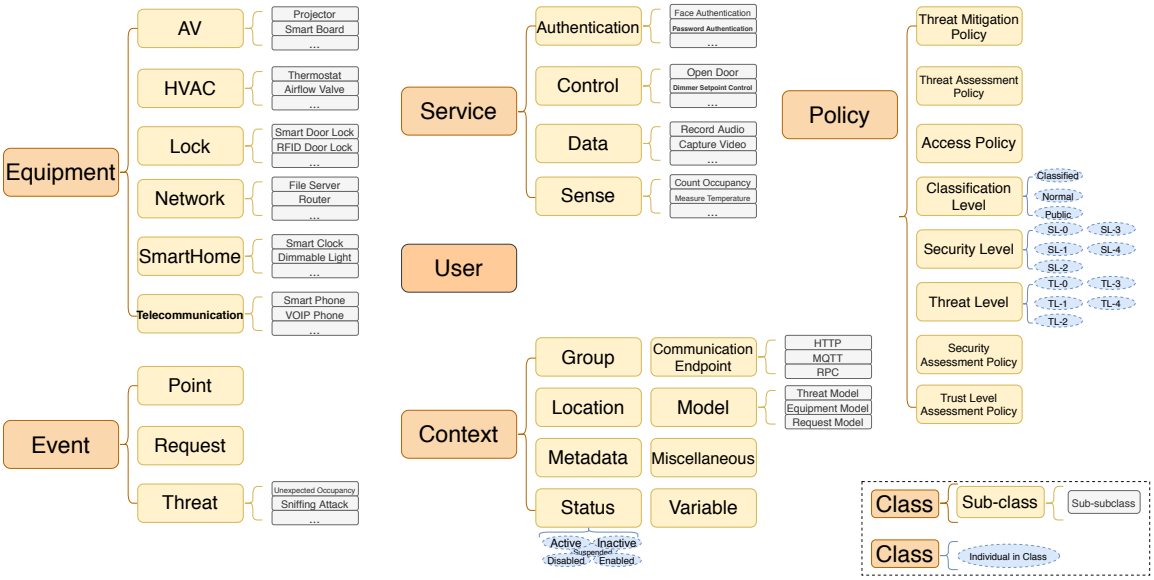


Figure 6.3: The class hierarchy of SSSO.

Service Class: We followed the service-oriented approach and microservice-based IoT architecture [27] where each connected device provides certain services encapsulated as microservices. For example, the smart speaker provides *Play_Video* service (play audio from an external or internal audio source), *Record_Audio* service (record the voice and save the audio file to a local or remote location), and *Voice_Authentication* service (recognize identity by voice). IoT devices in the smart space can

communicate with other components (e.g., other smart devices, control center, and MAPE-k engine) in the space through the publish-subscribe messaging service (e.g., MQTT), application programming interface (e.g., HTTP(s) API request), or remote procedure call (RPC). The interactions among users and smart spaces can be viewed as dynamically enabling and disabling services provided by different devices. For instance, if a meeting attendee wants to use the smart speaker to record the meeting, she can send a request from any available endpoints (e.g., recognized cell phone, control center, the smart speaker itself) to enable the *Record_Audio* service on the smart speaker.

Based on BlackBerry's customer needs, we comprehensively analyze microservices provided by various types of IoT devices deployed in security-critical businesses. By considering the potential risks of services and connections between services and critical data, we categorize services into four sub-classes: *Authentication*, *Control*, *Data*, and *Sense*. *Authentication Class* contains security-critical services used for authentication (e.g., voice authentication, password authentication). *Control Class* describes services used for one-time control. The controlled resources can be security-critical (e.g., open a door) and non-critical (e.g., change the brightness of a dimmable light). Services regarding continuous critical data collection and access, such as voice recording and video capturing, are classified as *Data Class*. *Sense Class* contains services that can provide critical or non-critical contextual information, such as occupancy count and environmental temperature. Such services are typically offered by sensors in the smart space. The four sub-classes mentioned above also have their own sub-classes, which refer to specific types of service.

Equipment Class: It belongs to devices under control in the smart space. Following the same method used for designing *Service Class*, we categorize devices into six sub-classes. *AV Class* contains equipment relevant to multi-media content, including audio and video. *HVAC Class* describes devices of HVAC systems. *Lock Class* consists of critical devices for physical access control, such as smart door locks

and RFID key readers. *Network Class* contains equipment responsible for accessing the Internet or the Intranet or storing and distributing digital content, such as file servers and routers. *Telecommunication Class* describes telecommunication devices, such as VOIP phones and smartphones. *SmartHome Class* contains smart devices that do not fall into the above categories and do not provide any services or access data that are security-critical. Examples are some non-critical and auxiliary devices, such as dimmable lights and smart sweepers. Similarly, each sub-class in *Equipment Class* has its own sub-classes, which refer to specific types of equipment.

User Class: It consists of users in the smart space. Each user is an individual (class instance) of *User Class*.

Policy Class: It refers to the adaptive security-relevant policy implemented in the security-enhanced smart space. *Policy Class* has eight sub-classes. We define three classification levels in *Classification_Level Class*, namely *Classified*, *Normal*, and *Public*, to describe the classification level of instances in ontology, such as locations, groups, and services. We define five levels of security, from SL-0 to SL-4 in *Security_-Level Class* to measure the security of the current environment and the trust level of users. Similarly, five levels of threat, from TL-0 to TL-4, are defined in *Threat_Level Class* to indicate how a threat compromises the security of the current environment. Generally, to ensure security, enabling service and keeping it running require both the security level of the smart space and the trust level of the service requester are equal to or greater than a certain threshold depending on services and context. A threat may degrade the security level of the environment depending on its severity and context, and may have a countermeasure described in *Threat_Mitigation_Policy Class*.

For better adaptability, the required security level for enabling a service and the threat level of a threat can be dynamic depending on the context of the smart space. Besides, system managers may implement access policies to achieve dynamic access control, such as role-based, group-based, and context-based access. Therefore, we

define *Security_Assessment_Policy Class* to store security policies used to assess the required security level of services. Similarly, *Threat_Assessment_Policy Class* consists of policies to assess the severity of threats. User-defined access policies belong to *Access_Policy Class*. Details about security/trust/threat levels and adaptive policies are discussed further in Section 6.4.

Context Class: It consists of contextual information that can help describe individuals of other classes more comprehensively. It is extendable so that ontology users can define any additional information of entities using this class. *Context Class* also acts as a knowledge base of the MAPE-k engine proposed in Section 6.4. Currently, there are eight sub-classes in *Context Class*. *Location Class* describes the location of individuals, such as the room to which a device/user belongs. *Group Class* groups individuals, such as user groups and equipment groups. *Metadata Class* is for storing metadata of individuals, such as the metadata of a service. We define five status indicators, *Active*, *Inactive*, *Suspended*, *Disabled*, and *Enabled*, in *Status Class* to describe the status of individuals, such as the active status of a service. *Variable Class* is for defining environmental variables of the system built on the ontology. *Model Class* has *Threat_Model Class*, *Equipment_Model Class*, and *Request_Model Class* as its sub-classes, used for storing the template of threats, devices, and requests, which are discussed further in Section 6.4. *Communication_Endpoint Class* describes the communication endpoint of services. The systems/services can communicate with each other through protocol and endpoint address described by *Communication_Endpoint Class*. Based on communication protocols, *Communication_Endpoint Class* has three sub-classes, namely *HTTP*, *MQTT*, and *RPC*. Individuals that do not fall into the above classes belong to *Miscellaneous Class*. Ontology users can define any additional information of entities in it.

Event Class: The event-driven architecture is an effective solution to the software system, which is loosely coupled and highly distributed [131]. As the microservice-based system typically has the same features, we introduce *Event Class* in ontology

to describe events in the smart space. *Event Class* has three sub-classes. *Point Class* refers to events regarding the value or status change of monitored endpoints. *Request Class* describes users' requests. *Threat Class* refers to the threat reported by equipment or the anomaly in the space. Each sub-class in *Threat Class* represents a specific type of threat/anomaly.

Object and Data Properties: Properties that map the relationships developed among individuals and represent values are indispensable for describing resources, policies, and context in the smart space. For instance, the security team wants to enforce an access policy that only users belonging to the *Facility Manager* group can access services provided by HVAC devices deployed in the building *BLD-A*. Object properties are required to build connections among services, devices, users, policies, and context, including locations and groups. Data properties are also needed to provide the value of endpoints and the content of policies. Explicitly, we define the six object properties and six data properties in SSSO. Figure 6.2 illustrates relationships among classes and object properties defined.

