Improving Cloud-Based Systems: Testing and Optimizing System Scalability

Jia Li

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

In recent years, cloud-based systems have gained significant popularity for delivering resilient services across diverse domains, including the Internet of Things, data centers, and e-commerce, where they must handle dynamic workloads and ensure seamless service delivery. However, maintaining scalability and meeting performance objectives under unpredictable conditions remains a persistent challenge.

This thesis addresses this challenge by focusing on verifying and optimizing the scalability of cloud-based systems through stress testing methodologies and self-adaptive techniques. First, a simulation framework is introduced to conduct stress testing by simulating large numbers of devices communicating with cloud applications at high frequencies, while reducing computational overhead through symbolic representations. This approach analyzes system behavior under extreme workloads and at the upper limits of its capacity.

When facing high workloads, the network connecting devices to the cloud applications can become a bottleneck and frequently congested. In the second part of this thesis, to mitigate network congestion, a generative self-adaptation framework leveraging genetic programming is proposed to evolve data forwarding logic in real time, minimizing Service Level Objective (SLO) violations and adaptation costs. Third, to ensure that cloud applications can scale to varying request loads, an auto-scaling framework is developed to dynamically adjust microservice instances in a coordinated manner, ensuring SLO compliance while optimizing resource usage.

By enhancing the scalability of cloud-based systems through these three methods, our research ensures optimal performance, reliable service delivery, and effective resource uti-

lization under varying and unpredictable workloads. The proposed solutions reduce operational costs and improve system autonomy, advancing the state of the art in designing and managing scalable cloud-based systems.

Acknowledgements

I am incredibly grateful to my supervisors, Prof. Shiva Nejati and Prof. Mehrdad Sabetzadeh, for their unwavering support throughout my PhD journey. Their mentorship has been invaluable to every aspect of my academic development. Without their help, I would not have come this far. I have learned from them not only how to conduct research, but also how to grow into an independent researcher.

I would also like to thank the members of my supervisory committee, Prof. Daniel Amyot and Prof. Olga Baysal, for their insightful feedback and thoughtful advice during the various milestones of my doctoral studies. I am also thankful to Prof. Pooyan Jamshidi and Prof. Stéphane Somé for their precious comments and questions on this thesis.

My sincere thanks go to Michael McCallen for supervising me during my internship at Cheetah Networks, and to my colleagues there for their collaboration and support.

I would like to thank my colleagues in the Sedna Research Lab at the University of Ottawa for their support, collaboration, and the many moments of shared growth. I am equally grateful to my friends whose encouragement has helped me through this journey.

Finally, I dedicate this thesis to my family. To my parents, Xin Li and Guirong Tan, thank you for your unconditional love and unwavering support. To our dog, Andy, I'm sorry I couldn't be by your side.

To those whom I love and those who love me, thank you. Because of you, I am who I am.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud-based systems have become a popular approach for delivering modern applications, offering flexibility and resilience through cloud computing technologies. As these systems are increasingly adopted across diverse domains such as the Internet of Things (IoT), data centers, e-commerce, and media streaming, they face growing demands that expose critical scalability challenges. For example, fluctuating workloads, dynamic user demands, and the need for high availability and low latency place significant pressure on the underlying infrastructure to scale efficiently. These challenges highlight the necessity for advanced research to develop effective mechanisms that ensure cloud-based systems can maintain performance and reliability while scaling to meet varying demands.

Figure 1.1: A schematic overview of a cloud-based system.

1.1 Context

Cloud-based systems are systems that adopt cloud computing to build, deploy and run applications by utilizing services and resources across cloud infrastructures [37]. These systems enable flexible and on-demand access to computing resources, facilitating efficient management and seamless delivery of services to a broad range of users.

Figure 1.1 illustrates a schematic view of a cloud-based system. At its core lies the cloud infrastructure, which hosts essential components such as service interfaces, and various platform services that together form the backbone of cloud applications. Cloud applications are software programs or services that run on cloud platforms rather than on local servers or individual devices. Deployed on the cloud infrastructure, cloud applications provide a

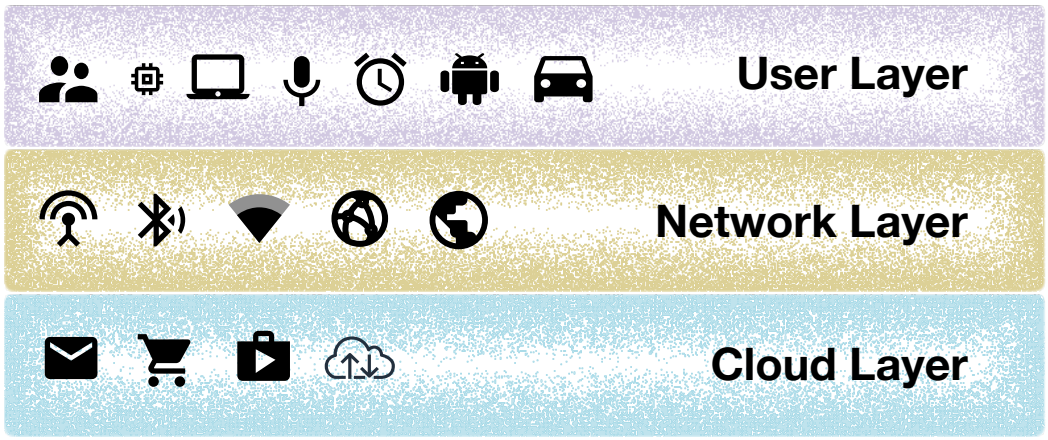

Figure 1.2: Three layers of cloud-based systems.

wide range of services to users. Surrounding this core, users, including both humans and devices, communicate with these services via the network.

Cloud-based systems can be divided into three layers, as shown in Figure 1.2: the user layer, the network layer and the cloud layer. The user layer consists of end users and devices that interact with cloud services. The network layer enables communication between users and the cloud, ensuring the efficient transmission of data. The cloud layer encompasses the cloud infrastructure and applications that host and deliver the services accessed by users.

The interactions among the three layers are fundamental to the operation of cloud-based systems. As the number of users and devices grows, the user layer places increasing demand on cloud services. Simultaneously, the network layer must handle higher volumes of data traffic with low latency and high reliability. At the same time, the cloud layer must dynamically allocate and manage computing resources to meet fluctuating workloads and service-level expectations. These growing demands across all layers introduce significant scalability challenges, including how to efficiently provision resources, maintain service

performance under varying loads, and ensure reliable communication.

Scalability is critical across all three layers of cloud-based systems, as it enables these systems to handle fluctuating workloads by dynamically adjusting services to meet predefined demands. These demands are typically defined through service level objectives (SLOs), which specify measurable performance targets or goals. Common SLOs include metrics such as system uptime, response time, throughput, and error rates. Each layer of cloud-based systems may prioritize different SLOs depending on its performance requirements.

To meet these SLOs, service providers should ensure that their cloud-based systems can scale effectively. Scalability in cloud-based systems is achieved through two main approaches: vertical scaling and horizontal scaling. Vertical scaling enhances the capacity of existing resources, such as CPU, memory, or bandwidth, while horizontal scaling adds more instances or nodes to distribute the workload across multiple resources [154]. Achieving scalability requires designing systems capable of efficiently utilizing resources and dynamically adapting to changing demands. In scenarios involving high-frequency, large-volume requests, ensuring scalability also involves careful resource management to avoid bottlenecks and maintain optimal performance.

To verify the scalability of a cloud-based system, testing is essential to determine whether the system meets SLOs under varying workloads. Stress testing, in particular, is required for analyzing how a system behaves under different workload, including extreme conditions and at the upper limits of its capacity [44].

This thesis addresses the problem of testing and optimizing scalability of cloud-based

systems. It proposes a simulation framework for conducting stress testing that enables evaluation of system performance under varying and extreme workload conditions. This testing framework enables testing and verifying the scalability of cloud-based systems or subsystems with limited computational resources, making it practical and efficient for evaluating different scalability approaches. Building on this foundation, the thesis presents self-adaptive frameworks that continuously monitor system states, learn and update control logic, and dynamically scale systems to meet SLOs in real time aiming to minimize manual intervention and improve the systems' ability to autonomously respond to changing environments. By enhancing the scalability of cloud-based systems through efficient stress testing and self-adaption, this thesis aims to ensure optimal performance, reliable service delivery, and effective resource utilization, even under intense and unpredictable workloads. The proposed solutions help reduce operational costs and improve user satisfaction by maintaining consistent service quality without over-provisioning resources.

1.2 Challenges

Cloud-based systems experience different scalability challenges across the user, network, and cloud layers. The primary goal of this thesis is to verify and enhance their scalability. To achieve this, we examine the scalability-related challenges specific to each layer and propose appropriate solutions.

1.2.1 User Layer

The scalability level of a cloud-based system refers to its ability to efficiently handle growth in workloads, users, or data without significant performance degradation (i.e., SLO violations). Devices at the user layer are typically built with fixed hardware configurations, meaning their CPU, memory, storage capacities are predetermined and non-expandable. These devices are primarily engineered for performance optimization and user experience, such as designing both hardware and software to ensure fast, efficient, and smooth operation. Unlike the cloud layer, which can dynamically allocate resources, user-layer devices operate independently with limited computational capacity, making it difficult to optimize their resource utilization. Given these limitations, scaling individual devices within their existing hardware constraints is inefficient. Instead, when demand increases, a more effective approach is to introduce additional devices or offload computing tasks to the cloud layer rather than enhancing the scalability of each device. This principle is widely applied in domains such as IoT systems and mobile applications. Enhancing scalability for individual devices at the user layer is therefore outside the scope of this thesis.

As cloud-based systems are increasingly deployed across diverse domains, service providers must ensure that their applications and systems can scale effectively to accommodate highly variable and often unpredictable workloads. While individual devices in the user layer are not designed with scalability as a primary goal, they play a critical role in evaluating the scalability of the overall cloud-based system. Stress testing the cloud and network layers requires large numbers of user-layer devices to generate varying and extreme workloads and assess the ability of the system or its components under test to scale dynamically while

maintaining SLOs such as response time and throughput. However, evaluating scalability in real-world environments is often cost-prohibitive and resource-intensive, making simulation a practical and necessary alternative. Existing simulation-based approaches [180, 196] often focus on detailed modeling of user or device behaviors, including intercommunication and energy consumption. While such detailed simulations are valuable for research goals such as optimizing device-level performance and evaluating communication protocol, they require significant computational resources, making them impractical for stress testing cloud-based systems, especially when simulating tens of thousands of devices communicating with cloud applications at high frequencies under limited resources. To overcome these limitations, we consider symbolic representations of devices and offer various flexible settings to reduce resource consumption, enabling the simulation of large numbers of devices for stress testing cloud-based systems.

1.2.2 Network Layer

The scalability of the network layer in cloud-based systems refers to the ability of communication between the user and cloud layers to sustain high workloads without violating SLOs such as network throughput and transit latency. Under high workloads, this communication often suffers from network congestion, leading to delays and packet loss that impact system performance. To maintain SLOs for network throughput and ensure performance under fluctuating workloads, various approaches have been proposed in the literature [9, 28, 89, 136]. Among these solutions, self-adaptation has emerged as a promising approach to address congestion and satisfy SLOs. The goal of self-adaptation is to

enable systems to autonomously respond to dynamic conditions and anomalies during operation, guided by engineers' specifications of desired qualities (i.e., SLOs) and adaptation strategies to achieve them [75].

Although network congestion is a well-studied problem, existing approaches [45, 100, 144, 163] tend to treat each SLO violation as an isolated event, generating an adaptation strategy and applying it to the running system whenever a violation occurs. However, as SLOs remain unmet during the adaptation process, resulting in issues such as packet loss and delay, these approaches can be both costly and ineffective. Therefore, a key challenge in achieving efficient scalability in the network layer of cloud-based systems lies in resolving network congestion in a way that minimizes not only the frequency and duration of SLO violations, but also the need for repeated adaptations. To address this challenge, we propose a generative self-adaptation framework that uses genetic programming to evolve data forwarding logic in real time, effectively resolving network congestion while reducing both SLO violations and the frequency of triggering adaptations.

1.2.3 Cloud Layer

The cloud layer of cloud-based systems encompasses both the cloud infrastructure and the cloud applications responsible for delivering services to users. As cloud applications often operate in dynamic and large-scale environments, they must be designed to handle fluctuating workloads, ensure high availability, and enable rapid updates. Traditional monolithic architectures, however, often struggle to meet these demands due to their tightly coupled components and limited scalability. To overcome these challenges, cloud-based systems are

increasingly adopting cloud-native architectures, which are specifically designed to take full advantage of cloud environments [204]. Cloud-native applications employ technologies such as microservices, containers, and continuous deployment pipelines to enhance scalability, flexibility, and resilience [159]. Unlike monolithic architectures, a cloud-native approach decomposes cloud applications into independent microservices, allowing each service to be deployed, updated, and scaled independently without disrupting the entire system. Figure 1.3 illustrates the key components of a cloud-native architecture. At its core, the figure depicts pods, which encapsulate the execution units of cloud applications. Surrounding the pods are four fundamental building blocks: containers, which provide lightweight and portable runtime environments; infrastructure, which offers the underlying computing resources; microservices, which break down applications into modular and independently deployable units; and APIs, which facilitate communication and integration between services. Together, these elements enable cloud applications to achieve adaptability and operational efficiency, making them well-suited for complex and evolving cloud environments [112].

For cloud-native applications (i.e., cloud applications that follow cloud-native architecture), existing scaling approaches typically adjust the number of microservice instances based on CPU or memory usage [57, 150, 155]. However, the performance of some applications is not accurately reflected by these metrics. For example, applications that rely heavily on GPUs for computation or require high I/O throughput may encounter performance bottlenecks long before CPU or memory usage reaches critical thresholds. As a result, scaling decisions driven solely by CPU or memory usage may be insufficient for applications that are GPU-intensive, I/O intensive or other types of resource-intensive applications that are not constrained by CPU or memory. Besides, existing approaches [155, 173] often focus

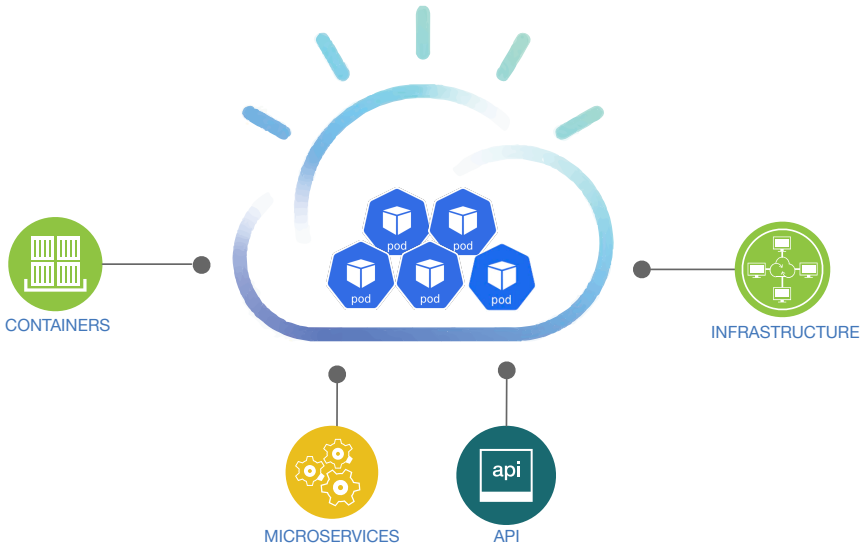

Figure 1.3: Key components of a cloud-native architecture.

on individual microservices without considering the inter-dependencies across the entire application. To effectively meet SLOs and optimize resource utilization, it is crucial to orchestrate the entire application and scale microservices in a coordinated manner. To address this challenge, we propose an auto-scaling framework that combines random search with self-adaptation to dynamically adjust microservice instances in a coordinated way, ensuring SLO compliance while optimizing resource usage in cloud-native applications.