For each top-level class in SSSO, we define an object property representing an individual having a relationship with an individual in that class. The six object properties are *hasService*, *hasEquipment*, *hasUser*, *hasPolicy*, *hasContext*, and *hasEvent*, respectively. For example, the statement that group *Facility Manager* has a user *Alice* can be written as "*Alice hasContext Facility_Manager*" or "*Facility_Manager hasUser Alice*". Besides, *hasName*, *hasDescription*, *hasMetadata*, *hasClass*, *hasValue*, and *hasData* are defined as six data properties. They are responsible for giving detailed information about an individual in terms of its name, description, metadata, relationship with a class, point value, and data content, respectively. For instance, an HTTP communication endpoint *edp1* with an address *http://10.1.2.3/svc/record* can be written as "*edp1 hasData "http://10.1.2.3/svc/record"^ rdfs:Literal*".

6.3.2 Features of Secure Smart Space Ontology

Overall, the proposed ontology has four features: service-oriented, security-enhanced, event-driven, and context-rich. Now, we discuss each feature in more detail.

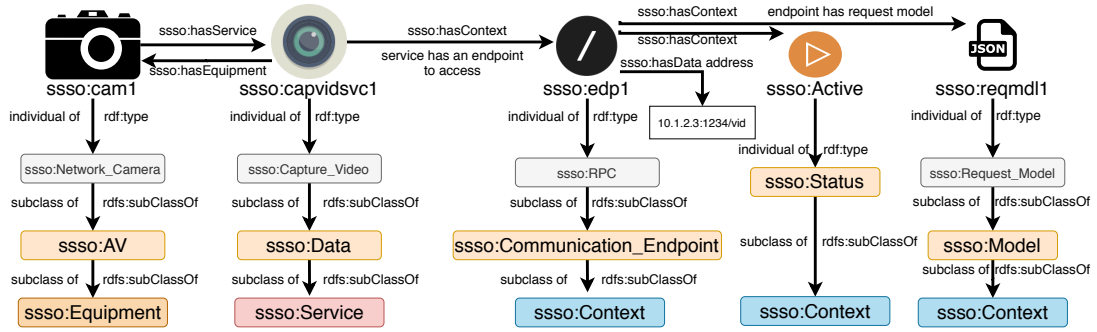


Figure 6.4: The service-oriented feature.

Service-oriented: We follow the ideas of microservice-based IoT architecture, where each device provides one or more services encapsulated as microservices. Each service has at least one communication endpoint with a specific protocol for service access. In the smart space backed by microservices, devices perform functions through services, users enable and disable specific services to make the best use of the space, services also detect and report status, events, and anomalies, and policies and preferences should be imposed on services to ensure security and achieve autonomic management. With the efforts considering the service-oriented system, in SSSO, *Service Class* is the core, and services in the smart space can be efficiently and comprehensively described through individuals in other classes and object and data properties. Figure 6.4 demonstrates the service-oriented feature and gives an example of a capture video service described by SSSO.

As shown in Figure 6.4, the service named "capvidsvc1" is a capture video service provided by the network camera named "cam1". The service has an "Active' status, indicating that it is currently enabled and running. The service has a communication endpoint with an address specified by a data property for controlling the service

through RPC. The communication endpoint also has a request model describing the metadata of the payload in the request. The system built on the ontology can follow the request model to interact with such a service. Therefore, the class hierarchy and object and data properties of SSSO can effectively describe the necessary information of services in smart spaces backed by devices with the microservice architecture. Figure 6.5 gives another example of describing details about the service regarding policies and more contextual information.

Security-enhanced: We introduce *Policy Class* and *Context Class* into SSSO for security enhancements. These two classes describe the access policy, security/trust/threat levels and assessment policies, threat mitigation policy, and contextual information regarding security. To prevent unauthorized activities, safeguard sensitive data, and take countermeasures against threats, these security-relevant attributes can be used to impose security control. Specifically, all activities in the smart space should conform to specific security policies and satisfy several conditions described by the ontology. For instance, enabling a service requires a certain security level and trust level. A threat has a certain threat level, which could degrade the current security level in the environment. Figure 6.5 demonstrates an example of using SSSO to describe the security policy regarding a capture video service hosted by a network camera. In this case Figure 6.5, enabling and maintaining the service *capvidsvc1* provided by *cam1* requires that both the security level of the environment *Room1* and the trust level of the service requester *Alice* satisfy the level of SL-4. In Section 6.4, we discuss the security-enhanced feature of SSSO and security policies in more detail.

Event-driven: The event-driven architecture (EDA) is an effective solution to the software system, which is loosely coupled and highly distributed [131]. As the microservice-based IoT system typically has the same features, EDA can also be applied to it. The event in EDA is a significant state change which the system should process and respond to. Similarly, in the microservice-based system, anything that

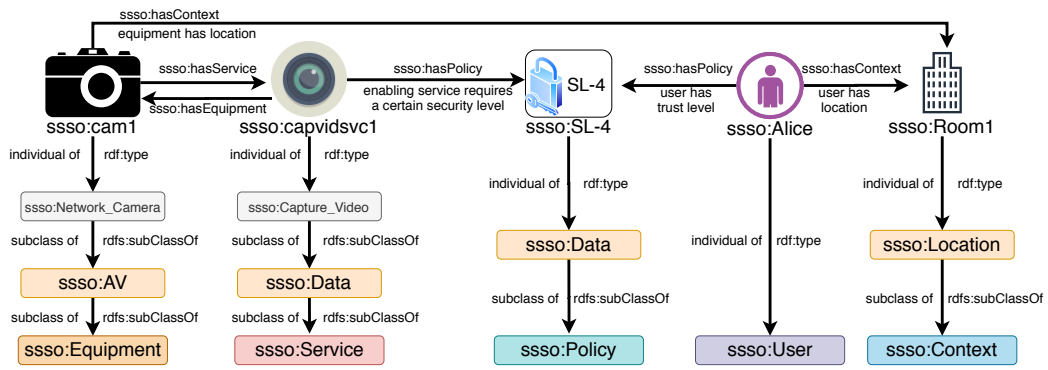


Figure 6.5: The security-enhanced feature.

happens in the system which changes the state or is going to change the state is the event. The event in EDA is a significant state change which the system should process, respond reasonably, and return the result. For instance, the user requesting to enable a service shown in Figure 6.5 is a request event. After receiving such a request event, the security manager built on SSSO adds such an event to *Request Class*, analyzes the situation, namely checking if all required security policies are met in this case. If satisfied, the system plans to enable the service and collect endpoint information described by SSSO. Finally, the system follows the address, protocol, request model to enable the service, and modify the status of the event and service described in SSSO. The request event triggers such a process, and enabling service is the result of the event. SSSO can well meet the needs of resources and information descriptions at all stages in the entire event processing logic.

Context-rich: SSSO explicitly has *Context Class* that describes contextual information that can help describe individuals of other classes more comprehensively. It is extendable so that developers can define any additional information of entities using this class, thus making the ontology scalable and compatible with other smart space scenarios with different granularity levels, such as a smart home and smart buildings. The rich context can also act as the knowledge base of the event processor, which would be discussed in Section 6.4.

6.4 Autonomic Security Manager

In this section, we follow the MAPE-k method and propose an autonomic security manager for IoT smart spaces built on top of resources described by SSSO. The manager maintains the security of smart spaces adaptively and can be encapsulated as a service running in the environment backed by microservice-based devices.

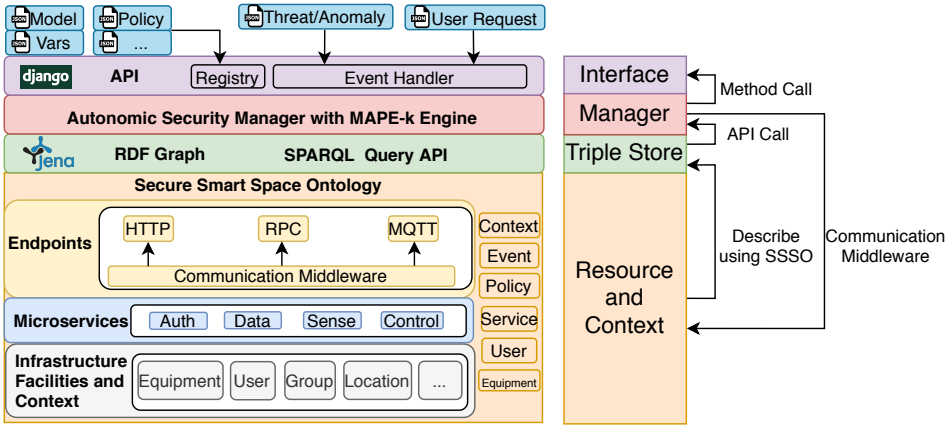


Figure 6.6: Overview of the architecture of the proposed autonomic security manager.

6.4.1 Overall Architecture

The overview of the architecture of the proposed autonomic security manager is shown in Figure 6.6, where interactions between layers are shown on the right. As shown in Figure 6.6, there are four layers in the system, namely *Resource and Context*, *Triple Store*, *Manager*, and *Interface*, marked in different colors. Now, we elaborate on four layers in the system.

The first layer is *Resource and Context*, which represents the physical infrastructure, facilities, and context in smart spaces. Devices provide functionalities through services encapsulated as microservices hosted on them. Each service exposes a communication endpoint through which the service can be accessed, controlled, and configured. For time series data provided by services in *Sense Class*, the endpoints keep the records of their address and protocol for ease of retrieval by other components in

the system. Using the SSSO, all resources and contextual information in this layer are converted into the RDF graph expressed as a collection of RDF triples.

Triple Store layer is responsible for storing those RDF triples generated in *Resource and Context* layer. We leverage Apache Jena to develop this layer. Apache Jena is an open-source semantic web framework with various components that can provide high-performance RDF storage and query services [132]. To fetch the knowledge expressed by the RDF data model, SPARQL was used to query the RDF graph [133]. Human-readable constraints and patterns of triples can be defined in SPARQL queries, and the RDF graph will be traversed to find the match. Hence, we deploy this layer as a container and exposes an API for querying the RDF graph using SPARQL.

Manager layer is where the MAPE-k method is implemented to achieve autonomic security management of resources in smart spaces. The autonomic security manager running in this layer monitors/receives events, analyzes situations, and autonomically learns and implements appropriate actions. During such a process, the manager queries/updates the resource and context RDF graph through the query API provided by the *Triple Store* layer. If needed, the manager can also utilize the communication middleware embedded in it, and follow the address, protocol, and request model stored in RDF triples to communicate with endpoints. Details about the implemented MAPE-k method and examples of manager's interactions are discussed in Section 6.4.4. *Manager* layer is encapsulated as a Python package and is convenient for developing components upon it.

Interface layer is a Django-based API encapsulated as a containerized service built on the *Manager* layer. It is acting as an interface to communicate with the autonomic manager, and responsible for exposing the methods provided by *Manager* layer in the form of API, and receiving requests and returning results in JSON. In this way, security management becomes a service in the smart space, and other resources in the space can interact with it as if they interact with other services. *Interface* layer has two main components, namely registry and event handler, which are detailed in

Section 6.4.2

6.4.2 Equipment Model and Device Registration

The registry in *Interface* layer is for registering new devices/services, and modifying policies/context in the SSSO RDF graph. While adding a new device, either an equipment model that describes the metadata about the device, or the UUID of the equipment model already stored in the RDF Graph is required. The equipment model contains the information including name, version, UUID, description, class the device belongs to, provided services, communication endpoints together with protocols, addresses, and request models, threats the device may report, and applicable security policies. Figure 6.7 illustrates the schema of the equipment model JSON format, and gives an example of describing a type of smart board using the equipment model. As in practice, it is common to deploy a large number of devices of the same model. Using the equipment model to describe devices can ease the workload of register and manage devices. Besides registering devices, the equipment model can provide important information to facilitate system development in the smart space.

When the event to register a new device received by the interface passes to the manager, the first step is to parse the equipment model from the input and store it in *Equipment_Model Class*. If the UUID of an existent equipment model is provided, the system will retrieve the model from the RDF graph through the SPARQL query. The manager creates an individual in the equipment class specified by the model with a newly generated UUID as its IRI in the ontology. For future reference, the UUID of an individual is the same as its IRI, and we use them interchangeably in this paper. Then, for each service defined in the equipment model, the manager inserts an individual in the corresponding service class, and adds detailed information such as endpoints and provided contextual information in corresponding classes defined by SSSO. Object and data properties are used to link newly-added individuals. The procedure is similar for adding information regarding threats specified in

the model. Figure 6.4 and Figure 6.5 together give a simple example about individuals and properties added in the graph when a new device is registered. Following a similar idea, when the registry receives the event to register/modify policies, users, and any other contextual information, the manager leverages the SPARQL query to add/delete/modify individuals and properties in the SSSO RDF graph.

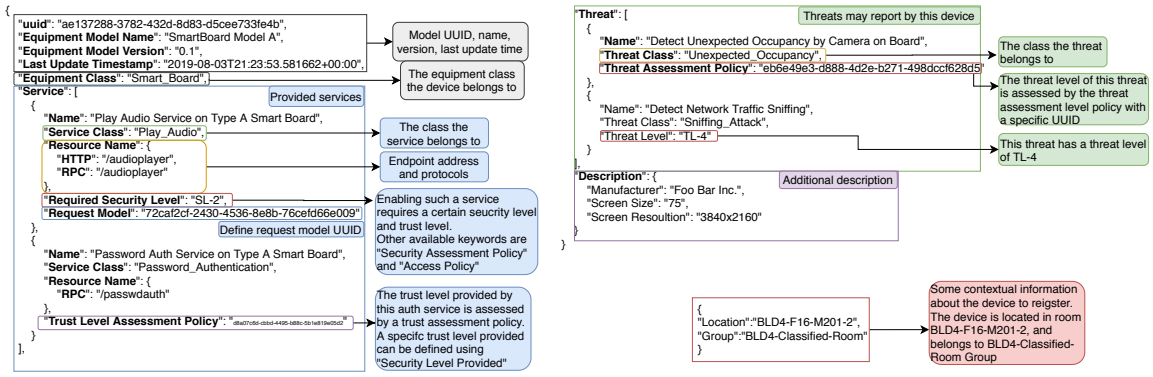


Figure 6.7: An example of an equipment model of a smart board.

6.4.3 Adaptive Security Policy

The security manager and SSSO support both deterministic and adaptive policies. As shown in Figure 6.3 and Figure 6.7, there are seven types of security policies applicable to services and threats in SSSO. In this section, we discuss each type of security policy and elaborate on the schema defining a security policy.

For safely accessing services, the most straightforward approach is to define a specific required security level using individuals in *Security_Level Class*. One example is the service demonstrated in Figure 6.5. However, a fixed security level may not be a good option, especially when the space manager desires to impose dynamic access control based on context. The same problem holds for the threat level of threats. We introduce adaptive policies in SSSO and the autonomic security manager to enable adaptability to various contexts. The adaptive policy is written in SPARQL, which can be evaluated by the manager and return a dynamic required security level/ posed threat level depending on the information stored in the SSSO RDF graph. Figure 6.8

```
OPTIONAL {
    ?loc rdf:type ssso:Location.
    ?cl rdf:type ssso:Classification_Level.
    ?eq ssso:hasService ?Service.
    ?eq ssso:hasContext ?loc.
    ?loc ssso:hasPolicy ?cl.
    FILTER (?cl NOT IN (ssso:Classified)). }.
BIND (IF(BOUND(?cl), ssso:SL-1, ssso:SL-2) as ?Security_Level)
```

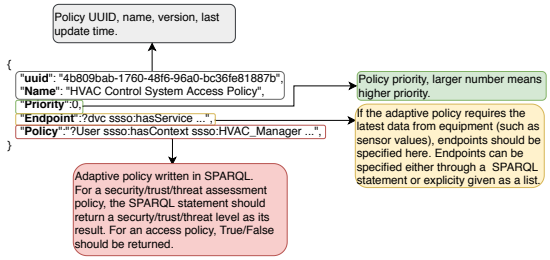
Listing 1: Example of a security assessment policy. The required security level for enabling services depends on the classification level of the room where the equipment is located. If the classification level is "Classified", the security level of SL-2 is required. Otherwise, SL-1 is needed. *?Service* is a reserved variable that has the UUID of the requesting service as its pre-defined value. *?User* is pre-defined as the service requester UUID. The evaluation result binds to the reserved keyword *?Security_Level*.

(a) depicts the schema of the adaptive policy. For the sake of performance, real-time time series data is not synchronously updated in the SSSO graph. As the adaptive policy may rely on the real-time data provided by services running on equipment (usually sensors), we introduce an optional attribute named "Endpoint" in the schema to let the manager first retrieve the latest data of the endpoint following the protocol, address, and request model stored in the SSSO graph, and use "hasValue" data property to insert/refresh the data of the endpoint. The UUID (IRI) of endpoints to refresh can be specified by a SPARQL statement or given explicitly as a list. After refreshing the value of endpoints, the security manager evaluates the statement defined in the "Policy" attribute, obtains the dynamic result as security/trust/threat level, and continues to process the event.