1.3 Research Methodology

To address the challenges discussed in the previous section, I adopt a methodology inspired by the Design Science Research Methodology (DSRM), proposed by Peffers et al. [164], which offers a well-grounded and structured approach for conducting research in infor-

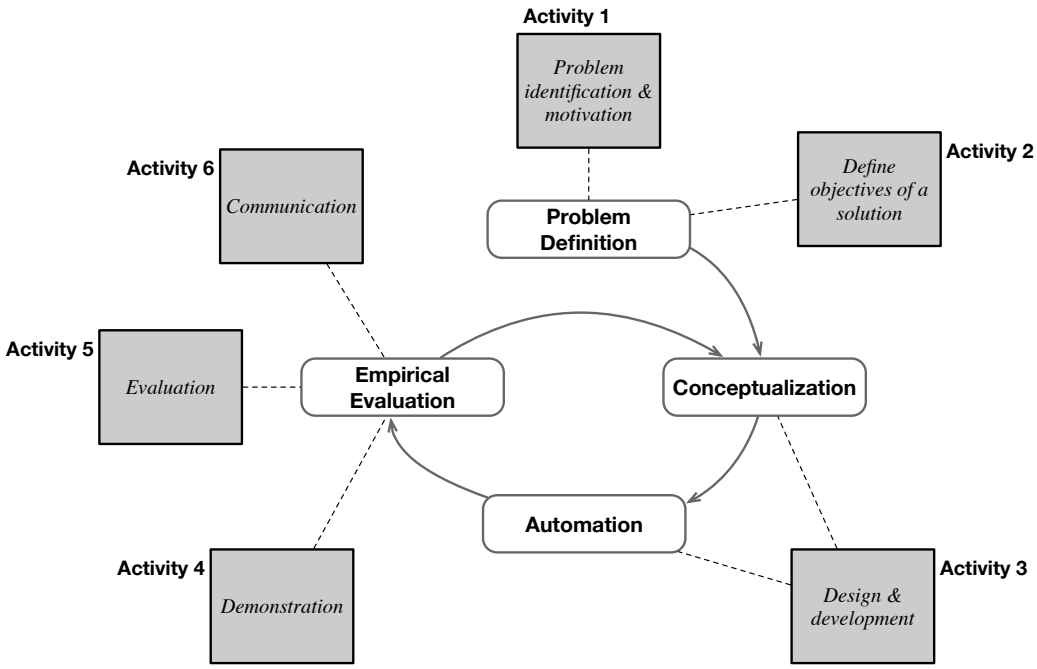

Figure 1.4: Research methodology adopted in this thesis, based on [164] (grey background), to address the challenges discussed in Section 1.2.

mation systems. The methodology include six core activities: (1) problem identification and motivation, (2) definition of the objectives for a solution, (3) design and development, (4) demonstration, (5) evaluation, and (6) communication. As illustrated in Figure 1.4, I consolidate these six activities into four steps to better align with the goals and structure of my research, described as follows:

1.3.1 Problem Definition

The first step in my methodology is to analyze and define the problem in cloud-based systems. This step aims to systematically identify the core challenges, constraints, and

requirements that must be addressed. A clear understanding of the problem space is essential to ensure that the proposed solution effectively tackles real-world scalability issues and aligns with practical needs.

The problems addressed in this thesis emerge from practical issues identified by our research group and reported by our industrial partner, as well as insights gained from the existing literature. The problem definition begins by examining the characteristics and limitations of current cloud-based systems, including a review of typical system architectures and state-of-the-art solutions, to identify existing gaps and challenges that remain unresolved. Based on this analysis, I identify both functional requirements, such as dynamic resource provisioning, and non-functional requirements, including responsiveness, cost-efficiency, and reliability. Finally, this step aims to identify the environmental and operational constraints, such as fluctuating workload patterns and service-level objectives (SLOs), which the adaptive system must adhere to.

This step aligns with *Activity 1: Problem Identification and Motivation*, which involves defining the specific research problem and justifying the value of a solution. By carefully analyzing the problem space, I ensure that the proposed solution captures the complexity of real-world challenges. Besides, *Activity 2: Definition Objectives of a Solution* is addressed by inferring the objectives of a solution from the identified problem and domain knowledge. This step lays the foundation for the subsequent phases of conceptualization, automation, and empirical evaluation.

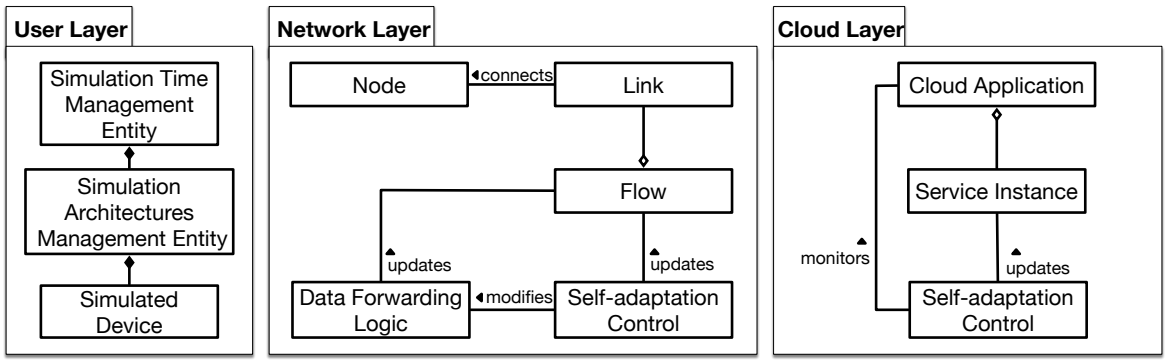

Figure 1.5: Overview of the conceptual models capturing key elements and their relationships in the user, network, and cloud layers of cloud-based systems discussed in this thesis. Detailed information can be found in the conceptual models presented in Chapter 3 and 4.

1.3.2 Conceptualization

The second step in my methodology is the creation of a conceptual model to provide a structured representation of the problem scope, focusing on the system or subsystem targeted for scalability and the necessary entities to address the problem. A conceptual model offers a high-level perspective by representing concepts (i.e., entities) and the relationships among them [130]. It serves as a basis for the subsequent development and implementation of the solutions.

This step corresponds to the design phase of *Activity 3: Design and Development*, where the objective is to design an artifact (i.e., any designed object in which a research contribution is embedded in the design) that can address the identified challenges. The conceptual model serves as a foundational artifact that defines the system's architecture and desired functionality, ensuring that the model captures the essential components necessary for addressing scalability challenges.

The process begins with identifying all necessary entities within the system and the solution. Careful consideration is given to each entity to ensure that all relevant components are included, forming a view of the system's structure. Once the entities are identified, they are organized into layers or packages based on their functionality. Figure 1.5 shows a brief conceptual model summarizing the scope of this thesis. For example, Chapter 4 focuses on the Network Layer, which includes network infrastructure entities such as Nodes and Links, along with Flows. It also covers the Data Forwarding Logic and the Self-adaptation Control, which manages the Flow to enable auto-scaling. Next, the relationships between these entities are established, defining how they interact and depend on one another. Finally, attributes and behaviors are assigned to each entity, specifying their key properties and actions.

This step helps me organize the problem and solution space, ensuring that all related entities and their relationships are defined. It also guides the automation and evaluation processes by providing a structured blueprint that can be transformed into executable models.

1.3.3 Automation

Building on the conceptual model, the third step in my methodology focuses on enabling automation to translate the abstract design into an executable solution. This step corresponds to the development phase of *Activity 3: Design and Development*, where the goal is to create an artifact capable of addressing the identified challenges. The proposed solution is integrated into the system to create a self-adaptive system that can dynamically ad-

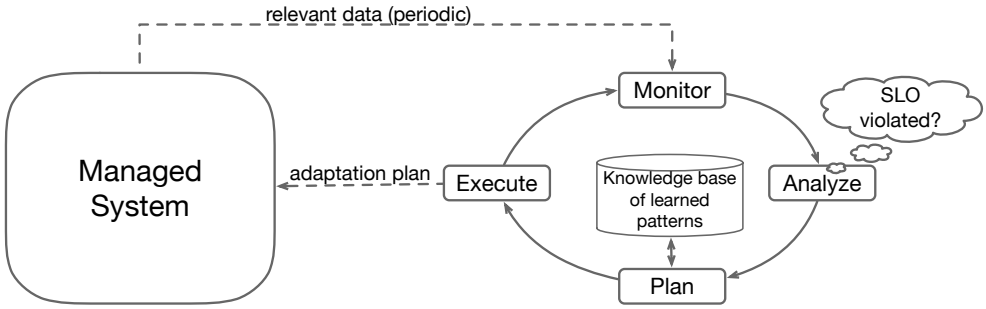

Figure 1.6: Overview of the MAPE-K based self-adaptation framework adopted in this thesis.

just its behavior in response to changes in workload, resource availability, or performance requirements, thereby improving scalability.

As shown in Chapter 4 and Chapter 5, I use the well-known Monitor-Analyze-Plan-Execute over a Knowledge base (MAPE-K) adaptation control loop [105, 144] to drive the self-adaptation process. This control loop, shown in Figure 1.6, consists of four steps:

1. **Monitoring:** Continuously observe the system and its environment to collect relevant data, such as workload, resource usage, and system performance.

2. **Analyzing:** Analyze the collected data to determine whether adaptation is necessary. This involves comparing the system's current state, particularly metrics related to service-level objectives (SLOs), against user-defined SLOs. If a violation is detected, the planning phase is triggered.

3. **Planning:** Generate adaptation strategies to address detected SLO violations. Various techniques, including random search, genetic programming, and machine learning methods such as random forests, are employed to identify optimal adaptation deci-

sions tailored to the system's current state and requirements.

4. **Executing:** Apply the adaptation plan generated in the previous step to the running system to resolve the SLO violation.

In addition to enabling self-adaptation, this phase also involves automatic code generation based on the conceptual model. As shown in Chapter 3, once system attributes and behavioral parameters are defined in a user specification, executable code is automatically generated to implement the solution. This approach ensures consistency between the design and deployment phases while reducing manual effort and minimizing the risk of implementation errors and the requirement of domain knowledge.

1.3.4 Empirical Evaluation

The final step in my methodology is the empirical evaluation of the proposed approaches. This phase assesses the effectiveness of these approaches using a combination of simulated and industrial cloud-based systems as case studies. These systems serve as practical examples to demonstrate how the proposed approaches can be applied in real-world scenarios and to explore the impact of different attribute settings on system performance.

This step encompasses *Activity 4: Demonstration*, where the goal is to showcase the use of the artifact to solve real-world instances of the problem. Demonstration involves applying the proposed solutions in various environments, including experimentation, simulation, and case studies, to assess their ability to dynamically respond to changing workload conditions and meet service-level objectives (SLOs). For example, in Chapter 3, I perform

stress testing on an IoT network monitoring system and an IoT connected-vehicle system. In Chapter 4, I evaluate the proposed approach on one industrial network and 26 synthetic networks. In Chapter 5, I assess the scalability achieved by the proposed method on an online shopping cloud-native application and a Large Language Model (LLM)-based chatbot application.

Following the demonstration, this phase involves *Activity 5: Evaluation*, where I systematically observe and measure how well the proposed solution addresses the identified problems. Evaluation is conducted by comparing the objectives defined in *Activity 2: Define Objectives of a Solution* with actual observed results. This phase uses various metrics to assess the artifact's effectiveness, including:

- **Quantitative Performance Metrics:** Metrics such as response time, throughput, and resource utilization are used to assess how well the system maintains SLOs under varying workloads.
- **Qualitative Assessment:** Insights are gathered from simulated and real-world applications, ensuring that the system effectively meets both functional and non-functional requirements. Feedback collected in the context of practical scenarios and actual usages of our proposed solutions is also incorporated to assess its applicability, usability, and impact in operational environments.

The proposed approaches are also compared against baseline techniques, including state-of-the-art methods from the literature and widely adopted industrial practices. This comparative analysis provides us with an understanding of the advantages and limitations of the proposed approaches.

In addition, this phase addresses *Activity 6: Communication*, which involves effectively communicating the problem, the proposed solution, and its utility and novelty to both researchers and practitioners. The design of the proposed approach, its empirical effectiveness, and its impact on enhancing scalability in cloud-based systems are emphasized to ensure the findings reach relevant audiences and contribute to the advancement of knowledge in the field.

The insights gained from this phase iteratively feed back into *Step 2: Conceptualization*, ensuring that the methodology remains responsive to evolving system complexities and practical needs. In particular, findings in this step prompt updates to the conceptual model and are reflected in subsequent publications. This feedback loop refines the understanding of system challenges and guides further refinements of both the problem scope and the proposed solutions, enhancing the overall effectiveness of the methodology.

1.4 Contributions

This section presents the key contributions of this thesis, including the research questions addressed and the research papers.

1.4.1 Research Questions

In this thesis, I address the challenges outlined in Section 1.2, which are related to the scalability of several cloud-based systems, including network systems, Internet of Things (IoT) systems, online shopping systems and Large Language Model (LLM)-based systems.

Specifically, I investigate the following three high-level research questions:

RQ1: Can we effectively simulate large numbers of devices in the user layer for stress testing other layers of cloud-based systems?

Within the scope of RQ1, I make the following contributions:

Con1.1 I propose a simulation framework which enables efficient simulation of large numbers of devices in the user layer that communicate with applications in the cloud layer using limited resources (e.g., a laptop), specifically for stress testing cloud-based systems in IoT domain.

Con1.2 To facilitate simulation construction for practitioners, I develop a domain specific language (DSL) based on Con1.1, to generate simulators from model-based specifications.

These contributions were published as a conference paper [120] at the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS 2022), then extended into a journal article [117] in IEEE Transactions on Software Engineering (TSE), and are presented in Chapter 3.

RQ2: How can network congestion be efficiently addressed while avoiding future congestion?

I make the following contributions towards answering RQ2:

Con2.1 I propose a generative self-adaptation framework, named GenAdapt, that uses genetic programming (GP) to adapt the control logic of the running system in such

a way that the system itself would learn how to steer clear of future SLO violations, without triggering self-adaptation too frequently.

Con2.2 I apply GenAdapt to address network congestion of Software-Defined Networks (SDNs). SDN is widely used in the network layer of cloud-based systems, particularly in modern data centers and IoT systems, due to its centralized, programmable control over distributed network resources [103]. My approach continuously learns and updates SDN data-forwarding logic to address network congestion while reducing the frequency of adaptation interventions over time.

These contributions were published as a conference paper [118] at the 17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2022), then extended into a journal article [119] in ACM Transactions on Autonomous and Adaptive Systems (TAAS), and are presented in Chapter 4.

RQ3: How can cloud-native applications in the cloud layer be effectively scaled to maintain SLOs under dynamic workload changes?

In this thesis, I make the following contribution related to RQ3:

Con3.1 I propose an auto-scaling framework for cloud-native applications in the cloud layer of cloud-based systems. It uses random search to dynamically generate scaling decisions based on the current environment and workload. This framework ensures compliance with various SLOs, while optimizing computational resources usage by adjusting the number of microservice instances in real time.

This contribution is presented in Chapter 5.

1.4.2 Research Papers

The contributions of this thesis are derived from my prior publications and a draft research paper, as outlined below.

- Jia Li, Behrad Moeini, Shiva Nejati, Mehrdad Sabetzadeh, and Michael McCallen. A lean simulation framework for stress testing IoT cloud systems. IEEE Trans. Software Eng., 50(7):1827–1851, 2024. [117]

- Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. Using genetic programming to build self-adaptivity into software-defined networks. ACM Trans. Auton. Adapt. Syst., 19(1):2:1–2:35, 2024. [119]

- Jia Li, Shiva Nejati, Mehrdad Sabetzadeh, and Michael McCallen. A domain-specific language for simulation-based testing of IoT edge-to-cloud solutions. In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022, pages 367–378, 2022. [120]

- Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. Learning self-adaptations for IoT networks: A genetic programming approach. In SEAMS '22: IEEE/ACM 17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Pittsburgh, PA, USA, May 21 - 29, 2022. [118]

- Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. Meeting SLOs with Efficiency: A Self-Adaptive Auto-Scaling Approach for Cloud-Native Applications (draft research paper).

1.5 Thesis Structure

This thesis is divided into seven chapters and is organized as follows.

Chapter 1 motivates testing and scaling cloud-based systems, presents the research questions and outlines the contributions.

Chapter 2 provides the necessary background pertaining to cloud-based systems, testing, scaling and optimization approaches.

The subsequent three chapters each focus on a different layer of cloud-based systems and are based on one or two of my prior publications or draft research papers. Considering that these chapters are based on multi-author publications, I include below a summary of my own contributions to the corresponding publications where relevant. Unless otherwise noted, I am the sole contributor to the thesis.

Chapter 3 presents our simulation framework for modeling large numbers of devices in the user layer to stress testing cloud-based IoT applications. It also includes the DSL developed based on this framework.

- This chapter is based on my publications [117, 120]. My contributions include designing the simulation framework and DSL. I also contributed with the evaluation design and comparing it with existing simulation and stress testing tools. Co-authors helped by providing initial research ideas, offering continuous and in-depth feedback during the writing and review process. One co-author also assisted by running some of the repetitive evaluation experiments.

Chapter 4 introduces our self-adaptive framework and our automated approach for scaling SDNs using GP.

- This chapter is based on my publications [118, 119]. The technical implementation and evaluation of the self-adaptive autoscaling approach are my contributions. The framework design is a joint work involving all authors. Co-authors also contributed with continuous, in-depth feedback during the writing and review process.

Chapter 5 describes my approach to auto-scale microservices in cloud-native applications, focusing on meeting diverse SLOs.

- This chapter is based on my draft research paper. The technical implementation and evaluation of the self-adaptive autoscaling approach are my contributions. The framework design is a joint work involving all authors. Co-authors also contributed with continuous, in-depth feedback during the writing and review process.

Chapter 6 elaborates on the related work pertaining to verifying and scaling cloud-based systems.

Chapter 7 summarizes the thesis contributions and discusses avenues for future works.

Chapter 2

Background

In this chapter, I overview the core concepts used throughout this thesis, including cloud infrastructures, and self-adaptation. In Section 2.1, I present Software Defined Networking (SDN), Kubernetes (K8S), and virtualization as a part of modern cloud resource management technologies. In Section 2.2, as I have already introduced the MAPE-K loop in Section 1.3.3, I introduce few main concepts of autoscaling including the horizontal scaling and vertical scaling, and the optimization techniques used in autoscaling, including random search, and genetic programming. Section 2.3 summarizes this chapter.

2.1 Cloud Resource Management

Effective cloud resource management is critical to ensuring optimal performance, cost efficiency, and scalability in cloud-based systems. This section explores some of the key

cloud resource management technologies relevant to this thesis, focusing on networking, orchestration and virtualization.

2.1.1 Software Defined Network

Traditional network infrastructures rely on vendor-specific network devices, such as switches and routers, with configurations tightly coupled to the manufacturers. Once deployed, these infrastructures are difficult to adapt, making it challenging to introduce new protocols or modify hardware components [131]. To address this limitation, Software-Defined Networking (SDN) has emerged as a solution.

Software Defined Networking is a paradigm to network management that separates the controlling responsibilities from network devices (e.g., routers and switches). [101] In SDN, network devices focus solely on forwarding data, forming what is known as the data plane, while the control plane and application plane handle the controlling tasks. We introduce the three planes as follows:

(1) **Data plane**: This plane includes network devices such as routers and switches, which handle packet forwarding based on rules defined and installed by the control plane, as well as their network connections.

(2) **Control plane**: This plane defines the behavior of the data plane. It consists of a central controller, known as the SDN controller, which acts as the "brain" of the network, maintaining a global view of the network and providing it to the application plane. The SDN controller manages and configures the network by installing routing rules on the network devices in the data plane.

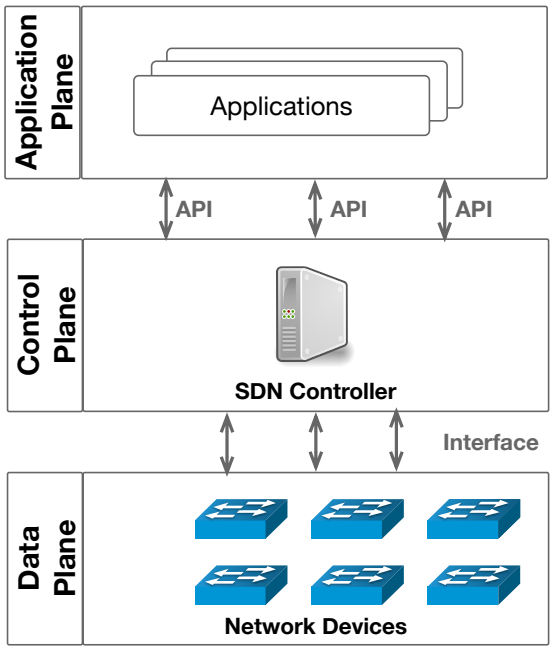

Figure 2.1: Three planes of SDN.

(3) **Application Plane**: This plane defines and enforces management rules through APIs provided by the control plane. It allows interaction and manipulation of network devices' behavior through the SDN controller. Applications in this plane can be used to configure, control, and monitor SDN.

Unlike traditional networks, SDN decouples the control plane from the data plane, enabling centralized management and dynamic configuration [7]. With a centralized controller acting as the "brain" of the network, SDN can optimize traffic flows, enhance security policies, and improve overall network efficiency. In addition, its programmability supports adaptive routing and automated resource allocation, making it valuable for modern cloud environments.

2.1.2 Virtualization

Virtualization is a technology that combines or divides computing resources to present one or many operating environments through methodologies such as hardware and software partitioning or aggregation, machine simulation, emulation, time-sharing, and others [48]. As a foundation technology, virtualization plays a key role in enabling cloud computing by providing a platform to run various services isolated on the cloud [170]. This technology enhances resource utilization, isolation, and flexibility within cloud environments.

Virtualization techniques, including full virtualization, para-virtualization, and containerization, enable cloud platforms to dynamically allocate resources based on workload demands, leading to better performance and cost savings. Full virtualization creates a complete virtual machine (VM) that emulates an entire physical computer, enabling unmodified operating systems and applications to run as if they were on real hardware. In para-virtualization, the guest operating system is modified to communicate directly with the hypervisor, the software layer that manages and allocates hardware resources to virtual machines. This approach improves performance but requires Operating System(OS) modifications. An example of para-virtualization is Xen [23]. Containerization, on the other hand, virtualizes the OS rather than the hardware. Containers, such as Docker [61], run isolated applications within the same OS kernel, enabling faster deployment, better scalability, and more efficient resource usage compared to VMs [26].

2.1.3 Kubernetes

Kubernetes (K8S) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications across multiple hosts [113]. It provides essential features such as self-healing, load balancing, and automated rollouts, making it a fundamental component in resource management of cloud-native architecture. Kubernetes facilitates efficient resource allocation through mechanisms like namespaces, resource quotas, and scheduling policies, ensuring that workloads are distributed effectively and operate optimally within shared infrastructure environments. These capabilities allow applications to maintain high availability and resilience.

In Kubernetes, `Pods` are the smallest deployable units that encapsulate one or more containers, along with their storage and network configurations, and a specification for how to run them. To ensure the desired number of Pods are running at any time, `ReplicaSets` are used to manage and maintain a stable set of replica Pods. Building upon ReplicaSets, `Deployments` provide declarative updates for Pods and ReplicaSets, supporting rolling updates, rollbacks, and scaling operations seamlessly. Kubernetes also leverages `Services` to expose Pods as network-accessible endpoints and manage communication between different parts of an application. Other essential components include `ConfigMaps` and `Secrets` for managing configuration data and sensitive information, and `Ingress` for controlling external access to services within the cluster.

Kubernetes follows a "master-slave architecture", where the master node manages and coordinates the overall cluster, while worker nodes host and run containerized workloads. This architecture enables Kubernetes to form a cluster that orchestrates containers at

scale. By utilizing containerized virtualization, Kubernetes runs microservices as isolated instances, supporting modular and scalable application designs. With the help of the provided API from Kubernetes, we can optimize the performance of applications by dynamically managing resources and adapting deployments to changing demands. This makes Kubernetes particularly useful for cloud-native applications, which demand flexibility, dynamic scaling, and robust fault tolerance in distributed environments.

2.2 Autoscaling

Scaling methods for cloud-based systems can generally be classified into horizontal scaling and vertical scaling. In the following subsections, we first introduce these two fundamental scaling approaches that enable systems to dynamically adapt to workload fluctuations.

Beyond these foundational methods, a wide range of technologies have been developed to automate scaling decisions across different layers and domains of cloud-based systems. Among them, random search and genetic programming are two prominent techniques for optimizing system configurations. In this thesis, we adopt both of them to address autoscaling challenges in cloud-based systems. This section introduces these two technologies in detail.

2.2.1 Vertical Scaling

Vertical scaling is a technique used to increase the capacity of a single instance by adding more resources, such as CPU, memory, or storage. Vertical scaling focuses on enhancing

a single component to handle increased demands. A simple example of vertical scaling is upgrading a server by adding more processors or memory [8].

In cloud-native architectures, vertical scaling becomes crucial when a single microservice experiences fluctuating resource needs. Microservices can range from simple to highly complex tasks, and their resource consumption can vary accordingly. For instance, a microservice performing machine learning tasks may sometimes handle lightweight requests and at other times require significant computational resources. In such cases, vertically scaling the instance allows it to dynamically adjust to the current demand [154]. Kubernetes offers a built-in mechanism called the Vertical Pod Autoscaler (VPA) which automatically adjusts CPU and memory requests for Pods based on actual usage, enabling better resource allocation and scheduling on available nodes.

2.2.2 Horizontal Scaling

Horizontal scaling is a fundamental technique in cloud-based systems for handling increasing workloads by adding more instances of resources—such as virtual machines, containers, or servers—rather than upgrading existing ones. In Kubernetes environments, this typically involves adjusting the number of replicas of a Pod, enabling applications to meet demand by deploying identical Pods [150]. Similarly, in Docker-based environments without orchestration tools, horizontal scaling refers to adding or removing containers of the same service.

Unlike vertical scaling, which enhances the capacity of a single instance, horizontal scaling distributes workloads across multiple instances, making it the dominant approach

in production systems for ensuring performance and reliability [126]. Tools like Kubernetes' Horizontal Pod Autoscaler (HPA) automate this process by dynamically scaling Pods based on metrics such as CPU utilization, within user-defined minimum and maximum thresholds.

Horizontal scaling is particularly valuable for applications facing unpredictable or rapidly growing traffic, as it allows seamless system expansion without significant downtime. This is especially critical in IoT applications, where a sudden surge in device connections can spike workloads. Without autoscaling, services risk overload, degraded performance, or outages. Although overprovisioning could handle peak demand, it leads to substantial resource waste and higher costs—especially in pay-as-you-go cloud environments [154].

By dynamically adjusting service replicas, horizontal autoscaling ensures efficient resource utilization while maintaining service quality. This adaptability makes horizontal scaling essential for building cost-effective, resilient, and high-performing cloud-based systems that can respond to fluctuating user demands in real time.

2.2.3 Random Search

Random Search is one of the simplest yet effective optimization techniques used to explore large and complex search spaces. Unlike systematic or gradient-based methods, Random Search operates by randomly sampling solutions from the search space and evaluating their quality according to a predefined fitness or objective function. This process is repeated until a satisfactory solution is found or a stopping condition is met (e.g., time or number of evaluations).

The key advantage of Random Search lies in its simplicity and generality—it makes no

assumptions about the structure, differentiability, or continuity of the search space, making it suitable for black-box optimization problems where little information is available about the underlying system. Despite its simplicity, Random Search often performs well for high-dimensional optimization tasks and is commonly used as a baseline to compare more sophisticated optimization techniques.

However, since Random Search does not use any information from previously evaluated solutions to guide future search directions, it may require a large number of evaluations to find good solutions, especially in large or rugged search spaces. This lack of guidance limits its efficiency compared to adaptive or population-based methods like Genetic Programming or Evolutionary Algorithms. Nonetheless, Random Search remains a valuable tool in scenarios where other optimization techniques are infeasible or when a quick, unbiased sampling of the solution space is needed.

2.2.4 Genetic Programming

Genetic Programming (GP) is a powerful evolutionary algorithm designed to automatically generate computer programs that solve specific problems. Unlike traditional programming paradigms, such as object-oriented programming or the use of object libraries, which often constrain programmers to predefined code structures, Genetic Programming aims to free the user from manual coding. GP enables computers to evolve their own programs, potentially producing innovative and highly effective solutions that might surpass human-designed code [108].

Inspired by the principles of evolutionary biology, GP operates on a population of

candidate programs that evolve over successive generations through a process of selection, crossover, and mutation. The general workflow of Genetic Programming involves four main steps:

(1) Initialization: Generate an initial population of random program trees composed of functions and terminals relevant to the problem domain.

(2) Evaluation: Execute each program and evaluate its performance based on a fitness function that measures how well it solves the target problem.

(3) Genetic Operators: Create a new generation of programs using reproduction (copying), crossover (recombining parts of two programs), and mutation (randomly altering a program).

(4) Selection of the Best: Identify the best-performing program discovered during the evolutionary process as the final solution.

To effectively apply Genetic Programming (GP), several key design choices must be considered. GP offers highly flexible solution representations, typically using tree structures to encode programs composed of boolean, arithmetic, logical operators, and conditional branching (e.g., IF/THEN statements). This flexibility allows GP to evolve a wide range of solutions, including complex and constrained programs. The evolutionary process is driven by genetic operators such as crossover and mutation to generate new candidate programs. Crossover recombines parts of existing solutions to exploit promising areas, while mutation introduces variations to explore new regions of the search space. Balancing these operators is essential to ensure both diversity and refinement of solutions over successive generations. Another critical aspect of GP is its evaluation mechanism. A fitness function assesses how

well each candidate program solves the target problem, guiding the evolutionary search toward better solutions. By iteratively favoring higher-quality programs, GP progressively improves performance.

The flexibility and adaptability of GP make it well-suited for addressing autoscaling challenges in cloud-based systems. Autoscaling requires dynamic and often complex decision-making to allocate resources efficiently under varying workloads. GP's ability to automatically generate and evolve adaptive control strategies enables it to discover effective autoscaling policies that balance performance and cost without manual intervention. By evolving interpretable and robust solutions, GP provides a promising approach for managing the scalability of cloud-based systems in real-time.

2.3 Summary

This chapter presents the necessary background to support the understanding of the core concepts explored throughout this thesis. Section 2.1 introduces key cloud resource management technologies, including Software-Defined Networking, virtualization, and Kubernetes. Section 2.2 follows with an overview of fundamental autoscaling concepts, such as horizontal and vertical scaling, along with optimization techniques commonly applied in autoscaling, including random search and genetic programming. The content of this chapter establishes a conceptual foundation for the verification and optimization approaches proposed in this thesis.

Chapter 3

A Lean Simulation Framework for Stress Testing Cloud-Based Systems

3.1 Introduction

Motivation. To verify the scalability of cloud-based systems, stress testing is required, where the system or part of the system is subjected to varying workloads to evaluate its ability to scale up and down while maintaining Service Level Objectives (SLOs). As cloud-based systems continue to be deployed in diverse domains such as emergency response, smart cities, smart agriculture, and autonomous vehicles [96], cloud service providers must address a critical challenge: how well their applications scale to accommodate fluctuating demands from a vast number of interconnected devices. Answering this question systematically necessitates stress testing, which assesses how a system performs under

extreme workloads and at its operational limits [44]. However, constructing physical testbeds for such evaluations is often cost-prohibitive, making simulation a practical alternative [118, 119, 190].

Cloud-based IoT systems, in particular, present unique scalability challenges. These systems comprise sensors, actuators, and edge devices, often resulting in a complex user layer with a high density of endpoints and frequent communication patterns. To demonstrate our stress testing approach, this chapter focuses on IoT cloud systems and introduces a simulation-based stress testing framework for cloud-based IoT systems.

IoT simulators may target one or several of these tiers, such as simulating IoT sensors and actuators [11, 106, 125], IoT edge devices [104, 180, 196], and IoT networks [34, 107, 141, 158]. Among these simulators, those emulating the traffic load communicated between edge and cloud systems, known as edge-to-cloud simulators [65], are applicable for stress testing IoT cloud systems.

Problem statement. There are a few IoT edge-to-cloud simulators in the literature [180, 196]; however, these simulators capture extensive details about IoT and edge devices, such as device locations, network protocols, battery capacities, and battery drainage rates. Due to their feature-rich nature, these simulators demand significant computational resources (CPU and memory) to simulate thousands of devices, making them unsuitable for stress testing: Many IoT solutions providers, including our industry partner (discussed later), rely on hosted services for simulation. The greater the resource demands of the simulator, the more challenging and costly it becomes to set up and run for stress testing, limiting the number of IoT and edge devices that can be simulated.

The main problem that we address in this chapter is as follows: A simulator for stress testing IoT cloud systems should support the simulation of large numbers of IoT and edge devices while minimizing computational resource utilization.