The table shown in Figure 6.8 (b) presents the rules for writing adaptive policy statements. For each policy type, the table presents the reserved keywords in the statement, the variable to which the result binds to, and the expected result. The reserved keywords are the name of SPARQL variables that have pre-defined values or to which the result binds. Listing 1 and Listing 2 give examples of a security assessment policy and an access policy, respectively.

```
?sl rdf:type ssso:Security_Level.
?eq ssso:hasService ?Service.
?eq rdf:type ?class.
?class rdfs:subClassOf* ssso:HVAC.
?User ssso:hasPolicy ?sl.
FILTER (?sl IN (ssso:SL-1, ssso:SL-2, ssso:SL-3, ssso:SL-4)).
?User ssso:hasContext ssso:HVAC_Manager.
```

Listing 2: Example of an access policy. Only authenticated users in the HVAC manager group can access services provided by HVAC devices. The evaluation result of the access policy is True/False, representing having access or not. No result bindings are needed.



(a) The schema of adaptive policies

Type	Reserved Keywords	Evaluation Result
Security Assessment Policy	?Security_Level, ?User, ?Service	?Security_Level binds to a security level.
Trust Assessment Policy	?Security_Level, ?User, ?Service	?Trust_Level binds to a security level.
Threat Assessment Policy	?Threat_Level, ?Equipment, ?Threat_Class	?Threat_Level binds to a threat level.
Access Policy	?User, ?Service	True or False
Threat Mitigation Policy	?Suspend, ?Disable, ?Enable, ?Threat_Level, ?Equipment, ?Threat_Class, ?Env	?Suspend binds to services to suspend. ?Disable binds to services to disable. ?Enable binds to services to enable ?Threat_Level binds to a threat level
Endpoint SPARQL Statement	?Endpoint, ?User, ?Service, ?Equipment, ?Threat_Class	?Endpoint binds to endpoints

(b) Rules for writing adaptive policy statements

Figure 6.8: (a) The schema of adaptive policies and its description. (b) Guidelines for writing adaptive policy statements.

6.4.4 MAPE-k Method for Autonomic Security Management

In this subsection, we discuss each phase of the MAPE-k method for autonomic security management in smart spaces.

Monitor Phase: We follow the microservice architecture pattern and assume that the IoT system has the ability to sense changes through various microservices and report anomalies to the autonomic security manager through communication protocols. In this case, there is no need to consider the low-level monitoring part. We could view the manager as an event-driven engine, as what has been discussed in Section 6.3. Once the anomaly or status change is reported to the autonomic manager, the even processing loop will be triggered.

Analyze Phase: Once the event is reported to the manager, the manager will analyze the event. We consider two types of events, the user's request and the threat.

Now we discuss the two cases separately.

When a user requests for enabling a service on a certain device, the endpoint will report such a request that contains the requested service UUID and requester UUID to the manager through the interface. After receiving the request, the manager first analyzes the situation to determine if the service can be enabled and running in a secure environment by querying the SSSO RDF graph using SPARQL. As each step shown in the grey box in Figure 6.9 (a), the analysis contains the four processes, which are detailed as follows: 1) Get the environmental context regarding the requested service. Precisely, the location or the group of the service, the classification level of the location or the group, and the trust level of the requester are acquired. 2) Obtain the security policy that applies to the service. If multiple security policies are applicable, the manager chooses the one with the highest priority. If the service does not have any security policy applicable to it, a default security assessment policy will be applied. For the sake of security, the default policy stipulates that services in *Data Class* and *Control Class* provided by equipment in *AV Class*, *Lock Class*, *Network Class*, and *Telecommunication Class* require a security level of SL-4; otherwise, a security level of SL-2 is required. 3) Evaluate the applicable adaptive security policy. For a security assessment policy, the evaluation result is the required security level. For an access policy, the request will be directly accepted if the evaluation returns True. If the service has a fixed required security level, the manager skips this step. 4) Check whether the condition satisfies the secure access equation defined in Equation (6.1), and plan appropriate actions in the following phase depending on the equation evaluation result. If the user authenticates through service in *Authentication Class*, the process is similar, but the manager obtains and evaluates the corresponding trust assessment policy, and gives the user a trust level by modifying the SSSO RDF graph through SPARQL.

In the event of a device reporting a threat, the manager generally goes through five steps, as shown in the grey box in Figure 6.9 (b). 1) Get the environmental context

regarding the threat. Specifically, the location or the group of the device, the current security level, the class to which the threat belongs, and the active threats and services in the location. 2) Obtain the threat policy that applies to the threat. Similarly, if multiple policies are available, the manager chooses the one with the highest priority. If the threat does not have any policy applicable to it, a default threat assessment policy will be applied. To ensure security, any threat reported by a device belonging to *AV Class*, *Lock Class*, *Network Class*, and *Telecommunication Class*, will pose a threat level of TL-4. Otherwise, the threat level of TL-2 will be applied to the threat. 3) Obtain the threat mitigation policy if applicable. If there is a threat mitigation policy, the manager will suspend/disable/enable services as stipulated by the policy as countermeasures. 4) Evaluate the applicable adaptive threat policy. For a threat assessment policy, the evaluation result is the posed threat level. If the threat has a fixed threat level, or the mitigation policy already returns the threat level after applying countermeasures, the manager skips this step. 5) Calculate the new security level of the environment(location or group) using Equation (6.2) and Equation (6.3). Then, the manager adds the threat to the RDF graph and re-evaluates all active services running in the environment by applying the same steps used in the analyzing phase for requests to enable them. All active services that fail to satisfy its security policy or the secure access equation will be suspended to avoid information disclosure and ensure security. If a device reports that a threat with a specific UUID is no longer active or has been resolved, the manager will remove the threat, update the security level, and re-evaluate all suspended services in the environment following the same steps. If the security level has improved due to the removed threat, the suspended threats may be resumed.

$$\begin{cases} \text{Trust Level} \geq \text{Security Level} \\ \text{Security Level} \geq \text{Required Security Level} - \text{CL Tuning Param} \end{cases} \tag{6.1}$$

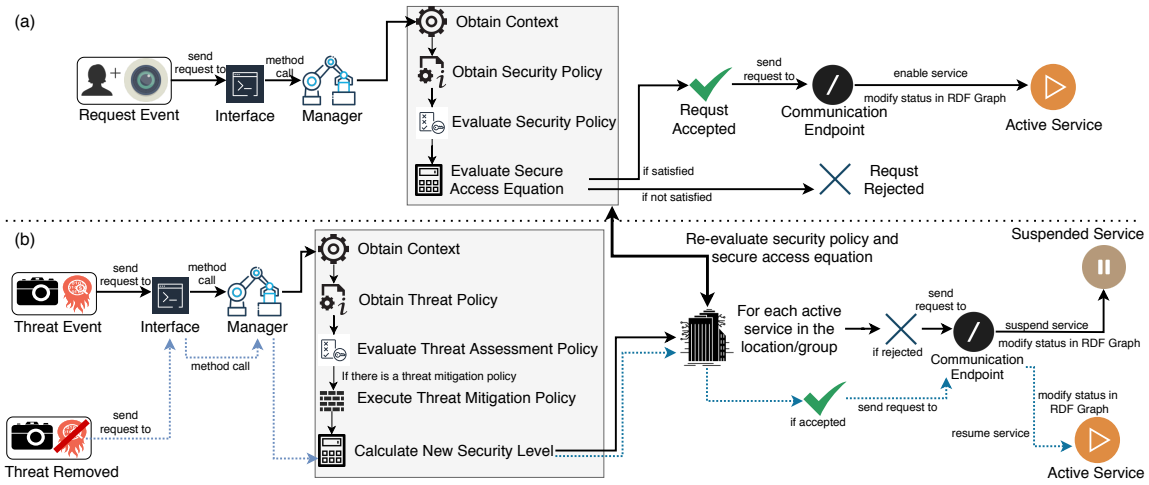


Figure 6.9: (a) The process of handling a user request event. (b) The process of handling an active/resolved threat.

where

$$\text{CL Tuning Param} = \begin{cases} 0, \text{if Classification Level is Classified} \\ 2, \text{if Classification Level is Normal} \\ 4, \text{if Classification Level is Public} \end{cases}$$

$$New\ Security\ Level = Current\ Security\ Level - max(Threat\ level\ of\ active\ threats) \tag{6.2}$$

besides, while calculating the new security level, the manager also follows

$$\begin{cases} \text{Four } TL-1 \text{ threats in different classes are equivalent to one } TL-2 \text{ threat} \\ \text{Three } TL-2 \text{ threats in different classes are equivalent to one } TL-3 \text{ threat} \\ \text{Two } TL-3 \text{ threats in different classes are equivalent to one } TL-4 \text{ threat} \\ New\ Security\ Level = 0, \text{if Current Security Level} \\ \qquad\qquad - max(Threat\ level\ of\ active\ threats) < 0 \end{cases} \tag{6.3}$$

Plan Phase: During the plan phase, the manager plans appropriate actions to be executed for maintaining security in the execute phase, depending on the results of the analyze phase, and minimizes manual work. As shown in Figure 6.9, after analyzing the request and the situation, the manager needs to change the status

of the service and update the SSSO RDF graph if the user's request is accepted. Therefore, the manager needs to retrieve relevant triples, including the status of the requested service, communication endpoint, protocol, and request model from the SSSO RDF graph, and prepare commands to enable the service through the request and update the graph for the execute phase. In the event of handling a new threat, if the security level has degraded, the manager has to re-evaluate the security policy for each active service in the current environment. In case there are any services with an unmet condition, the manager plans to suspend these services that may indeed pose security risks. The manager has to evaluate relevant SPARQL quires to retrieve relevant triples from the graph and prepare commands to suspend services. Similarly, such plan phase procedures are also applicable to a threat removal event. In that case, the threat will be removed, and the manager plans to resume previously suspended service automatically if the security level has improved.