Contributions. In this chapter, we develop a lean, purpose-built, edge-to-cloud simulation framework for stress testing IoT cloud systems. Our framework introduces the following novel design features to address scalability challenges in stress-testing of IoT cloud systems:

F1. IoT devices are represented symbolically. Existing IoT simulators capture IoT devices as standalone processes or objects to model their detailed features [180], [106], [165]. In contrast, our simulation framework represents IoT devices symbolically, focusing exclusively on capturing key IoT device properties relevant to stress testing: device payload and communication methods with edge devices. This design feature, denoted by **F1**, results in more parsimonious and efficient simulators, helping address the run-time bloat observed in existing edge-to-cloud simulators.

F2. Edge devices are augmented with configurable variables to account for the variability in start times of edge-device execution and the intervals between data transmissions from IoT devices. These variables minimize bursts in edge-to-cloud data transmission, improving the operational capacity of simulators and enhancing the realism of simulations. This design feature, denoted by **F2**, leads to more consistent network bandwidth usage over time, helping mitigate bursty communication.

F3. Edge devices are grouped into clusters using simulation nodes. These nodes enable more scalable deployment of simulators. Our simulation framework supports the grouping

of simulated edge devices into different clusters, referred to as simulation nodes, which can be executed directly on the native host machine, in independent containers and/or virtual machines, or in a combination of these environments. This design feature, denoted by **F3**, helps address the problem of scaling under limited computational resources. Specifically, **F3** enables more efficient resource allocation, ensuring that each simulation node receives the necessary resources based on its workload. This approach further ensures isolation and stability; issues within one node are contained within its container, preventing system-wide impacts. While some current IoT simulators employ containerization and virtualization, our experiments in Sections 3.2 and 3.4 indicate that none match our method in supporting a large number of IoT and edge devices under restricted CPU and memory resources.

Figure 3.1 provides an overview of the architecture of simulators in our framework, illustrating the three features described above. In our design, edge devices are grouped into simulation nodes. These nodes can run on a native operating system or use virtual machines or Docker containers with designated resources (**F3**). Each edge device connects to an array of IoT devices, which are represented by variables capturing the relevant properties of the IoT devices (**F1**). Unlike IoT devices, each edge device functions as an independent thread, communicating with the cloud. The method, data rate, and frequency of edge-to-cloud communication depend on the number and properties of the IoT devices, as well as the offset and speed variables (**F2**). Offset refers to the timing of edge-device executions, while speed denotes the rate at which edge devices send IoT-device data to the cloud.

To automate the creation of IoT edge-to-cloud simulators, our framework is supported by a domain-specific language (DSL), IoTECS (IoT Edge-to-Cloud Simulation Language).

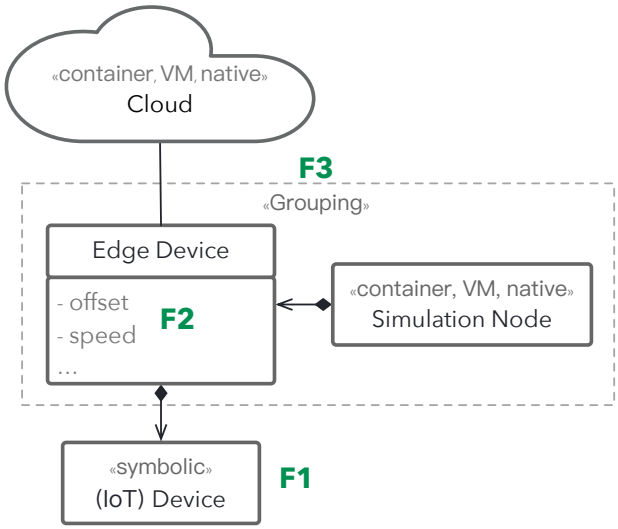

Figure 3.1: Novel design features of our edge-to-cloud simulation framework.

Specifications (models) written in IoTECS are automatically translatable into Java-based simulators. Note that while IoTECS specifications are not executable per se, the simulator code is automatically and entirely derived from these specifications. The generated simulators are then executed to produce simulation results.

Our simulation framework is the outcome of a collaborative research initiative with Cheetah Networks (https://cheetahnetworks.com). Cheetah Networks is a Canadian company specializing in providing solutions for monitoring and optimizing the performance of IoT networks. Their platform employs machine learning to analyze data from IoT and edge devices, assisting network operators, businesses, and organizations in improving the reliability, efficiency, and security of their networks. The company's services cater to various industries, including mining, oil and gas, telecommunications, electric utilities, and smart cities, with clients spanning North America, Europe, and the Middle East.

To ensure that rollouts of new services are not affected by scalability issues when released to clients, Cheetah Networks performs extensive stress testing prior to releases. Since building a testbed with thousands of actual IoT and edge devices is prohibitively expensive, the company uses simulation to mimic real-world loads on their cloud-based platform. Nevertheless, this approach has its own challenges: As we detail in Section 3.2, existing IoT simulators are not suitable for stress testing. Cheetah Networks has therefore relied on in-house testing tools to meet their stress testing needs. These in-house tools are expensive to maintain, require significant manual effort from the company's testing team to configure for different cloud services, are difficult to scale, and are prone to configuration errors. Moreover, these tools expose the testing team to intricate network programming, thereby significantly increasing the learning curve for new staff. The primary goal of our collaboration with Cheetah Networks has been to develop a simulation-based stress testing framework that helps address these challenges. Since its development, our simulation framework has been adopted by Cheetah Networks and, at the time of writing [117], has been continuously applied for more than six months (without any involvement from the researchers) to stress-test some of the company's cloud-based services.

The contributions of this chapter support the thesis contributions as follows:

Con1.1 I propose a simulation framework which enables efficient simulation of large numbers of devices in the user layer that communicate with applications in the cloud layer using limited resources (e.g., a laptop), specifically for stress testing cloud-based systems in IoT domain.

Con1.2 To facilitate simulation construction for practitioners, I develop a domain specific

language (DSL) based on Con1.1, to generate simulators from model-based specifications.

Our toolset and empirical data are publicly available [188] to facilitate reproducibility and so that others can build upon our work.

Structure. The rest of this chapter is organized as follows: Section 3.2 compares the design of our simulation framework with that of two state-of-the-art edge-to-cloud simulators both qualitatively and quantitatively. Section 3.3 presents our proposed DSL, IoTECS. In Section 3.4, we evaluate the applicability and usefulness of simulators generated from IoTECS specifications for stress testing of two real-world IoT cloud systems. The evaluation assesses different simulator configurations, and compares with baseline tools. Section 3.5 summarizes the chapter.

3.2 Challenges and Design Decisions

In this section, we evaluate the designs of two IoT simulators from the literature: EMU-IoT [180] and Fogbed [55]. Similar to our work, these simulators focus on edge-to-cloud simulation, making them the closest comparisons available. Below, we detail the design of each of these two simulators and then evaluate the design against the three challenges that respectively motivated the three features discussed in Section 3.1. For easier reference, we denote the challenges as **C1** (runtime bloat), **C2** (bursty communications), and **C3** (scaling under limited computational resources).

EMU-IoT. An overview of the design of EMU-IoT [180] is shown in Figure 3.2. In

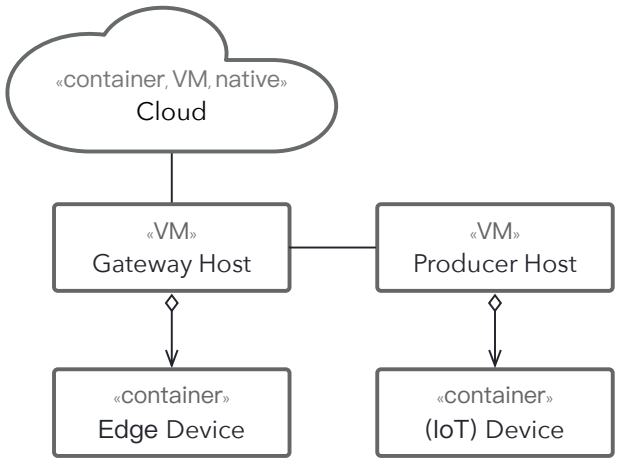

Figure 3.2: The design of EMU-IoT simulator [180].

EMU-IoT, IoT devices are hosted on virtual machines, known as Producer Hosts. Each IoT device is encapsulated within a Docker container. The IoT devices transmit packets to their corresponding edge devices, which in turn relay the packets to the cloud. The edge devices also function within Docker containers. Gateway Hosts, which operate on virtual machines, host edge devices and enable their connection to cloud systems.

Challenge **C1** is not addressed by EMU-IoT, noting that IoT devices operate as standalone processes, and there is no mechanism to prevent excessive system resource consumption when multiple IoT devices are instantiated. As for challenge **C2**, EMU-IoT employs a load balancer to monitor network resources. Its load balancing policy aims to fully utilize the capacity of hosts by ensuring that the first host is utilized before moving to the next in sequence. This approach, however, does not optimize network usage. Hence, communication bursts may happen, especially with a large number of simulated IoT and edge devices.

Regarding challenge **C3**, EMU-IoT isolates each IoT and edge device in separate con-

tainers, as opposed to our framework, which containerizes clusters of edge devices rather than individual ones. The containerization mechanism of EMU-IoT requires significant computational resources for execution, as we demonstrate below.

To further deepen our comparison, we use EMU-IoT to simulate $10,000$ IoT devices and 100 edge devices. To capture this system using EMU-IoT, we need to create 10,100 containers. As for the number of virtual machines (VM), we consider three alternative configurations of EMU-IoT:

- CONFIG1: 50 Producer Hosts, each hosting 200 IoT devices and two Gateway Hosts, each hosting 50 edge devices;
- CONFIG2: 25 Producer Hosts, each hosting 400 IoT devices and two Gateway Hosts, each hosting 50 edge devices;
- CONFIG3: 10 Producer Hosts, each hosting 1000 IoT devices and two Gateway Hosts, each hosting 50 edge devices.

CONFIG1, which is the configuration used to originally evaluate EMU-IoT [180], requires 52 VMs, CONFIG2 requires 27 VMs, and CONFIG3 requires 12 VMs. We note that hosting these many virtual machines and containers requires powerful server machines. We have executed each configuration of EMU-IoT so that IoT devices send small size packets (8bytes) to their corresponding edge devices, and edge devices forward these packets to the cloud without doing any processing. Table 3.1 shows the number of packets received by the cloud for each configuration. As shown in the table, zero or very few packets sent by IoT devices are recieved by the cloud, making this tool unsuitable for cloud stress testing.

Table 3.1: Simulating 10,000 IoT devices using the architecture of EMU-IoT.

Config	CONFIG1	CONFIG2	CONFIG3
# of packets received by the cloud	845	0	0
percentage of packet loss	97.9%	100%	100%

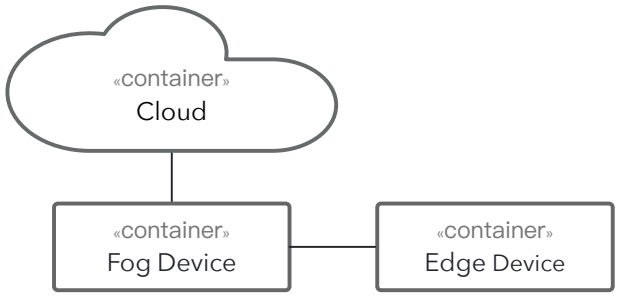

Figure 3.3: The design of the Fogbed simulator [55].

Fogbed. The design of Fogbed [55] is shown in Figure 3.3. Fogbed introduces a fog layer located between the edge and cloud layers. Like edge devices, fog devices are also used for distributing computation, communication, control and storage closer to end users. This fog layer consists of fog devices that receive data packets from edge devices, perform fog computing, and transmit the processed results to the cloud. In Fogbed, every individual edge device, fog device and cloud node is simulated as a Dockerized node.

Fogbed does not explicitly capture IoT devices. Unlike IoTECS and EMU-IoT, where edge devices transmit data packets from IoT devices to the cloud, Fogbed's edge devices generate data packets as a result of edge computing. Since Fogbed does not deal with IoT devices, the runtime bloat challenge, **C1**, is irrelevant to Fogbed.

While IoTECS includes design feature **F2** (Figure 3.1) to optimize network-bandwidth usage, Fogbed does not offer any built-in optimization methods for this purpose. Never-

Table 3.2: Simulating 10,000 IoT devices using the architecture of Fogbed.

Config	CONFIG1	CONFIG2	CONFIG3	CONFIG4
# of packets received by the cloud	1908.2	6386.4	11850	15959.4
percentage of packet loss	95.2%	84.0%	70.4%	60.1%

theless, since Fogbed explicitly captures the network layer, it can be enhanced by network control algorithms that dynamically optimize network bandwidth usage, thus providing low-level facilities to mitigate challenge **C2**. Fogbed employs containerization at the level of edge devices, fog devices, and cloud nodes. However, it does not cluster edge or IoT devices and lacks means for addressing challenge **C3**.

Similar to EMU-IoT, we assess Fogbed by simulating 10,000 IoT devices and 100 edge devices. As mentioned earlier, Fogbed cannot directly capture IoT devices. To do so using Fogbed, we extend it with Docker images of edge devices from our simulator. These images symbolically execute the behaviour of IoT devices. Specifically, we use 100 edge device Docker containers, each hosting 100 IoT devices. We consider four different configurations:

- CONFIG1, where all edge devices communicate with one fog node;
- CONFIG2, where they communicate with two fog nodes;
- CONFIG3, where they communicate with five fog nodes;
- CONFIG4, where they communicate with ten fog nodes.

The fog nodes are configured to receive packets from edge devices and transmit them to the cloud without any processing. Similar to the previous experiment, we execute Fogbed,

where edge devices send 8-byte packets from each IoT device to their corresponding fog nodes, which then forward these packets to the cloud. Table 3.2 shows the number of packets received by the cloud for each configuration. The table indicates that packet loss exceeds 60% for these configurations, demonstrating that less than half of the packets reach the cloud. This suggests that Fogbed is ineffective for cloud stress testing. As we will demonstrate in Section 3.4, simulators built using IoTECS, which leverage features **F1**, **F2**, and **F3** discussed in Section 3.1, can simulate up to 19,000 IoT devices without packet loss, using setups similar to those in the experiments of this section in terms of packet size and sending frequency.

Based on our analysis in this section, we conclude that existing IoT edge-to-cloud simulators have not been developed with stress testing as one of their intended use cases. While the existing simulators may suffice for small-scale testing involving tens or a few hundred devices with updates happening in the order of minutes, they are inadequate for high-throughput IoT systems that handle thousands of devices and require frequent updates in the order of seconds or sub-seconds. For this reason, we empirically compare the performance of simulators generated from IoTECS specifications with two widely used stress testing tools, JMeter and Locust, rather than IoT edge-to-cloud simulators in Section 3.4.

3.3 Simulation Framework

In this section, we introduce our simulation framework. The framework is centred around a domain-specific language (DSL) named IoTECS (IoT Edge-to-Cloud Simulation Language). Specifications written in IoTECS are translatable into Java, resulting in fully

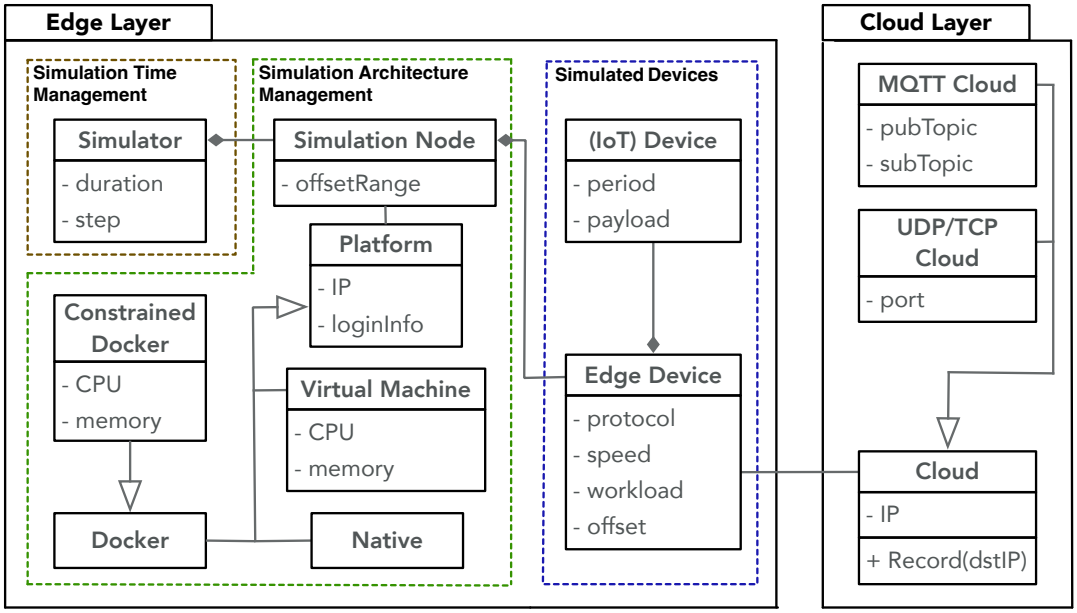

Figure 3.4: IoTECS's underlying conceptual model.

operational simulator code once compiled. This generated simulator can then be executed to produce simulation results. IoTECS is built on top of the conceptual model shown in Figure 3.4. This conceptual model aims to support a scalable and parameterizable architecture for capturing communication between edge devices and cloud applications, enabling simulation-based stress testing of edge-to-cloud solutions. We present our conceptual model in Section 3.3.1, where we further elaborate on our decisions regarding the behaviour of our simulation framework related to the model's concepts. In Section 3.3.2, we discuss IoTECS's syntax and usage, and in Section 3.3.3 — the sanity checks we have implemented in the language to ensure the consistency of IoTECS specifications.

3.3.1 The IoTECS Conceptual Model

The concepts in the conceptual model of Figure 3.4 are arranged under two packages: Edge Layer and Cloud Layer. These two layers communicate through a network in a real IoT system. To ensure that the network does not act as a limiter for how far simulated edge devices can stress the cloud systems, we assume that the edge and cloud layers are connected via an ideal network with minimally low transmission time and no loss. In our conceptual model, the topology of this network is abstracted away, and the network-related information is limited to IPs and communication protocols. In addition, in the case of UDP and TCP protocols, one needs to specify the port numbers, and in the case of MQTT — the publish topics and the subscribe topics. This information is captured as necessary within the edge and cloud concepts.

Cloud Layer. The `Cloud` concept shown in Figure 3.4 represents a cloud application under test. In IoTECS, `Cloud` is specialized into `MQTT Cloud` and `UDP/TCP Cloud`. For an edge-to-cloud simulator to be able to connect to the UDP/TCP cloud under test, we need the `IP` address of the cloud's host machine and the `port` at which the cloud receives incoming data. For an MQTT cloud, we need the `IP` address of the MQTT broker, the publish topic(s) and the subscribe topic(s) for sending packets to the cloud and receiving packets from the cloud, respectively. The `Record(dstIP)` method is responsible for collecting incoming and outgoing packets in the cloud. By default, `dstIP` is set to the IP address of the cloud's host machine when working with a UDP/TCP cloud, to that of the broker when working with an MQTT cloud. The default `dstIP` value enables the `Record(dstIP)` method to collect incoming and outgoing packets on the cloud's host

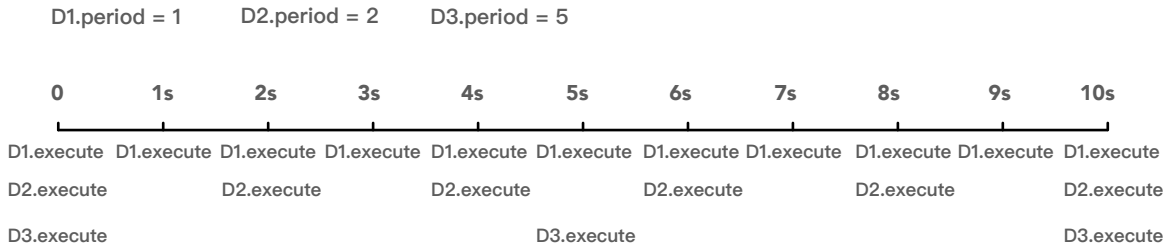

Figure 3.5: An example simulation run representing the execution of three IoT devices, D1, D2, and D3, over a simulation duration of 10s with each step taking 1s. Each IoT device regularly executes based on its period.

machine for UDP/TCP and on the broker for MQTT. For industrial systems, the cloud may include several components, each deployed on a separate container [161]. In order to assess the performance of a specific cloud component along the path leading to the container of that component, we need to assign the IP address of that container to the `dstIP` variable of the `Record(dstIP)` method. For example, MQTT clouds often include a broker and several microservices deployed on separate containers [5, 60]. Our industrial case study is an example of such cloud architecture. We will discuss in Section 3.4.3 how we set `dstIP` to ensure that we asses the performance of the desired cloud component during stress testing. In IoTECS, we can test several cloud applications simultaneously by instantiating `Cloud` multiple times.

Simulation Time Management. The top-level `Simulator` concept in Figure 3.4 captures the attributes required for managing time. We execute each simulation for a given duration and divide this duration into equal time steps. We use the `duration` and `step` attributes to refer to the total simulation duration and to the duration of individual time steps, respectively.

Simulated Devices. The concepts here are `IoT Device` and `Edge Device`. IoT

devices encompass sensors (e.g., temperature sensors) and actuators (e.g., traffic lights). IoT devices connect to and communicate periodically with edge devices by wired or wireless links. For example, a temperature sensor may communicate temperature readings to an edge device every 5s. Edge devices can assume a variety of roles, but their primary function is controlling the data flow at the boundary of a network. Edge devices may perform part of the required processing close to the source (IoT devices) instead of relying on the cloud to do all the processing, which is expensive and can further lead to latency [203]. Edge devices nonetheless usually have limited processing and storage capabilities, and their view on data is limited as well. Edge devices are thus not a substitute for the processing and storage done by the cloud.

It is crucial that IoTECS should be able to simulate edge devices with high fidelity. For this, we do not need to explicitly simulate IoT devices through individual processes or threads. Neither do we need to distinguish between IoT sensors and actuators, noting that both can have two-way communication with edge devices. Instead, as explained in Section 3.1, we utilize design feature **F1**, i.e., the symbolic representation of IoT devices, to address the challenge of run-time bloat. This approach, as we elaborate below, integrates relevant properties of IoT devices into the execution of (simulated) edge devices. For succinctness, where there is no ambiguity, we refer to a sensor or actuator simply as "device". In contrast, when we mean an edge device, we use the term "edge device".

Each instance of `IoT Device` has a `period` attribute, indicating the time interval (in terms of the number of simulation steps) between two successive executions of the IoT device. For example, Figure 3.5 shows a simulation run where the duration is 10s (`duration` = 10) and each step takes 1s (`step` = 1s), dividing the total duration into ten sequential

time slots. An IoT device with `period` = 1 executes its function every step and the one with `period` = 2 executes every other step. IoT devices, whenever they execute, generate a data packet whose size is specified by the `payload` attribute.

Each instance of `Edge Device` owns a group of IoT devices. At every time step, each edge device collects the data packets generated by its associated IoT devices. As mentioned earlier, in IoTECS, the IoT device concept does not induce executable entities. Instead, each instance of an edge device, which is an executable entity in IoTECS, is responsible for emulating the behaviour of the IoT devices associated with it. We provide a precise characterization of the behaviour of edge devices later, when we discuss Algorithm 3.1.

Edge devices send the collected packets to the cloud following the communication protocol indicated by the `protocol` attribute in `Edge Device`. In parallel with sending packets, the edge device performs two more operations: (a) receiving data from the cloud, and (b) performing compute-intensive operations (i.e., edge computing). Edge devices use a `workload` attribute to specify the time duration, measured in milliseconds, seconds or minutes, spent on compute-intensive operations.

Even though edge devices operate at every time step, in real-world applications, they do not all start their operation at the same instant, and may send data packets with some time gaps within a time step [132]. For every edge device, we use two attributes `offset` and `speed` to capture the randomness in start time and the time gap between the sending of packets, respectively. For instance, if we set `offset` of an edge device to 10ms, at every time step, the edge device waits for 10ms before it starts its operation. In this chapter, we assign offset values drawn from a uniform distribution to simulate scenarios where edge

devices executes with minor and random time delays relative to each other. The `speed` attribute specifies how many packets an edge device sends to the cloud per simulation time step. To illustrate, let the simulation time step be 1s. If we set `speed` to 100, the edge device sends up to 100 packets over 1s. That is, the edge device waits for 10ms after sending each packet to the cloud. In the above, we say "up to" because the edge device may send fewer packets if its associated IoT devices produce less packets than what the speed attribute allows to be transmitted. Similarly, if we set `speed` to 1000, the edge device waits for 1ms after sending each packet. To indicate that an edge device can send packets at maximum speed over the time step duration (i.e., without any waiting after sending a packet), one can set `speed` to a designated value, `MAX`. When an edge device's `speed` value is set to "MAX", the edge device consecutively sends all of its packets without any delays between each transmission.

If a large number of (simulated) edge devices attempt to send all their packets to the cloud without any time gap and any offsets, i.e., offsets set to zero, the simulator may not be able to cope with all the send requests, or these requests may congest the network. In both situations, one may experience packet loss or latency at the (simulated) edge or within the network. In view of our goal, which is to stress test cloud systems, we need to make sure that we do not incur packet loss or latency at the edge or in the network. Using the speed attribute, we can pace the send requests in a way that would prevent the simulator and the network from being overwhelmed. By offsetting the start time of edge devices, we avoid situations where all edge devices within the same simulation node send packets to the cloud simultaneously. Further, by staggering the start times of edge devices, we reduce the utilization of I/O resources of the simulation node and the network.

Algorithm 3.1 The behaviour of Edge Device e.

```
Input:
T: Simulation duration
q: Size of the simulation time step
n: Number of IoT devices associated to e
1  sleep(e.offset)
2  run in parallel:
3      Send Process:
4          for i = 0 to T/q do
5              startTime ← getWallClockTime()
6              for every j ∈ {1, ..., n} do
7                  if (i mod e.devices[j].period == 0) then
8                      Send e.devices[j].payload to e.cloud using e.protocol
9                      if e.speed != MAX then
10                         sleep(q / e.speed)
11                 currentTime ← getWallClockTime()
12                 if (q ≤ currentTime - startTime) then
13                     Break out of the loop (of line 6)          ▷ go to the next step
14     Receive Process:
15         Receive packets from e.cloud
16     Compute Process:
17         Perform CPU-intensive operations for duration e.workload   ▷ do edge computing
```

These two attributes, `offset` and `speed`, belong to design feature **F2**. As explained in Section 3.1, this feature helps mitigate bursts in edge-to-cloud data transmission, allowing us to rule out the simulator and the network as the root cause for packet loss and thus localize packet-loss occurrences to the cloud. In our empirical evaluation of Section 3.4, we demonstrate the impact of `speed` and `offset` on our results.

Algorithm 3.1 formally specifies the behaviour of edge devices in IoTECS. An edge device, denoted by 'e' , waits for a duration determined by e's offset parameter (line 1). Then, the edge device performs three operations in parallel: sending packets, receiving packets and performing (edge) computing. At every time step i between 0 to $\frac{T}{q}$ where T

is the simulation duration and q is the size of the step, the edge device e checks each of its associated IoT devices (line 6). If the period of an IoT device indicates that it should be executed at step i (line 7), then e sends the associated payload to the cloud (line 8). After sending the data, e may wait for some time gap whose size depends on the speed parameter of e (lines 9-10). The loop for sending the IoT devices' data to the cloud (lines 6-13) ends as soon as the time step q has elapsed (line 12-13). As a result, if the time step is too short or too many IoT devices are associated to e, some may fail to send their data packets to the cloud.

The receive method is used for receiving packets transmitted from the cloud to edge device e (lines 14-15). To simulate edge computing in IoTECS, edge devices perform some compute-intensive operation for the duration specified by the `workload` attribute (lines 16-17). We note that, to properly simulate edge computing, one needs to explicitly perform operations that keep the edge CPU busy so that the CPU is not allocated to other parallel tasks. To this end, we perform a series of floating-point operations. Algorithm 3.1 stops when the simulation duration T elapses.

Simulation Architecture Management. An important factor in edge-to-cloud simulation is the ability to simulate a large number of IoT and edge devices. As discussed above, each edge device manages the sending of the packets related to its IoT devices sequentially and within the simulation time step q. This assumption matches practice, since in the real-world, edge devices directly handle their related IoT devices. The edge devices themselves, however, run in parallel and are independent from one another. The concepts under Simulation Architecture Management (Figure 3.4) enable a more orderly handling of concurrency, in turn allowing us to optimize the number of edge devices that can run in

parallel.

To maximize the capacity of our simulator in terms of the number of parallel edge devices under limited computational resources, we introduce an additional tier into our simulator's architecture to group the edge devices into clusters. This tier, corresponding to design feature **F3** in Section 3.1, is represented by the `Simulation Node` concept in the model of Figure 3.4. The `Simulation Node` concept includes the offsetRange attribute which limits the offset values of its associated edge devices. That is, the offsetRange value indicates the maximum offset values that edge devices associated to the simulation node can have. We specify offsetRange as a percentage of the simulation time step. To illustrate, let the simulation time step be 1s. If we set `offsetRange` to 20%, the offset value for all edge devices associated to this simulation node is a value between 0 to 0.2s, or 0 to 200ms.

Each simulation node is associated with a platform, represented by the `Platform` concept. Edge devices that belong to a given simulation node all execute in parallel and on the platform of the simulation node. The `Platform` concept has as attributes `IP` and `loginInfo` to enable access to the platform host machine. In IoTECS, `Platform` is specialized into `Native` (i.e., running on a native operating system), `Virtual Machine` and `Docker`. For a virtual machine (VM), we need to specify the `CPU` and `memory` attributes, respectively indicating the CPU and memory allocated to the VM. Docker containers can be unconstrained or constrained. For the latter (i.e., `Constrained Docker`), similar to a VM, we need to specify the `CPU` and `memory` attributes.

The idea of clustering edge devices into simulation nodes in order to scale simulators is inspired by the notion of "super indivduals" in massively multi-agent simulators [186].

Algorithm 3.2 The behaviour of Simulation Node sn.

Input:
m : Number of edge devices associated to sn
q : Size of the simulation time step

```
1 for every i ∈ {1, ..., m} do
2     initialize sn.edgeDevice[i]
3     select a random number ran in [0 .. sn.offsetRange · q]
4     sn.edgeDevice[i].offset ← ran
5 In parallel, run all sn.edgeDevice[i] (i ∈ {1, ..., m})      ▷ run Algorithm 3.1 for each sn.edgeAgent[i]
```

In our context, such clustering provides a mechanism to reduce contention over CPU, memory and ports. For example, instead of running 100 parallel edge devices on a single computer, we create ten Dockers by dividing the single computer resources (CPU, memory and ports) between the Dockers equally. We then use each Docker to host ten (simulated) edge devices. As we will demonstrate in our evaluation of Section 3.4, the Docker option considerably improves the scalability of our simulator, compared to the native option.

Another motivation for defining simulation nodes is enabling the restriction of simulation resources. By setting limits on CPU and memory for virtual machines and (constrained) Dockers, users can more effectively manage simulation costs, especially when utilizing third-party computation resources. We will revisit this topic in our conclusion section (Section 3.5), where we discuss cost-awareness for simulation.

The behaviour of `Simulation Node` is specified in Algorithm 3.2. Each simulation node, upon creation, initializes each of its associated edge devices (lines 1-2). To assign the offset parameter for each edge device associated with the simulation node (line 3-4), we generate a random number within the following range: $[0..offsetRange \cdot q]$, where q is the simulation time step. To ensure that all values within this range are equally likely to be selected, we use a discrete uniform distribution for generating (integer) offset values.

Subsequently, the edge devices are executed in parallel (line 5). That is, Algorithm 3.1 is called for each edge device.

Algorithm 3.3 shows the overall behaviour of our simulator. The simulator starts by initializing the platforms of the simulation nodes (line 3). It then transfers the simulation code to the platforms (line 4). Next, the simulation nodes start to run in parallel on their respective platforms (line 5). The nodes run until the simulation duration elapses (line 6), at which point the nodes and their platforms are cleaned up (lines 7-9).

Our simulator can be configured to represent different numbers of IoT and edge devices. As these numbers increase, the application(s) in the cloud layer are put under more stress. In this way, our simulator makes it possible to determine how far the cloud application(s) under test can be stretched without degradation in their quality of service. Upon the termination of a simulation round, we collect certain metrics, defined in Section 3.4.4. These metrics help determine (1) whether, given the computational resources available, the simulator can mimic the desired numbers of IoT and edge devices without getting overwhelmed, and (2) whether the cloud system(s) are able to process and respond in reasonable time to all the messages received from the simulator.