To further reduce manual work, the manager will reason about the root cause of the failed attempt to enable service and try to resolve it, in case that it rejects a user's request due to any violated security policies or the unsatisfied secure access equation in the analyze phase. Suppose the trust level of the requester is under the threshold. In that case, the manager will calculate the additional trust level the user needs, obtain the context, find available authentication services in the environment which can provide such an amount of trust level through SPARQL queries, and prompt the user for performing authentication using a particular authentication service on a specific device. If there are multiple choices, the manager will randomly choose one of them. For the sake of safety, relying on a single authentication method is not recommended. Hence, the introduction of randomness could also reduce the likelihood of system breaching. During this process, the engine will also ignore the authentication services that have been used by this user to avoid duplicated authentication methods and ensure security. On the other hand, suppose the failed attempt is due to the insufficient security level of the space. The engine will try to execute any applica-

ble countermeasures defined by threat mitigation policies to mitigate threats. If not applicable, the manager will suggest the user disable the device that incurs threats. The security level of the space may improve, and consequently, the request may be accepted.

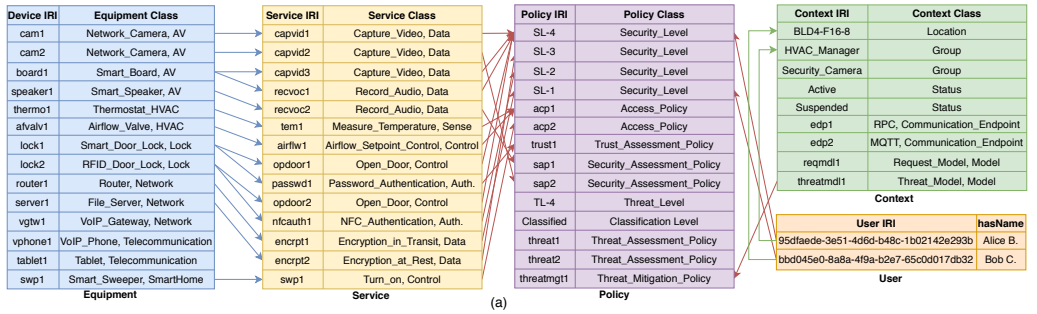
Execute Phase: During the execute phase, the manager executes commands prepared in the plan phase. For the system with the microservice architecture, the execute phase can be simplified. We implement a communication middleware in the manager so that it can send requests to controlled services and executes the planned actions following the communication protocol, endpoint address, and request payload obtained in the plan phase. Besides, the manager executes SPARQL queries and updates the SSSO RDF graph.

Knowledge Base: The whole RDF graph based on SSSO acts as the knowledge base of the autonomic security manager. The RDF graph can be queried and updated using SPARQL through the triple store API, representing retrieving information and updating the knowledge base.

6.5 Implementation and Evaluation

We use Python 3.8.2 to implement the autonomic security manager and the interface layer. For the triple store layer, we leverage Apache Jena with the Fuseki component. All three implemented layers are containerized using Docker and can be deployed as services in the microservice-based smart space. The containerized autonomic security manager can be scaled horizontally and vertically to meet the demands of large-scale deployments and serve a large volume of requests. As the evaluation, based on a current BlackBerry customer problem, we model a smart conference room with 32 devices, 66 services, 30 potential threats, and 28 adaptive policies using SSSO, and deploy the implemented autonomic security manager. The partial overview of the modeled space is shown in Figure 6.10 (a).

Based on BlackBerry and the customer's input, we design a series of 160 events,



Device IRI	Equipment Class
cam1	Network_Camera, AV
cam2	Network_Camera, AV
board1	Smart_Board, AV
speaker1	Smart_Speaker, AV
thermo1	Thermostat_HVAC
afvalv1	Airflow_Valve, HVAC
lock1	Smart_Door_Lock, Lock
lock2	RFID_Door_Lock, Lock
router1	Router, Network
server1	File_Server, Network
vgtw1	VoIP_Gateway, Network
vphone1	VoIP_Phone, Telecommunication
tablet1	Tablet, Telecommunication
swp1	Smart_Sweeper, SmartHome

Equipment

Service IRI	Service Class
capvid1	Capture_Video, Data
capvid2	Capture_Video, Data
capvid3	Capture_Video, Data
recvoc1	Record_Audio, Data
recvoc2	Record_Audio, Data
tem1	Measure_Temperature, Sense
airflw1	Airflow_Setpoint_Control, Control
opdoor1	Open_Door, Control
passwd1	Password_Authentication, Auth.
opdoor2	Open_Door, Control
nfcauth1	NFC_Authentication, Auth.
encrpt1	Encryption_in_Transit, Data
encrpt2	Encryption_at_Rest, Data
swp1	Turn_on, Control

Service

Policy IRI	Policy Class
SL-4	Security_Level
SL-3	Security_Level
SL-2	Security_Level
SL-1	Security_Level
acp1	Access_Policy
acp2	Access_Policy
trust1	Trust_Assessment_Policy
sap1	Security_Assessment_Policy
sap2	Security_Assessment_Policy
TL-4	Threat_Level
Classified	Classification Level
threat1	Threat_Assessment_Policy
threat2	Threat_Assessment_Policy
threatmgt1	Threat_Mitigation_Policy

Policy

Context IRI	Context Class
BLD4-F16-8	Location
HVAC_Manager	Group
Security_Camera	Group
Active	Status
Suspended	Status
edp1	RPC, Communication_Endpoint
edp2	MQTT, Communication_Endpoint
reqmdl1	Request_Model, Model
threatmdl1	Threat_Model, Model

Context

User IRI	hasName
95dfaede-3e51-4d6d-b48c-1b02142e293b	Alice B.
bbd045e0-8a8a-4f9a-b2e7-65c0d017db32	Bob C.

User

(a)

Event ID	Event Description	Event Type	Response
1	Register a new device "Smart Board".	Register Device	1 device, 9 services, 11 endpoints, 1 equipment model... added
2	Register a new device "Smart Speaker".	Register Device	1 device, 3 services, 3 endpoints, 1 equipment model... added
3	System administrator initializes a conference room and modifes the classification level of the room BLD4-F16-8 as "Classified".	Modify Policy	BLD4-F16-8 hasPolicy Classified, BLD4-F16-8 hasPolicy SL-4
4	Alice authenticates through the password authentication service porvided by the smart door lock.	User Request	Alice hasPolicy SL-2. Alice has the trust level of SL-2.
5	Alice requests to enable the capture video service provided by the smart board.	User Request	Request rejected as the trust level is insufficient. Prompt for Face Auth on Smart Board.
6	Alice authenticates through the face authentication service provided by the smart board.	User Request	Alice hasPolicy SL-4. Alice has the trust level of SL-4.
7	Alice requests to enable the capture video service provided by the smart board.	User Request	Request accepted. Enable the capture video service capvid1.
7	The network camera in BLD4-F16-8 reports the unexpected occupancy threat.	New Threat	BLD4-F16-8 hasPolicy SL-0. The security level degrades to SL-0. capvid1 service is suspended.
...	...	...	...

(b)

Figure 6.10: (a) The partial overview of the modeled security-enhanced smart conference room. (b) The partial description of the series of 160 events.

as shown in Figure 6.10 (b). The full information, including individuals, properties, policy definitions, descriptions, and event responses, is available on our artifact page. The series of events cover various event types. We validate the autonomic security management through such an event series. The autonomic security manager can adaptively respond to the events and maintain the security of the smart space through the MAPE-k method proposed in Section 6.4.4. The responses of the manager are partially presented in Figure 6.10 (b). We also evaluate the performance of the proposed solution for managing security on a large-scale deployment with 20,000 devices, 180,000 services, 260,002 contextual entities, and in total, 1,860,701 triples. We deploy the system on a laptop with a 2.50GHz Intel Core i5-2520M processor and 8 GB of memory without scaling any layer. Without considering the communication delay for sending requests to the endpoint, the autonomic security manager can respond to new device registration, request handling, and threat handling within two seconds on average. This shows the applicability of the proposed system for a large-scale smart space. The manager is applicable to other smart space scenarios with different granularity levels, such as a smart home and smart buildings.