3.3.2 The IoTECS DSL

IoTECS aims to support practitioners in creating simulators that are instances of the conceptual model of Figure 3.4. We provide an implementation of IoTECS using Xtext [64]. Specifically, we use Xtext's grammar language to define IoTECS's grammar. To generate executable simulators from specifications written in IoTECS, we use Xtend [63]. More

Algorithm 3.3 The bahviour of Simulator *sim*.

Input: l: Number of simulation nodes

1 **for** every $i \in \{1, \ldots, l\}$ **do**
2 Let p denote $sim.sn[i].platform$
3 Initialize p
4 Transfer $sim.sn[i]$ to p using $p.IP$ and $p.loginInfo$
5 In parallel, run all $sim.sn[i]$ ($i \in \{1, \ldots, l\}$) ▷ run Algorithm 3.2 for each $sim.sn[i]$
6 sleep ($sim.duration$)
7 **for** every $i \in \{1, \ldots, l\}$ **do**
8 Terminate $sim.sn[i]$ if it is still running
9 Clean up $sim.sn[i].platform$ and $sim.sn[i]$

precisely, we use Xtend to retrieve all the objects defined in an IoTECS specification; these objects in turn drive the instantiation of a number of a-priori-defined Xtend templates. Our Xtend templates include: (1) Java-based implementations of the algorithms in Section 3.3.1, (2) scripts for setting up / starting virtual machines and containers, (3) scripts for uploading and running simulator code on remote platforms, (4) scripts for downloading simulation results from simulation nodes, and (5) scripts for analyzing and reporting simulation results.

In the rest of this section, we provide the syntax of IoTECS using the Backus-Naur Form (BNF) notation: non-terminal symbols are surrounded by angle brackets, while terminal symbols are enclosed in single quotes; ::= means that the symbol on the left must be replaced with the expression on the right; a vertical bar is used to separate alternatives; a star stands for zero or more occurrences; a plus indicates one or more occurrences; square brackets represent optional elements; parentheses are used for grouping; literals are enclosed in ‘’. Below, we present and exemplify the syntax of IoTECS. Full details about the design of IoTECS can be found in our artifacts [188].

Cloud. The syntax of `Cloud` is defined as follows:

⟨*Cloud*⟩ ::= ‘`Cloud:`’ ⟨*ID*⟩ ‘`{`’ (α1)

‘`IP:`’ ⟨*IP*⟩ (α2)

(‘`port:`’ ⟨*integer*⟩ | (α3)

‘`pubTopic:`’ ⟨*string*⟩ [‘`, subTopic:`’ ⟨*string*⟩]) (α4)

[‘`Methods: {`’ ‘`Record(`’ ⟨*IP*⟩ ‘`)`’ ‘`}`’] (α5)

‘`}`’

⟨*ID*⟩ ::= (‘`a`’-‘`z`’ | ‘`A`’-‘`Z`’ | ‘`0`’-‘`9`’)+ (α6)

⟨*IP*⟩ ::= (‘`0`’-‘`9`’)+‘`.`’(‘`0`’-‘`9`’)+‘`.`’(‘`0`’-‘`9`’)+‘`.`’(‘`0`’-‘`9`’)+ (α7)

The above syntax requires a unique `ID` to be assigned to each cloud instance (line α1). IoTECS supports two types of IoT cloud systems: one uses MQTT as the messaging protocol and the other uses UDP or TCP. For the first type of cloud systems, IoTECS requires the `IP` address of the MQTT broker, a topic for publishing messages to the broker and a topic for subscribing responses from the broker (lines α2 - α4). The subscribe topic is optional because some cloud systems may not send packets back to edge devices. For the TCP or UDP cloud systems, the `IP` address and a `port` of the cloud's host machine are needed to enable communication between the edge devices and the cloud (lines α2 - α3). The optional ‘`Record(`⟨`IP`⟩`)`’ method (line α5) initiates a network-traffic monitor for the specified `IP`. This enables the recording of all incoming and outgoing packets associated with `IP`. The recorded data can be used for evaluating stress testing metrics like packet loss and delay.

Figure 3.6 shows a snippet of an IoTECS specification specifying two `Cloud` instances. For cloud instance C1, the `IP` and `port` attributes respectively specify the IP address of the

```
1  Cloud:C1 {
2      IP:192.168.0.2
3      port:1883
4  }
5  Cloud:C2 {
6      IP:192.168.0.3
7      pubTopic:pub
8      subTopic:sub
9  }
```

Figure 3.6: Example cloud systems defined using IoTECS.

cloud's host machine and the port at which incoming UDP or TCP packets are received. Both IPv4 and IPv6 addresses are supported for the `IP` attribute. In contrast, C2 is an instance of MQTT cloud. The `pubTopic` and `subTopic` attributes specify the publish topic and the subscribe topic of the MQTT broker (within the cloud) respectively. We note that the code for instances of `Cloud` is not meant to be generated by IoTECS. Rather, each `Cloud` instance represents an existing cloud system that needs to be stress-tested.

Simulator. Each IoTECS specification has exactly one instance of the `Simulator` concept. The syntax of `Simulator` is defined as follows:

⟨*Simulator*⟩ ::= '`Simulator:`' '`{`' (β1)

'`duration:`' ⟨*integer*⟩ ⟨*TimeUnit*⟩ (β2)

'`step:`' ⟨*integer*⟩ ⟨*TimeUnit*⟩ (β3)

'`simulationNodes:{`'⟨*ID*⟩'`[`'⟨*integer*⟩'`]`' (β4)

('`,`'⟨*ID*⟩'`[`'⟨*integer*⟩'`]`')* '`}`'

'`}`'

⟨*TimeUnit*⟩ ::= '`ms`' | '`s`' | '`min`' | '`h`' (β5)

```
1  Simulator: {
2      duration:10s
3      step:1s
4      simulationNodes:{SN1[5],SN2[1]}
5  }
```

Figure 3.7: Example simulator defined using IoTECS.

In the above syntax, `duration` (line β2) is used to specify the total duration of simulation. On the other hand, `step` (line β3) is used to declare the simulation time step. To illustrate, consider the snippet in Figure 3.7. Here, the simulator has its `duration` set to 10s and its `step` to 1s; this results in dividing the simulation into 10 steps of equal length with each step running for 1s. The time unit for `duration` and `step` can be in milliseconds (ms), seconds (s), minutes (m) or hours (h) (line β5). A simulator needs to declare its simulation nodes which are expressed as a comma-separated list after the keyword `'simulationNodes'` (line β4). Each item in the list is the ID of a simulation node type followed by the number of node instances of this type in square brackets. (see line 4 of Figure 3.7 for an example). On this line, we are stating that the simulator has six simulation nodes: five nodes of type `SN1` and one node of type `SN2`. We next illustrate how to define the simulation node types (in this case, `SN1` and `SN2`) and their associated execution platforms.

Simulation nodes and platforms. As discussed in Section 3.3.1, IoTECS uses the notion of simulation node for grouping simulated edge devices and to further specify the platform on which a group of edge devices runs. The syntax of `Simulation Node` is defined as follows:

⟨*SimulationNode*⟩ ::= ‘`SimulationNode:`’ ⟨*ID*⟩ ‘`{`’ (γ1)

‘`platform:`’ ⟨*ID*⟩ (γ2)

‘`offsetRange:`’ ⟨*integer*⟩‘`%`’ (γ3)

‘`EdgeDevices:{`’⟨*ID*⟩‘`[`’⟨*integer*⟩‘`]`’ (γ4)

(‘`,`’⟨*ID*⟩‘`[`’⟨*integer*⟩‘`]`’)* ‘`}`’

‘`}`’

Each instance of a simulation node must have a (unique) ID (line γ1). The ID of the platform on which the simulation node runs is indicated after the keyword ‘platform’ (line γ2). The keyword ‘offsetRange’ specifies an upper bound for the offset values for all the edge devices related to this simulation node (line γ3). Recall from Section 3.3.1 that, by introducing offsets, one can circumvent bursts, i.e., situations where all edge devices transmit at the same time. Using offsets increases the realism of simulations for applications where bursts are unlikely to occur or are explicitly mitigated. In the snippet of Figure 3.8, we define two simulation node types, `SN1` and `SN2`, each having a platform, an offsetRange and a set of edge devices. Similar to our convention for specifying the simulation nodes contained in a simulator (see Figure 3.7), we define the edge devices contained in a simulation node via an edge-device type and a number of instances (line γ4). For example, in Figure 3.8, `SN1` has ten edge devices: seven of type `E1` and three of type `E2` (we will momentarily illustrate the specification of edge-device types). For `SN1`, the `offsetRange` is set to 20% of the time step duration, i.e., 200ms, while for `SN2`, `offsetRange` is set to 60% of the time step duration, i.e., 600ms. Hence, the offsets of

```
1  SimulationNode: SN1 {
2      platform:P1
3      offsetRange:20%
4      EdgeDevices:{E1[7],E2[3]}
5  }
6  SimulationNode: SN2 {
7      platform:P2
8      offsetRange:60%
9      EdgeDevices:{E1[30]}
10 }
11
12 Platform: P1{
13     type: Docker
14     IP: 192.168.0.4
15     username: user2
16     CPU: 4
17     memory: 2G
18 }
19
20 Platform: P2{
21     type: Docker
22     CPU: 2
23     memory: 2G
24 }
```

Figure 3.8: Example simulation nodes and platforms defined using IoTECS.

edge devices belonging to `SN1` are selected from the range $[0..200ms]$, while the offsets of edge devices belonging to `SN2` are selected from the range $[0..600ms]$.

A platform represents either a physical or virtual host for a simulation node. The syntax of `Platform` is defined as follows:

$\langle Platform \rangle$ ::= '`Platform:`' $\langle ID \rangle$ '`{`' ($\delta 1$)

'`type:`' $\langle Type \rangle$ ($\delta 2$)

['`IP: `' $\langle IP \rangle$ '`userName:`' $\langle string \rangle$] ($\delta 3$)

['`CPU:`' $\langle integer \rangle$ '`memory:`' $\langle integer \rangle$ '`G`'] ($\delta 4$)

'}'

⟨*Type*⟩ ::= '`Native`' | '`Docker`' | '`VM`' (δ5)

Each platform has a type (line δ2) that assumes one of the following values: `Native`, `VM` or `Docker` (line δ5). To be able to run a simulation node when its platform is remote (i.e., not the local host), the simulation code and scripts need to be transferred to and set up on the platform first. For this purpose, we need to specify an IP (line δ3) and login credentials (username and password) in instances of the `Platform` concept. As a security measure, we elect to not capture passwords in IoTECS; this is to avoid storing sensitive information in plain text. Instead, users are prompted directly for a password by any host involved in the simulation process, as per the scripts generated from an IoTECS specification. The `Platform` concept is illustrated in the snippet of Figure 3.8. Here, `P1`, which is of type `Docker`, is specified on lines 12-18. When a platform is hosted locally, one does not need `IP` and `loginInfo`. This is illustrated by `P2` — another platform of type `Docker` — on lines 20-24 of the snippet. For a `Docker` platform that is constrained and for any `VM` platform, we specify the required attributes indicated in our conceptual model of Figure 3.4. For example, platform `P1` is a constrained Docker with 4 CPUs and 2G of memory.

Edge and IoT devices. The syntax for defining an `Edge Device` is as follows:

⟨*EdgeDevice*⟩ ::= '`EdgeDevice:`' ⟨*ID*⟩ '`{`' (ϵ1)

'`protocol:`' ⟨*Protocol*⟩ (ϵ2)

'`speed:`' ⟨*integer*⟩ | '`MAX`' (ϵ3)

'`cloud:`' ⟨*ID*⟩ (ϵ4)

'`devices:{`'⟨*ID*⟩'`[`'⟨*integer*⟩'`]`' (ϵ5)

(‘,’⟨*ID*⟩‘[’⟨*integer*⟩‘]’)* ‘}’

‘`workload:`’ ⟨*integer*⟩⟨*WorkloadTimeUnit*⟩ (ϵ6)

‘}’

⟨*Protocol*⟩ ::= ‘`UDP`’ | ‘`TCP`’ | ‘`MQTT`’ (ϵ7)

⟨*WorkloadTimeUnit*⟩ ::= ‘`ms`’ | ‘`s`’ | ‘`min`’ (ϵ8)

Each edge device has a set of IoT devices (line ϵ5) associated to it. For example, in the snippet of Figure 3.9, edge-device type `E2` is associated with 10 IoT devices of type `D1` and 20 IoT devices of type `D2` (we will shortly exemplify IoT-device types).

Each edge device communicates with one cloud system which is indicated after the keyword ‘cloud’ (line ϵ4). For example, in Figure 3.9, `E1` is specified as communicating with cloud system `C1`. The protocol used by an edge device for communication with the cloud is captured by the `protocol` attribute (line ϵ2). Currently, IoTECS supports the `UDP`, `TCP` and `MQTT` protocols. The `speed` attribute describes the number of packets sent to the associated cloud system in one time step. If one does not want to constrain the number of packets, the `speed` attribute should be set to the designated value of ‘MAX’. For instance, in Figure 3.9, the `speed` is set to 100 for `E1` and 1000 for `E2`. Since `step` has been set to 1s (see Figure 3.7), `E1` waits 10ms between sending two consecutive packets; this wait time is 1ms for `E2`. The `workload` attribute (line ϵ6) indicates the amount of edge computing to be done by the edge device (lines 16-17 of Algorithm 3.1). The time unit for `workload` can be milliseconds (ms), seconds (s) or minutes (m). Note that, in parallel with sending packets and edge computing, an edge device also receives packets from the cloud (lines 14-15 of Algorithm 3.1). The protocol for receiving data from the cloud is the same

```
1  EdgeDevice: E1 {
2      protocol:TCP
3      speed:100
4      cloud:C1
5      devices:{D1[100]}
6  }
7  EdgeDevice: E2 {
8      protocol:MQTT
9      speed:1000
10     cloud:C2
11     devices:{D1[10],D2[20]}
12     workload:20ms
13 }
```

Figure 3.9: Example edge devices defined using IoTECS.

as that used for sending data to the cloud (i.e., is as specified by the `protocol` attribute).

`IoT device` is the lowest-level entity in IoTECS and its syntax is defined as follows:

$\langle Device \rangle$ ::= '`Device:`' $\langle ID \rangle$ '`{`' (ζ1)

'`period:`' $\langle integer \rangle$ (ζ2)

'`payload:`' ($\langle integer \rangle$ $\langle PayloadUnit \rangle$) | $\langle string \rangle$ (ζ3)

'`}`'

$\langle PayloadUnit \rangle$ ::= '`b`' | '`kb`' | '`mb`' (ζ4)

Each IoT device (type) has two attributes. The first one is `period` (line ζ2) which determines the frequency of execution. For example, in the snippet shown in Figure 3.10, device `D1` generates a packet every second, while device `D2` generates a packet every two seconds. The second attribute is `payload` (line ζ3). This attribute specifies the actual payload content, e.g., `payload: "23C"`, or alternatively, the size of the packet that the device sends every time the device executes. In the latter case, the payload unit can be

```
1 Device: D1 {
2     period:1
3     payload:60b
4 }
5 Device: D2 {
6     period:2
7     payload:100b
8 }
```

Figure 3.10: Example (IoT) devices defined using IoTECS.

bytes (b), kilo bytes (kb) or mega bytes (mb). When a size unit is indicated for the payload, the content of the payload is generated randomly as per the requested size. This option, illustrated in Figure 3.10, is convenient when the actual payload is unimportant for simulation purposes (e.g., in a simple cloud storage application).

3.3.3 Sanity Checks

This section describes the syntactic and semantic sanity checks implemented in IoTECS's Eclipse-based text editor. Users get instant errors or warnings for any violations. There are four groups of sanity checks as we explain below:

Declaration/usage checking. IoTECS requires explicit declarations for all entities in a specification. For example, to define a `Simulator` instance, one must specify its simulation node(s), which can be declared before or after the `Simulator` instance. Furthermore, IoTECS ensures that every declaration is used at least once in the specification. This prevents unused or redundant entities, which may indicate errors or oversights. Unused declarations trigger warnings.

Type checking. IoTECS enforces type checking to ensure consistent and correct usage

of entity types throughout a specification. For example, IoTECS expects IDs of simulation nodes to follow the keyword 'simulationNodes' when a `Simulator` instance is being defined. If IoTECS detects IDs of other entity types (e.g., devices or platforms) instead, it reports a type mismatch error.

Protocol checking. IoTECS verifies that the `Protocol` attribute of an edge device matches the cloud instance type it communicates with. If an edge device has an MQTT cloud instance as its `cloud` attribute, its communication protocol should be "MQTT". Likewise, if it has a UDP/TCP cloud instance as its `cloud` attribute, its communication protocol should be either "UDP" or "TCP", depending on the packet type used for cloud communication.

Time-step checking. IoTECS verifies that the simulation time step is a divisor of the simulation duration. This ensures that one has a whole number of steps within the simulation duration.

3.4 Evaluation

In this section, we evaluate the applicability and usefulness of simulators generated from IoTECS specifications for stress testing IoT cloud systems. We use the term simulator to refer to the executable code generated from instantiating the edge-layer concepts of the model of Figure 3.4; we use the term cloud to refer to instances of the cloud concept in this model. To evaluate the performance of simulators derived from IoTECS specifications, we use two real-world IoT cloud systems as case studies. The first case study is a cloud-based

IoT monitoring system developed by one of our industry partners; this cloud system operates based on both UDP and TCP protocols. For our first case study, due to confidentiality concerns, we have built our own cloud system and made it publicly available instead of using our partner's system directly. The second case study involves an industrial, cloud-native IoT connected vehicle system that monitors sensors located on connected vehicles and operates based on the MQTT protocol. In this section, we refer to the first case study system as the the UDP/TCP cloud system and the second one as the MQTT cloud system. To enhance readability and conciseness in the remainder of this section, we will use the terms "our simulator" or "the simulator" to denote a simulator that has been derived from an IoTECS specification. The research questions (RQs) that we investigate are as follows:

RQ3.1. (Simulator Configuration) How can we configure our simulator so that it can, with limited computational resources, simulate a large number of IoT and edge devices? With RQ3.1, we examine which simulator platform option(s) — among Native, Docker and Virtual Machine as envisaged by the conceptual model of Figure 3.4 — would allow one to maximize the number of instantiated IoT and edge devices, thereby best supporting our motivating use case in Section 3.2. In particular, we require the whole simulator to run on a single machine (laptop). This requirement is in line with the needs of our partner; they need the simulator machine to be portable so that it can be brought to different sites and connected to different networks. In addition, we examine the impact of the speed value for edge devices on the performance of the simulator. The simulator performance is assessed by measuring (a) packet loss at the machine that hosts the simulator and (b) packet transmission times. Since the focus of RQ3.1 is on the simulator, our analysis is

agnostic to the choice of the cloud system. For RQ3.1, we elect to use the UDP/TCP cloud system. This cloud has lower execution time compared to the MQTT cloud, in turn allowing us to run more extensive experimentation.

RQ3.2. (Scalability) What is the impact of offset values on simulator scalability? In many real-world scenarios, it is reasonable to assume that requests from edge devices are distributed more evenly, unlike the worst-case scenario explored in RQ3.1 where all edge devices initiate communication with the cloud simultaneously. To systematically examine our simulator's scalability under the assumption of more evenly distributed requests, RQ3.2 considers various ranges of offset values and assesses how widening the offset ranges affects scalability. Indirectly, RQ3.2 further enables us to empirically demonstrate the benefit of introducing offsets (in situations where engineers can control or have prior knowledge about offset ranges) as a way to increase the cloud's capacity to handle larger numbers of IoT and edge devices. To answer RQ3.2, we use the best-performing platform identified in RQ3.1 and the UDP/TCP cloud.

RQ3.3. (Comparison with Baselines) How effective is our simulator at stress testing IoT cloud systems compared to state-of-the-art stress testing tools? We benchmark our simulator against two widely used, open-source baseline tools for stress testing: Apache JMeter [16] and Locust [124]. Both tools are commonly used for stress testing [18, 91, 94, 99, 110, 116]. We recall our conclusion from Section 3.2, where we determined that existing IoT edge-to-cloud simulators are inadequate for large-scale stress testing. In light of this conclusion, we choose to compare with JMeter and Locust (rather than IoT simulators) as our baselines, noting that JMeter and Locust are general-purpose, industry-strength performance testing tools. In Section 3.4.2, we assess the ability of JMeter and Locust

to tackle the three stress testing challenges, C1, C2, and C3, outlined in Section 3.2. In this RQ, we assess whether and how our simulation architecture is advantageous over JMeter and Locust in terms of the number of IoT and edge devices that can be successfully simulated. For a fair comparison, we use the UDP/TCP cloud: JMeter offers direct support for TCP, while Locust — originally designed solely for HTTP — can be easily adapted through existing libraries to work directly with UDP/TCP. Similar to RQ3.1 and RQ3.2, our focus in RQ3.3 is on assessing the performance of the simulator. We thus measure packet loss incurred at the host machine and transmission delay times when running our simulator or the baselines tools.

RQ3.4. (Usefulness) Can our simulator be used effectively for stress testing real-world IoT cloud systems? We use the best-performing simulator platform identified in RQ3.1 (among Native, Virtual Machine and Docker) to answer this research question. We assess how well one can use our simulator to determine the service capacity of an IoT cloud system, i.e., the maximum number of IoT and edge devices that a cloud system can effectively handle. For RQ3.4, we use both of our case studies, i.e. both the UDP/TCP and MQTT cloud systems. In contrast to the previous RQs which focused on studying and optimizing the behaviour of our simulator, RQ3.4 is concerned with in-the-field usage of the simulator, i.e., determining the limits of an IoT cloud under test. For RQ3.4, we measure packet loss at the destination machine hosting the cloud under test.

3.4.1 Case Studies

In this section, we describe our case-study systems: The UDP/TCP cloud and the MQTT cloud.

3.4.1.1 The UDP/TCP Cloud

Our UDP/TCP cloud, which is part of an industrial IoT network monitoring system, implements a packet loopback that echoes any UDP or TCP packet it receives from the edge layer back to the sender. This cloud system is used for measuring the quality of the network connection between the edge devices and the cloud.

To develop a simulator for stress testing this case-study system, we create two instances of the UDP/TCP cloud concept described in Section 3.3.1. One instance is configured to receive and transmit UDP packets, while the other handles TCP packets. We configure both instances by setting their IP attributes to match the cloud system's IP address. Further, we assign the port of the UDP cloud instance to correspond to the UDP socket's port, and the port of the other instance to match the port for the TCP socket. Since the cloud's host machine serves as the destination for all inbound packets, for the `Record` method in our conceptual model, we use the default value of the `dstIP` parameter, which corresponds to the IP address of the cloud's host machine.

3.4.1.2 The MQTT Cloud

The IoT connected vehicle system monitors a large fleet of vehicles in real time by tracking and storing their location, speed, odometer and fuel-tank temperature. Users can access

the latest information of the fleet through a web application that flags the vehicles that exceed predefined thresholds for fuel-tank temperature or speed. This system operates on a Kubernetes [113] cluster and follows a state-of-the-art big data processing approach, Lambda architecture [135], which combines both real-time and offline processing capabilities.

Figure 3.11 provides a conceptual representation of the architecture of the IoT connected vehicle system. This system includes an MQTT broker that accepts packets from vehicles. At each time step, each vehicle sends to the broker a packet consisting of the vehicle's GPS location, speed, odometer and fuel-tank temperature. The information in these packets enables the cloud system to monitor and track the vehicles' position, driving direction, and speed as well as the temperature of the vehicles' fuel tanks.

The broker forwards these packets to a stream processor that in turn sends the packets to two different destinations: (1) an in-memory data grid which is a fast storage system for online processing, and (2) a persistent storage system for offline processing. Applications that need real-time data, such as real-time diagnostics and driver-behaviour analysis, query the fast in-memory grid, while applications that do not need real-time data, such as profiling drivers' data based on machine learning, use the persistent database.

To develop a simulator for the IoT connected vehicle system based on our approach, we create an edge device instance for each vehicle. Further, we create an instance of the MQTT cloud concept, and set its `IP` attribute to the broker's IP in Figure 3.11 and its `pubTopic` parameter to the publish topic that the broker accepts. We consider the in-memory data grid as the final destination for all packets and set the `dstIP` parameter of

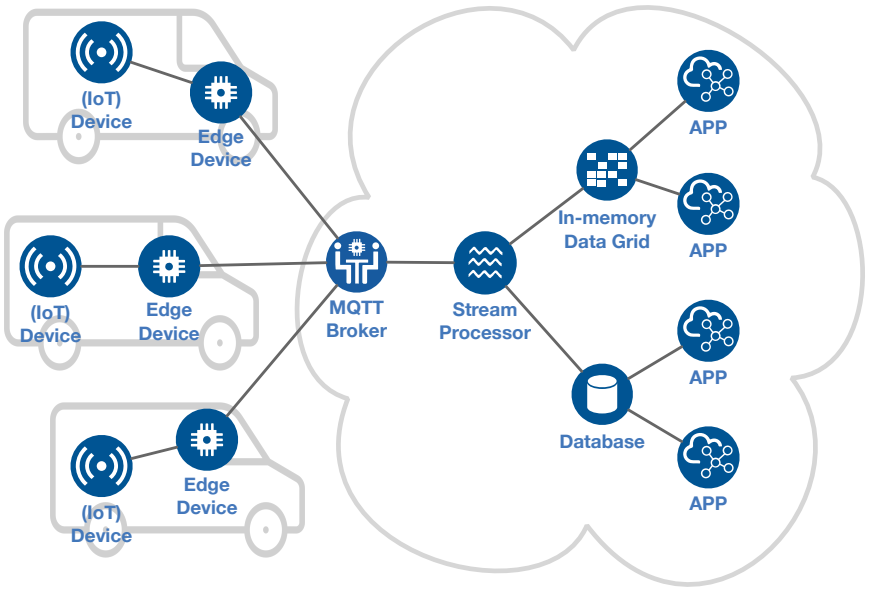

Figure 3.11: Overview of our IoT connected vehicle case study.

the `Record` method in our conceptual model to the IP address of the in-memory data grid in Figure 3.11. Note that since the IoT connected vehicle system does not send any data back to the vehicles, there is no need to set the `subTopic` parameter for our instance of the MQTT cloud concept.

3.4.2 Baseline Tools for Stress Testing

We select Apache JMeter and Locust as baseline tools to compare the performance of our simulator against. JMeter and Locust are both open-source tools for performance testing web applications. JMeter is developed by the Apache Software Foundation and written in Java. It supports various communication protocols including HTTP, FTP, TCP

and SOAP/REST. Locust is built using Python and offers an easy-to-use web interface. Locust comes with built-in support for only HTTP/HTTPS.

We use JMeter and Locust to stress test our UDP/TCP cloud system. JMeter already supports TCP; hence, we need to adapt it only for UDP. As for Locust, an adaptation to both UDP and TCP is required. To adapt JMeter, we use plugins for UDP support [166]; and to adapt Locust, we use Python's socket module, which provides UDP and TCP support.

We assess the performance of JMeter and Locust in tackling stress testing challenges **C1**, **C2**, and **C3** detailed in Section 3.2. Since JMeter and Locust are general-purpose tools, they do not have predefined means for capturing IoT and edge devices. Consequently, they offer no direct solution for simulating large arrays of IoT and edge devices without causing runtime bloat, i.e., challenge **C1**. Furthermore, these tools lack built-in mechanisms for addressing challenges **C2** and **C3**. To compare simulators derived from IoTECS specifications with JMeter and Locust, we adapted JMeter and Locust to symbolically represent IoT and edge devices using their parameters, as detailed in Section 3.4.3.3. Through this adaptation, we effectively implemented design feature **F1** (Section 3.1) into JMeter and Locust. Incorporating design features **F2** and **F3** into these tools would nonetheless necessitate a reconsideration of the tools' designs and significant re-engineering and implementation effort and was therefore not attempted. Since JMeter and Locust lack support for creating simulation nodes on virtual machines or Docker containers, when using JMeter and Locust, we run all simulation nodes natively on the host machine.

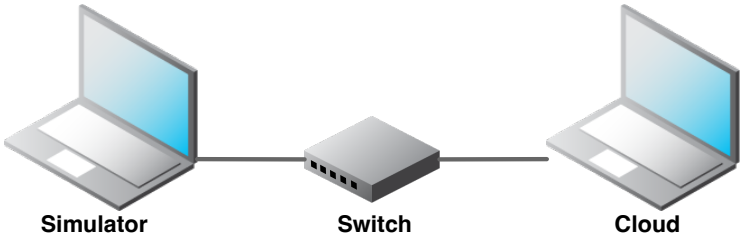

Figure 3.12: Physical setup for our evaluation.

3.4.3 Experiment Design

To answer our research questions, we use two computers as shown in Figure 3.12: one for running our simulator or the baseline tools, and one for running our case studies, i.e., the UDP/TCP and MQTT cloud systems described in Section 3.4.1. As discussed in Section 3.3.1, in our work, we assume that the simulated edge layer and the cloud layer are connected via an ideal network with minimally low transmission time and no loss. We create this ideal network using a single network switch (see Figure 3.12). This setup, while simple, provides a high-fidelity testbed for stress testing of cloud systems, as it routes all network traffic through a switch (ideal network) rather than a real-world network that may not be as reliable or predictable.

To simulate requests from edge devices (left laptop in Figure 3.12), we use a machine with a 2.5 GHz Intel Core i9 CPU and 64 GB of memory. For running the cloud (right laptop), we use a machine with a 2.3 GHz Intel Core i9 CPU and 32 GB of memory. We connect the two machines using an unmanaged NETGEAR GS308v3 Gigabit Ethernet switch. Below, we describe the experimental setup for RQ3.1-RQ3.4.

Table 3.3: Parameters for the RQ3.1 experiments.

duration	2s
time step	0.5s
# of IoT devices per edge device	100
# of edge devices per sim node	10
(IoT) Device.period	1
(IoT) Device.payload	8B
Edge Device.workload	0
memory	2G
CPU	4
Edge Device.speed	Max, 500, 250, 167, 125
Platform	Native & VM & CDC & UDC

3.4.3.1 Experimental Setup for RQ3.1

We conduct experiments to answer **RQ3.1** using the configuration parameters shown in Table 3.3. We choose a simulation duration of 2s and a time step of 0.5s (i.e., four steps in each simulation). Our preliminary experiments showed that the simulation duration of 2s with four steps are sufficient to observe the behaviour of the simulator and the differences among different simulators' configurations. For the device payload size, we follow the recommendation of Cheetah Networks, and set it to 8B. We set the period of each IoT device to 1 to maximize the number of messages sent in each time step. We assign four CPU cores and 2G of memory to each virtual machine and constrained Docker hosting edge devices, in accordance with the computational resources typical of low-cost edge devices [80]. We vary the number of simulation nodes from 8 to 12, corresponding to simulating 8,000 to 12,000 IoT devices. This range, in addition to addressing RQ3.1, enables us to compare our simulators' performance with the results reported in Section 3.2. There, we employed two

state-of-the-art edge-to-cloud simulation tools, EMU-IoT and Fogbed, to simulate 10,000 IoT devices.

For edge devices, we eliminate edge-computing time by setting `edge.workload` to zero, noting that our goal is maximizing communication with the cloud. We set the number of IoT devices per edge device to 100 and the number of edge devices per simulation node to 10. That is, each simulation node captures the behaviour of 1000 IoT devices. For the purpose of RQ3.1 experiments, we vary the number of simulation nodes to determine which simulator platform enables capturing more IoT devices. Bundling 1000 IoT devices into each simulation node allows us to increase the simulator load by 1000 as we increase the number of simulation nodes. For the speed of edge devices, we insert a gap time of 0ms, 1ms, 2ms, 3ms and 4ms between the consecutive messages sent, resulting in the edge device speed to be set, respectively, to MAX, 500, 250, 167 and 125. Note that further decreasing the speed (i.e., increasing the gap time) leads to inconsistency in our experimental setup since each edge device needs to be able to send 100 messages in a single time step. If the gap time is 5ms, then the total gap time for 100 messages is 0.5s. This is the same as the time step, so there is no time left for sending messages. We execute each configuration of our simulator for four platform options: Native, Virtual Machine (VM), Constrained Docker (CDC) and Unconstrained Docker (UDC).

In total, to answer **RQ3.1**, we perform 100 experiments (# of platforms = 4) $\times$ (# of speeds = 5) $\times$ (# of sim nodes = 5) . We repeat each experiment 10 times to account for random variation.

Table 3.4: Parameters for the RQ3.2 experiments.