6.6 Summary

In this chapter, we addressed Object 4 by proposing the Secure Smart Space Ontology and autonomic security manager. The ontology is service-oriented, security-enhanced, event-driven, and context-rich. It is extendable and adheres to the principles of the IoT system with microservice architecture. Based on SSSO, we proposed an autonomic security manager with the MAPE-k method, which could autonomically manage the security while minimizing manual work. We evaluated the proposed solution through a case study on a current BlackBerry customer problem and assessed the performance of the system under a large-scale deployment with over 1.8 million triples. The manager could adaptively respond to all events and autonomically manage the security of the space.

Chapter 7

Discussion, Future Work, and Conclusion

7.1 Discussion

As serverless computing and microservices have become some of the most popular use cases and examples of distributed systems, optimal function container placement has become an urgent problem. While container orchestration solutions usually provide various scheduling policies, most of them leverage simple strategies that treat the serverless function as a black box, inevitably resulting in performance degradation. To solve this issue, we propose an ANN-based adaptive function placement algorithm in Chapter 3 that takes both serverless functions and VMs into consideration. Our work could shed some light on optimal function placement for FaaS platforms. In this way, we point out a promising direction to optimize microservice-based distributed systems. Our results show that the throughput of functions could be improved with negligible overhead by considering the workload characteristics of serverless functions when scheduling function containers. While the proposed function placement algorithm outperforms other strategies according to the evaluation results, it also has some limitations: 1) In our setting, all VM configurations are identical. 2) The overhead is tested on relatively small scales. 3) The model for predicting normalized throughput does not evolve over time.

With the new development, deployment, management, testing, and billing models

introduced by FaaS, applications based on serverless microservices have emerged as a new type of cloud-based and event-driven distributed systems. While FaaS providers take over most operational responsibilities, the unpredictability of performance and cost and trade-offs between them have become significant concerns. In Chapter 4, we propose, to the best of our knowledge, the first analytical models to accurately get the average end-to-end response time and cost of serverless applications with four types of structures in their workflows. In Chapter 5, we present a heuristic algorithm for optimizing the performance and cost of serverless applications, which can help developers solve trade-offs between performance and cost. Both the proposed models and optimization algorithms are evaluated through real experiments on AWS, which involves multiple serverless applications with complex structures.

While the proposed models have a high time complexity, we validate the applicability of the models for now and the foreseeable future by an empirical study and analysis under the worst-case scenario. The proposed models and optimization algorithms can be updated by new monitoring data depending on the degree of uncertainty in the underlying FaaS infrastructure and the serverless application itself. However, if the initial performance profiling phase has been done properly, the performance and cost models and optimal configurations should remain valid for a long time. Other reasons for updating the model will be changing the function configurations, the application architecture, function source code, or optimization constraints. From a practical point of view, these parameters are not expected to change frequently. If any of the parameters mentioned above, except function source code, changes, users only need to rerun performance and cost models and optimization algorithms, which can be completed easily in a short time. If the function source code changes, updating the performance profile is required for the modified function, which is comparatively time-consuming and perhaps costly. Based on current input, the proposed models can only predict the average response time and cost, which may be less important than the 95th percentile in some cases. However, to model the 95th percentile, we need the exact distribution

of the input size and the response time of each function. It is beyond the scope of the study presented in this thesis to model percentiles and deviations.

In Chapter 6, we aim at solving the security issue of distributed systems, specifically microservice-based IoT systems. We propose SSSO with four features to describe and abstract smart spaces equipped with microservice-based IoT devices. Then, we use SSSO as the knowledge base in the MAPE-k loop engine to achieve autonomic security management in IoT smart spaces. We present a case study on a smart conference room with 32 devices and 66 services and evaluate the proposed manager using a series of 160 events. However, several limitations have been identified: (1) We use Python to simulate behaviors of IoT devices, and we do not use the actual equipment to implement such an autonomic security management system. Since the microservice-based IoT devices can be treated as a piece of software, we do not expect any major impact on our results. (2) We only use a series of events to evaluate our security solution and do not evaluate the ontology in regular IoT systems. Leveraging IoT security testbeds is an effective method for solution evaluations [134]. However, there is no available IoT testbed suitable for microservice architecture systems at the moment. (3) We design a default security assessment policy and a default threat assessment policy based on the relationships among services, devices, and critical data. Such default policies may not be the best practices in some scenarios in which the underlying system architecture, space granularity, or on-premise security guidelines differ considerably. In such a case, an on-site security professional may be advised to optimize default security and threat assessment policies.

7.2 Future Work

The studies of this thesis suggest several avenues for future research.

1. It is valuable to study the adaptive function placement algorithm for large-scale heterogeneous clouds composed of different flavors of VMs and PMs. Efficient

and scalable online machine learning techniques could be leveraged to update the function performance prediction model for enhanced predictive accuracy.

2. It is practically and industrially important to investigate the integration of the adaptive function placement algorithm into the Kubernetes control plane as a pluggable scheduler.

3. The performance and cost models and optimization algorithms with better efficiency and time complexity may be explored in a future study.

4. Besides average response time and cost, it is significant to model other performance SLAs that involve percentiles, such as the 95th percentile of response time and cost.

5. It is helpful to automate the process of profiling serverless functions, extracting the workflow of serverless applications, and predicting the performance and cost. A toolchain consisting of such functionalities would be valuable to serverless application developers.

6. It is useful to implement and analyze the proposed autonomic security manager in more real-world scenarios and develop an IoT testbed suitable for microservice architecture systems.

7.3 Conclusion

In this thesis, we addressed four research objectives by presenting the solution to optimize microservice-based distributed systems from three aspects. More specifically, we focused on improving SLA adherence of FaaS platforms, performance and cost modeling and optimization for applications based on serverless microservices, and autonomic security management for smart spaces based on IoT systems with microservice architecture.

In Chapter 3, we improved SLA compliance by proposing and solving the function placement problem. We proposed an ANN-based placement algorithm, which could select the VM from the VM pool leading to the best predictive performance to place the function container by considering the workload profile of functions and performance metrics of VMs. We conducted extensive experiments to evaluate the implemented function placement algorithm and compare it with three common placement strategies. Our evaluation results showed that the proposed placement algorithm could improve the throughput of serverless functions without incurring significant overhead.

In Chapter 4 and Chapter 5, we addressed the problem of modeling and optimization of performance and cost for serverless applications. In Chapter 4, we first laid out a formal definition of the serverless workflow considering several serverless features on clouds. Then we solved the unpredictable performance and cost problems by proposing the performance and cost models. The proposed models could accurately estimate the average end-to-end response time and cost of serverless applications. We checked the validity of the proposed models by extensive evaluation of five serverless applications deployed on AWS. In Chapter 5, we formulated two optimization problems, namely the best performance under the budget constraint and the best cost under the performance constraint. We answered them by proposing a heuristic algorithm with four greedy strategies. Again, we verified the validity of the proposed algorithm through experimental evaluations on AWS.

In Chapter 6, we achieved autonomic security management for smart spaces equipped with microservice-based IoT devices. We first proposed the Secure Smart Space Ontology for describing resources and context in security-enhanced smart spaces, which is service-oriented, security-enhanced, event-driven, and context-rich. Such a formal description of the IoT environments facilitates analysis and reasoning about the current state of the space in a machine-understandable fashion. We used SSSO as the knowledge base in the MAPE-k loop engine to achieve autonomic security manage-

ment in IoT smart spaces. We implemented an autonomic security manager that has four layers with scalability. The autonomic manager could monitor and analyze events and context, and plan and execute adaptive countermeasures with minimum human intervention at a large scale. Based on the current BlackBerry customer problem, we modeled a smart conference room and evaluated our work through a series of events. The performance of the proposed solution was also assessed through a large-scale deployment.

Bibliography

[1] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing serverless computing: Introducing an adaptive function placement algorithm," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 203–213.

[2] 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, 2021.

[3] C. Lin, H. Khazaei, A. Walenstein, and A. Malton, "Autonomic security management for iot smart spaces," *ACM Transactions on Internet of Things*, vol. 2, no. 4, pp. 1–19, 2021.

[4] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-hall, 2007.

[5] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. pearson education, 2005.

[6] *Challenges with distributed systems*, https://aws.amazon.com/builders-library/challenges-with-distributed-systems/, [Online; accessed January-16-2021], 2019.

[7] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[8] H. Huang, J. Lin, B. Zheng, Z. Zheng, and J. Bian, "When blockchain meets distributed file systems: An overview, challenges, and open issues," *IEEE Access*, vol. 8, pp. 50 574–50 586, 2020.

[9] F. Yang and A. A. Chien, "Large-scale and extreme-scale computing with stranded green power: Opportunities and costs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1103–1116, 2017.

[10] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Access control for emerging distributed systems," *Computer*, vol. 51, no. 10, pp. 100–103, 2018.

[11] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and Others, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, Springer, 2017, pp. 1–20.

[12] A. Fox *et al.*, "Above the clouds: A berkeley view of cloud computing," 2009.

[13] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2016, pp. 261–268.

[14] *Docker: Empowering app development for developers*, https://www.docker.com/, [Online; accessed May-11-2021], 2021.

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

[16] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018.

[17] 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.

[18] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.

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

[20] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, IEEE, 2017, pp. 847–852.

[21] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, IEEE, 2015, pp. 171–172.

[22] H. Zhu and I. Bayley, "If docker is the answer, what is the question?" In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2018, pp. 152–163.

[23] *Docker hub quickstart*, https://docs.docker.com/docker-hub/, [Online; accessed June-8-2021], 2021.

[24] H. Fingler, A. Akshintala, and C. J. Rossbach, "Usetl: Unikernels for serverless extract transform and load why should you settle for less?" In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 23–30.

[25] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 923–935.

[26] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2016, pp. 1–6.

[27] D. Lu, D. Huang, A. Walenstein, and D. Medhi, "A secure microservice framework for iot," in *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2017, pp. 9–18.

[28] C. Lin, S. Nadi, and H. Khazaei, "A large-scale data set and an empirical study of docker images hosted on docker hub," in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020.

[29] *Release: Aws lambda on 2014-11-13*, https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/, [Online; accessed February-2-2020], 2014.

[30] *Aws lambda limits*, https://docs.aws.amazon.com/lambda/latest/dg/limits.html, [Online; accessed February-2-2020], 2020.

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

[32] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, e4792, 2018.

[33] *New–provisioned concurrency for lambda functions*, https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/, [Online; accessed November-12-2020], 2019.

[34] *Aws lambda pricing*, https://aws.amazon.com/lambda/pricing/, [Online; accessed November-24-2020], 2020.

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

[36] *Create a serverless workflow with aws step functions and aws lambda*, https://aws.amazon.com/getting-started/tutorials/create-a-serverless-workflow-step-functions-lambda/, [Online; accessed February-2-2020], 2020.

[37] *Serverless reference architecture: Image recognition and processing backend*, https://github.com/aws-samples/lambda-refarch-imagerecognition, [Online; accessed January-8-2021], 2016.

[38] *Ad hoc big data processing made simple with serverless mapreduce*, https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/, [Online; accessed January-8-2021], 2016.

[39] *Serverless application lens aws well-architected framework*, https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf, [Online; accessed January-8-2021], 2019.

[40] *Aws step functions*, https://aws.amazon.com/step-functions/, [Online; accessed February-24-2021], 2020.

[41] A. Eivy, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[42] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[43] E. Van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The spec cloud group's research vision on faas and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pp. 1–4.

[44] I. Lee and K. Lee, "The internet of things (iot): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, 2015.

[45] B. Russell and D. Van Duren, *Practical internet of things security*. Packt Publishing Ltd, 2016.

[46] *Artifact repository: Pacslab/smartspread*, https://github.com/pacslab/smartspread/, [Online; accessed April-20-2021], 2021.

[47] *Artifact repository: Pacslab/slapp-perfcost-mdlopt/*, https://github.com/pacslab/SLApp-PerfCost-MdlOpt/, [Online; accessed April-20-2021], 2021.

[48] *Artifact repository: Pacslab/ssso-autosecmng/*, https://github.com/pacslab/SSSO-AutoSecMng/, [Online; accessed April-20-2021], 2021.

[49] S. Ginzburg and M. J. Freedman, "Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, 2020, pp. 43–48.

[50] A. Pérez, S. Risco, D. M. Naranjo, M. Caballer, and G. Moltó, "On-premises serverless computing for event-driven data processing applications," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019, pp. 414–421.

[51] Á. L. García, J. M. De Lucas, M. Antonacci, W. Zu Castell, M. David, M. Hardt, L. L. Iglesias, G. Moltó, M. Plociennik, V. Tran, *et al.*, "A cloud-based framework for machine learning workloads and applications," *IEEE Access*, vol. 8, pp. 18 681–18 692, 2020.

[52] S. Ghaemi, H. Khazaei, and P. Musilek, "Chainfaas: An open blockchain-based serverless platform," *IEEE Access*, vol. 8, pp. 131 760–131 778, 2020.

[53] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 273–283.

[54] A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, and T. Dauphinee, "Migrating from monolithic to serverless: A fintech case study," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 20–25.

[55] F. Samea, F. Azam, M. W. Anwar, M. Khan, and M. Rashid, "A uml profile for multi-cloud service configuration (umlpmsc) in event-driven serverless applications," in *Proceedings of the 2019 8th International Conference on Software and Computer Applications*, 2019, pp. 431–435.

[56] V. Lenarduzzi and A. Panichella, "Serverless testing: Tool vendors' and experts' points of view," *IEEE Software*, vol. 38, no. 1, pp. 54–60, 2020.

[57] K. Djemame, M. Parker, and D. Datsev, "Open-source Serverless Architectures: An Evaluation of Apache OpenWhisk," *Proceedings - 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC 2020*, pp. 329–335, 2020.

[58] S. K. Mohanty, G. Premsankar, and M. Di Francesco, "An evaluation of open source serverless computing frameworks," *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 2018-December, pp. 115–120, 2018.

[59] D. Balla, M. Maliosz, and C. Simon, "Open Source FaaS Performance Aspects," *2020 43rd International Conference on Telecommunications and Signal Processing, TSP 2020*, pp. 358–364, 2020.

[60] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 154–160.

[61] 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*, 2019, pp. 1063–1075.

[62] 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.

[63] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," *Middleware 2020 - Proceedings of the 2020 21st International Middleware Conference*, pp. 356–370, 2020.

[64] B. Yang, F. Tan, Y.-S. Dai, and S. Guo, "Performance evaluation of cloud service considering fault recovery," in *IEEE International Conference on Cloud Computing*, Springer, 2009, pp. 571–576.

[65] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 5, pp. 936–943, 2011.

[66] H. Khazaei, J. Mišić, V. B. Mišić, and N. B. Mohammadi, "Availability analysis of cloud computing centers," in *2012 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2012, pp. 1957–1962.

[67] X. Li, S. Liu, L. Pan, Y. Shi, and X. Meng, "Performance analysis of service clouds serving composite service application jobs," in *2018 IEEE International Conference on Web Services (ICWS)*, IEEE, 2018, pp. 227–234.

[68] J. Vilaplana, F. Solsona, I. Teixidó, J. Mateo, F. Abella, and J. Rius, "A queuing theory model for cloud computing," *The Journal of Supercomputing*, vol. 69, no. 1, pp. 492–507, 2014.

[69] H. Chen, C. Zhou, Y. Qin, A. Vandenberg, A. V. Vasilakos, and N. Xiong, "Petri net modeling of the reconfigurable protocol stack for cloud computing control systems," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 393–400.

[70] M. M. Alansari and B. Bordbar, "Modelling and analysis of migration policies for autonomic management of energy consumption in cloud via petri-nets," in *2014 International Conference on Cloud and Autonomic Computing*, IEEE, 2014, pp. 121–130.

[71] P. Rygielski and S. Kounev, "Data center network throughput analysis using queueing petri nets," in *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2014, pp. 100–105.

[72] Y. Cao, H. Lu, X. Shi, and P. Duan, "Evaluation model of the cloud systems based on queuing petri net," in *International Conference on Algorithms and Architectures for Parallel Processing*, Springer, 2015, pp. 413–423.

[73] C. Rista, M. Teixeira, D. Griebler, and L. G. Fernandes, "Evaluating, estimating, and improving network performance in container-based clouds," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2018, pp. 00 514–00 520.

[74] C.-Z. Xu, J. Rao, and X. Bu, "Url: A unified reinforcement learning approach for autonomic cloud management," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 95–105, 2012.

[75] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 3–14.

[76] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC international conference on performance engineering*, 2015, pp. 145–156.

[77] C. Witt, M. Bux, W. Gusew, and U. Leser, "Predictive performance modeling for distributed batch processing using black box monitoring and machine learning," *Information Systems*, vol. 82, pp. 33–52, 2019.

[78] 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*, 2020, pp. 265–276.

[79] L. Versluis, E. Van Eyk, and A. Iosup, "An analysis of workflow formalisms for workflows with complex non-functional requirements," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 107–112.

[80] N. M. Calcavecchia, O. Biran, E. Hadad, and Y. Moatti, "Vm placement strategies for cloud scenarios," in *2012 IEEE Fifth International Conference on Cloud Computing*, IEEE, 2012, pp. 852–859.

[81] M. R. Chowdhury, M. R. Mahmud, and R. M. Rahman, "Study and performance analysis of various vm placement strategies," in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, IEEE, 2015, pp. 1–6.

[82] A. B. Alam, T. Halabi, A. Haque, and M. Zulkernine, "Multi-objective interdependent vm placement model based on cloud reliability evaluation," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, IEEE, 2020, pp. 1–7.

[83] M. Ghobaei-Arani, A. A. Rahmanian, M. Shamsi, and A. Rasouli-Kenari, "A learning-based approach for virtual machine placement in cloud data centers," *International Journal of Communication Systems*, vol. 31, no. 8, e3537, 2018.

[84] Y. Qin, H. Wang, S. Yi, X. Li, and L. Zhai, "Virtual machine placement based on multi-objective reinforcement learning," *Applied Intelligence*, pp. 1–14, 2020.

[85] A. Khosravi, L. L. Andrew, and R. Buyya, "Dynamic vm placement method for minimizing energy and carbon cost in geographically distributed cloud data centers," *IEEE Transactions on Sustainable Computing*, vol. 2, no. 2, pp. 183–196, 2017.

[86] L. Ismail and H. Materwala, "Energy-aware vm placement and task scheduling in cloud-iot computing: Classification and performance evaluation," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 5166–5176, 2018.

[87] A. Ibrahim, M. Noshy, H. A. Ali, and M. Badawy, "Papso: A power-aware vm placement technique based on particle swarm optimization," *IEEE Access*, vol. 8, pp. 81 747–81 764, 2020.

[88] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Cost-driven scheduling of grid workflows using partial critical paths," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1400–1414, 2011.

[89] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[90] X. Lin and C. Q. Wu, "On scientific workflow scheduling in clouds under budget constraint," in *2013 42nd International Conference on Parallel Processing*, IEEE, 2013, pp. 90–99.

[91] H. R. Faragardi, M. R. S. Sedghpour, S. Fazliahmadi, T. Fahringer, and N. Rasouli, "Grp-heft: A budget-constrained resource provisioning scheme for workflow scheduling in iaas clouds," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[92] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2114–2129, 2019, ISSN: 2161-9883.

[93] D. Lu, Z. Li, and D. Huang, "Platooning as a service of autonomous vehicles," in *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, IEEE, 2017, pp. 1–6.

[94] L. Sun, Y. Li, and R. A. Memon, "An open iot framework based on microservices architecture," *China Communications*, vol. 14, no. 2, 2017.

[95] L. Liu and M. T. Özsu, *Encyclopedia of database systems*. Springer New York, NY, USA: 2009, vol. 6.

[96] O. Lassila and R. R. Swick, "Resource description framework (rdf) model and syntax specification," 1999.

[97] M. Tao, J. Zuo, Z. Liu, A. Castiglione, and F. Palmieri, "Multi-layer cloud architectural model and ontology-based security service framework for iot-based smart homes," *Future Generation Computer Systems*, vol. 78, pp. 1040–1051, 2018.

[98] S. D. Nagowah, H. B. Sta, and B. Gobin-Rahimbux, "An overview of semantic interoperability ontologies and frameworks for iot," in *2018 Sixth International Conference on Enterprise Systems (ES)*, IEEE, 2018, pp. 82–89.

[99] F. Latfi, B. Lefebvre, and C. Descheneaux, "Ontology-based management of the telehealth smart home, dedicated to elderly in loss of cognitive autonomy.," in *OWLED*, vol. 258, 2007.

[100] L. Chen and C. Nugent, "Ontology-based activity recognition in intelligent pervasive environments," *International Journal of Web Information Systems*, vol. 5, no. 4, pp. 410–430, 2009.

[101] A. Evesti, R. Savola, E. Ovaska, and J. Kuusijärvi, "The design, instantiation, and usage of information security measuring ontology," in *The Second International Conference on Models and Ontology-based Design of Protocols, Architectures and Services*, 2011, pp. 1–9.

[102] A. Evesti, J. Suomalainen, and E. Ovaska, "Architecture and knowledge-driven self-adaptive security in smart space," *Computers*, vol. 2, no. 1, 2013.

[103] S. Borgo, A. Cesta, A. Orlandini, and A. Umbrico, "An ontology-based domain representation for plan-based controllers in a reconfigurable manufacturing system," in *The Twenty-Eighth International Flairs Conference*, 2015.

[104] N. Seydoux, K. Drira, N. Hernandez, and T. Monteil, "Autonomy through knowledge: How iot-o supports the management of a connected apartment," in *Semantic Web Technologies for the Internet of Things (SWIT)*, CEUR-WS, 2016.

[105] Y. I. Khan and M. U. Ndubuaku, "Ontology-based automation of security guidelines for smart homes," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, IEEE, 2018, pp. 35–40.

[106] D. G. Korzun, S. I. Balandin, and A. V. Gurtov, "Deployment of smart spaces in internet of things: Overview of the design challenges," in *Internet of Things, Smart Spaces, and Next Generation Networking*, Springer, 2013, pp. 48–59.

[107] S. Balandin and H. Waris, "Key properties in the development of smart spaces," in *International conference on universal access in human-computer interaction*, Springer, 2009, pp. 3–12.

[108] D. Korzun, I. Galov, and S. Balandin, "Development of smart room services on top of smart-m3," in *Open Innovation Association*, IEEE, 2013.

[109] V. Catania and D. Ventura, "An approch for monitoring and smart planning of urban solid waste management using smart-m3 platform," in *Proceedings of 15th conference of open innovations association FRUCT*, IEEE, 2014, pp. 24–31.

[110] D. G. Korzun, A. V. Borodin, I. V. Paramonov, A. M. Vasilyev, and S. I. Balandin, "Smart spaces enabled mobile healthcare services in internet of things environments," *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, vol. 6, no. 1, pp. 1–27, 2015.

[111] H. Moeini, I.-L. Yen, and F. Bastani, "Routing in iot network for dynamic service discovery," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2017, pp. 360–367.

[112] H. Moeini, W. Zeng, I.-L. Yen, and F. Bastani, "Toward data discovery in dynamic smart city applications," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, 2019, pp. 2572–2579.

[113] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, pp. 159–169, 2018.

[114] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, IEEE, 2012, pp. 73–80.

[115] *Cybera – a connected future for all albertans*, https://www.cybera.ca/, [Online; accessed March-16-2021], 2021.

[116] *Docker hub*, https://hub.docker.com/, [Online; accessed March-16-2021], 2021.

[117] A. Kopytov, "Sysbench manual," *MySQL AB*, pp. 2–3, 2012.

[118] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[119] *Best practices for working with aws lambda functions*, https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html, [Online; accessed March-12-2021], 2021.

[120] A. Kumar and S. Mahendrakar, *Serverless Integration Design Patterns with Azure: Build powerful cloud solutions that sustain next-generation products*. Packt Publishing, 2019, ISBN: 9781788390835. [Online]. Available: https://books.google.ca/books?id=KB2IDwAAQBAJ.

[121] *Aws samples*, https://github.com/aws-samples/, [Online; accessed March-1-2021], 2021.

[122] *Azure samples*, https://github.com/Azure-Samples, [Online; accessed March-1-2021], 2021.

[123] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, 2019.

[124] O. Goldreich, *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press, 2010.

[125] J. E. Kelley Jr, "Critical-path planning and scheduling: Mathematical basis," *Operations research*, vol. 9, no. 3, pp. 296–320, 1961.

[126] R. Minerva, A. Biru, and D. Rotondi, "Towards a definition of the internet of things (iot)," *IEEE Internet Initiative*, vol. 1, pp. 1–86, 2015.

[127] S. Misra, M. Maheswaran, and S. Hashmi, *Security challenges and approaches in internet of things*. Springer, 2017.

[128] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, no. 1, pp. 41–50, 2003.

[129] D. L. McGuinness, F. Van Harmelen, *et al.*, "Owl web ontology language overview," *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[130] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen, "The protégé owl plugin: An open development environment for semantic web applications," in *International Semantic Web Conference*, Springer, 2004, pp. 229–243.

[131] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, no. 12, pp. 10–1571, 2006.

[132] A. Jena, *Semantic web framework for java*, 2007.

[133] E. Prud, A. Seaborne, *et al.*, "Sparql query language for rdf," 2006.

[134] O. A. Waraga, M. Bettayeb, Q. Nasir, and M. A. Talib, "Design and implementation of automated iot security testbed," *Computers & Security*, vol. 88, p. 101 648, 2020.