Edge Device.speed	MAX
sim node(SN).offsetRange	0, 20%, 40%, 60%, 80%

3.4.3.2 Experimental Setup for RQ3.2

To address **RQ3.2**, we examine how different values for the `offsetRange` parameter of simulation nodes would impact the scalability of our simulator. We use the best-performing platform from RQ3.1 alongside the UDP/TCP cloud for testing. Table 3.4 shows the parameters specific to RQ3.2. For the parameters that are not listed in Table 3.4, we use the same values as those in RQ3.1 (see Table 3.3). We vary the `offsetRange` attribute of simulation nodes by using the following values: 0%, 20%, 40%, 60%, and 80%. We set the value of the `speed` attribute to MAX. The impact of changing `speed` is addressed by the RQ3.1 results. For the RQ3.2 experiments, we specifically study the impact of changing the value of `offsetRange` on the performance of the simulator. To illustrate the impact of `offSetRange`, we set the number of simulation nodes to range from 15 to 19. We increase the number of simulation nodes compared to those for RQ3.2 to demonstrate how offset variables enhance simulation capacity.

In total, to answer **RQ3.2**, we perform 50 experiments (# of platforms = 2) $\times$ (# of offsetRange = 5) $\times$ (# of sim nodes = 5). We repeat each experiment 10 times to account for random variation.

3.4.3.3 Experimental Setup for RQ3.3

To answer **RQ3.3**, we employ JMeter and Locust as baseline tools for conducting stress testing with the UDP/TCP cloud. For our simulator, we use the same setup as that used for RQ3.1 (see Table 3.3). In line with RQ3.1, we execute each baseline tool for a duration of 2 seconds. To achieve this, we set the `duration` parameter of JMeter and the `run-time` parameter of Locust to 2. Neither of the tools explicitly support the concept of time step. We implement a custom variable in the scripts running the tools to mimic the time step and set that variable to 0.5s. For both tools, we represent each edge device as a single `thread`. While we can organize the threads representing edge devices into groups, such thread groups cannot act as simulation nodes because they cannot run independently on virtual machines or Dockers. Thus, for both baselines, we can only run all the edge devices (threads) as native on the host machine. For JMeter, we employ the `loop count` parameter as a representation of the number of IoT devices associated with an edge device. We set `loop count` to 100. For Locust, we initiate an instance of a created class inherited from the original `user` class provided by Locust to represent an IoT device, and configure the `user count` (i.e., the number of instantiations) to be 100. For both tools, we use the same payload as that used for RQ3.1, i.e., 8B. We initially use the same range for simulation nodes as in RQ3.1, i.e., 8 to 12. We then modify the number of simulation nodes to 4 for JMeter and Locust, while increasing it to 15 for our simulator. This aims to determine the maximum number of nodes that JMeter, Locust, and our simulator can handle without packet drops on the simulator side.

In total, to answer **RQ3.3**, we perform 18 experiments (# of baselines = 2) $\times$ (# of

Table 3.5: Parameters for the RQ3.4 experiments with the MQTT cloud.

duration	10m
time step	0.1s
(IoT) Device.period	1
(IoT)Device.payload	166B
Edge Device.workload	0
sim node (SN).offsetRange	0
memory	4G
CPU	4

sim nodes = 9) with the baseline tools, perform 2 experiments (# of platforms = 2) × (# of sim nodes = 1) and reuse 2 experiments (# of platforms = 2) × (# of sim nodes = 1) with our simulator. We repeat each experiment 10 times to account for random variation.

3.4.3.4 Experimental Setup for RQ3.4

To address **RQ3.4**, we use our simulator to determine the service capacity of both the UDP/TCP cloud and the MQTT cloud systems. For the UDP/TCP cloud, we consider the experimental setup used for RQ3.1 (see Table 3.3), except that we consider only the platform option that, based on the results of **RQ3.1**, is capable of successfully executing on a single machine and sending all messages to the UDP/TCP cloud.

Table 3.5 shows the parameters used in our experiments with the MQTT cloud for **RQ3.4**. Recall from Section 3.4.1 that the MQTT cloud case study represents a connected vehicle system where one edge device is installed on each vehicle. We set the simulation duration to 10 minutes, which is close to the average transport time for vehicles (e.g., trucks) to travel per trip (from one location to another location by motor vehicles) [90].

We set the time step to 0.1s for our simulation, which corresponds to a data transmission rate of 10Hz. This decision aligns with existing research in connected vehicles [195], which suggests that data transmission rate typically ranges from 1 to 10Hz, reflecting real-world scenarios. That is, vehicles send one to ten messages per second to roadside infrastructures or cloud systems. Specifically, a frequency of 10Hz is often used for safety messages and location data exchanges within cooperative awareness services [195].

Each packet in this case study contains all the vehicle's information, i.e., Vehicle ID, GPS location, speed, odometer, and fuel-tank temperature. The payload of each IoT device is configured as a JSON string. The payload size is on average 166B and varies slightly (standard deviation of ≈4 bytes) depending on the specific values included in the JSON string. In our setup, we set the period of IoT devices to 1, indicating that they generate data at a frequency of one per simulation step; this is to maximize the number of messages sent in each time step. We set the `edge.workload` parameter to 0 because edge devices in this system do not perform edge computing tasks. To simulate the behaviour of edge devices running on vehicles, we allocate 4 CPU cores and 4G of memory to the platform with constraint computational resources. These settings align with the current embedded hardware commonly used in vehicles. For the rest of the parameters not included in Table 3.5, we use the same values as those in RQ3.1.

To determine the capacity of each of our case-study cloud systems, we vary the number of simulation nodes from 29 to 34 and find the threshold at which packet loss is observed on the cloud side.

For **RQ3.4**, we reuse 10 experiments (# of platforms = 2) × (# of sim nodes = 5)

from RQ3.1 with the UDP/TCP cloud and perform 12 experiments (# of sim nodes = 6) × (# of platforms = 2) with the MQTT cloud. Each experiment is repeated 10 times to account for randomness.

3.4.4 Metrics

We measure two main metrics in our experiments: (1) the number of packets that are dropped during the simulation, and (2) the packet transmission time, i.e., the amount of time it takes for the packets to be transmitted from the simulator to the cloud and be processed by the cloud. In the setup of Figure 3.12, packets can be dropped (1) on the simulator side when the simulator fails to send all the messages it is expected to send, (2) on the switch (network) side when the switch fails to handle the traffic that needs to pass through it, e.g., due to congestion, or (3) on the cloud side, when the cloud fails to receive all the packets that have reached its host computer, or fails to send responses for the packets it has received. We separately measure packet drop at the three above points and refer to them respectively as SimDrop, NetDrop and CloudDrop. Note that in order to measure these values, we have to keep track of packet counts at the level of the simulator, the cloud and also the network adapters on both host machines. To count the number of packets sent from and received by the network adapters of the host machines of the simulator and the cloud, we use the Wireshark tool [210] which is the world's most widely used network protocol analyzer. Using Wireshark, we have confirmed that, in our experiments, the packet drop at the network adaptors is zero or negligible (less than five

packets out of hundreds of thousands). In addition, due to our utilization of a switch with a data transfer rate of up to 1 Gbps per port, capable of managing the transmission of more than $8,000$ simulation nodes per step without surpassing its capacity, to connect the computers, the packet drop for the network is zero as well. Hence, we do not report NetDrop as it is always zero for our experiments. The packet drop values measured at the simulator (i.e., SimDrop) and at the cloud (i.e, CloudDrop) as well as packet transmission time (as defined above) are thus the only measures that are of interest for capturing the behaviour of the simulator and the cloud.

Among these measures, SimDrop determines if the simulator is able to scale and that it does not fail under its own load. If there is packet drop by the simulator, it means that the simulator is overwhelmed. The CloudDrop and the packet transmission time (TransTime, for short) determine how well the cloud is able to handle the messages it receives from the edge.

3.4.5 Results

3.4.5.1 RQ3.1. (Simulator Configuration)

Table 3.6 shows the average values for SimDrop, CloudDrop, and packet transmission time obtained by running our simulator using the parameters in Table 3.3. We perform RQ3.1 experiments for 8 to 12 simulation nodes that correspond to simulating $8,000$ to $12,000$ IoT devices. By considering a range of simulation nodes, we validate the consistency of

our results, ensuring that the results remain robust for different numbers of simulated IoT devices. As discussed in Section 3.4.3.1, for RQ3.1, we run our simulator using four different platform options, i.e., Native, VM, CDC and UDC. In the Native option, we run all the edge devices in parallel on the simulator host machine without grouping them into separate simulation nodes. In contrast, when we use VM, CDC, UDC, we group every ten edge devices into a simulation node and run each simulation node on a separate VM, CDC or UDC. For each VM and each CDC, we use the CPU and memory sizes specified in Table 3.3.

Table 3.6: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.1. The values highlighted grey in the table are the results used to answer RQ3.4 for the UDP/TCP cloud system.

#SN	Speed	MAX	500	250	167	125
			CDC			
	SimDrop	0	0	0	0.2	0
	CloudDrop	2314.5	0	0	0	0
	TransTime(ms)	6.61	0.49	0.74	0.52	0.29
			UDC			
	SimDrop	0	0	0	0	0
	CloudDrop	2805.5	0	0	0	0
	TransTime(ms)	7.05	0.48	0.61	0.46	0.31
12			**VM**			
	SimDrop	42.6	3254.4	2922.2	4383.9	6294.2
	CloudDrop	28.1	0	0	0	0
	TransTime(ms)	0.47	0.44	0.27	0.28	0.28
			Native			
	SimDrop	25051.1	25563.8	25958.1	26089.8	26312.4
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.35	0.26	0.26	0.26	0.26

(To be continued)

Table 3.6: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.1. The values highlighted grey in the table are the results used to answer RQ3.4 for the UDP/TCP cloud system.

#SN	Speed	MAX	500	250	167	125
			CDC			
	SimDrop	0	0	0	0	0
	CloudDrop	3246.1	70.5	0	0	0
	TransTime(ms)	10.57	4.31	1.32	0.33	0.28
			UDC			
	SimDrop	0	0	0	0	0
	CloudDrop	3741.2	0	0	0	0
	TransTime(ms)	8.11	1.30	0.87	0.37	0.30
11			**VM**			
	SimDrop	71	119.10	1272.9	2407.3	4410.3
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.38	0.28	0.28	0.28	0.28
			Native			
	SimDrop	21022.4	21543.4	21980.3	22060.8	22599.6
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.26	0.26	0.28	0.33	0.30
			CDC			
	SimDrop	0	0	0	0	0
	CloudDrop	1517.2	0	0	0	0
	TransTime(ms)	9.01	1.74	0.70	0.29	0.27
			UDC			
	SimDrop	0	0	0	0	0
	CloudDrop	296.6	0	0	0	0
	TransTime(ms)	3.52	1.00	0.61	0.34	0.29
10			**VM**			
	SimDrop	69.1	423.7	178.3	710.1	2322.9
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.31	0.28	0.28	0.50	0.28

(To be continued)

Table 3.6: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.1. The values highlighted grey in the table are the results used to answer RQ3.4 for the UDP/TCP cloud system.

#SN	Speed	MAX	500	250	167	125
			Native			
	SimDrop	17252.6	17546.1	18103	18128.8	18537
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.26	0.26	0.29	0.32	0.26
			CDC			
	SimDrop	0	0	0	0	0
	CloudDrop	768	0	0	0	4.3
	TransTime(ms)	4.71	1.38	0.62	0.28	0.62
			UDC			
	SimDrop	0	0	0	0	2.2
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.62	0.61	0.51	0.30	0.28
9			**VM**			
	SimDrop	0	107.7	79.6	50.4	844.8
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.29	0.26	0.26	0.30	0.28
			Native			
	SimDrop	13239.5	13834	13951.9	14209.4	14136.1
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.26	0.28	0.28	0.26	0.26
			CDC			
	SimDrop	0	0	0	0	0
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.99	0.98	0.35	0.28	0.26
			UDC			
	SimDrop	0	0	0	0	0
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.41	0.49	0.41	0.29	0.27
8						

(To be continued)

Table 3.6: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.1. The values highlighted grey in the table are the results used to answer RQ3.4 for the UDP/TCP cloud system.

#SN	Speed	MAX	500	250	167	125
			VM			
	SimDrop	26.8	0.9	0	0	100.7
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.27	0.27	0.27	0.26	0.30
			Native			
	SimDrop	9323.7	9858.3	9935.7	10176.4	10168
	CloudDrop	0	0	0	0	0
	TransTime(ms)	0.26	0.26	0.30	0.26	0.29

* VM: Virtual Machine

* CDC: Constrained Docker Container

* UDC: Unconstrained Docker Container

As shown in Table 3.6, there is substantial packet drop on the simulator side when we use the Native option. This is because, although VM- and Docker-based solutions require additional resources, the simulated edge devices running natively share the host's resources without isolation, leading to resource contention and scheduling delays. This indicates that in this architecture where edge devices are not grouped into simulation nodes, the simulator is simply unable to send all the packets it is supposed to send. This highlights the importance of the hierarchical structure provided by the `Simulation Node` concept in IoTECS. Similarly, the VM option, while generating SimDrop values lower than those of the Native option, is unsuitable for stress testing as it still leads to non-negligible packet drop on the side of the simulator.

For the simulation configurations executed on CDC and UDC, SimDrop values are zero

(or negligible). For these configurations, the cloud side drops packets, particularly when edge devices send their packets at once and without having any gap time in between (i.e., when the edge device speed is MAX). By inserting a gap time on the side of edge devices (i.e., lowering the speed), the cloud will receive the packets at a lower rate and can avoid packet drop.

Note that transmission-time values for the Native and VM should be discarded. This is because, for these options, there is a high packet drop at the simulator, and hence, considerably fewer packets are transmitted from the simulator to the cloud and vice versa. Therefore, the transmission times are computed for a smaller portion of packets, leading to unrealistically small values. For the CDC and UDC options where the simulator is able to complete its task without packet drop, the transmission time decreases as we reduce the speed of the edge devices. This trend is expected, since by reducing the speed, we avoid the cloud side from being overwhelmed, hence reducing the packet transmission time.

The answer to **RQ3.1** is that our simulator can be tuned such that it can, on a conventional laptop, simulate the sending of $24,000$ packets per second without packet drop. Further, our results show the importance of grouping edge devices into simulation nodes to manage resource usage on the simulator's host machine and thus enable the simulator to scale as much as possible. In our experimental setup, containerization (through Dockers) turned out to be the best option for scaling. Among the main benefits of IoTECS are its support for simulation nodes and its ability to abstract from platform details so that engineers can explore different platform options.

Table 3.7: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.2.

#SN	OffsetRange	80%	60%	40%	20%	0
			CDC			
	SimDrop	0	0	0	0	307.8
	CloudDrop	2971.5	3161.2	3363.3	3503.0	3697.1
19	TransTime(ms)	8.13	8.39	8.65	8.74	10.04
			UDC			
	SimDrop	0	0	0	0	369.4
	CloudDrop	2702.3	3076.7	3178.0	3590.3	3486.4
	TransTime(ms)	8.49	8.75	9.02	9.17	10.54
			CDC			
	SimDrop	0	0	0	0	263.5
	CloudDrop	2642.1	2810.7	2990.2	3114.8	3308.9
18	TransTime(ms)	7.89	8.21	8.40	8.29	9.48
			UDC			
	SimDrop	0	0	0	0	316.3
	CloudDrop	3136.0	2904.2	3480.8	3236.9	3576.1
	TransTime(ms)	8.41	8.16	8.68	8.70	9.94
			CDC			
	SimDrop	0	0	0	0	206.3
	CloudDrop	2793.4	2971.7	3161.4	3293.5	3471.1
17	TransTime(ms)	7.43	7.31	7.77	7.84	8.91
			UDC			
	SimDrop	0	0	0	0	247.6
	CloudDrop	3024.0	2996.2	2908.5	3261.2	3362.6
	TransTime(ms)	7.81	8.06	8.31	8.23	9.35
			CDC			
	SimDrop	0	0	0	0	161.3
	CloudDrop	2492.2	2651.3	2820.6	2938.2	3115.8
16	TransTime(ms)	6.98	7.11	7.31	7.39	8.35
			UDC			
	SimDrop	0	0	0	0	193.6
	CloudDrop	2923.8	2704.0	3125.9	3328.3	3458.2
	TransTime(ms)	7.21	7.43	7.67	7.75	8.76
			CDC			
	SimDrop	0	0	0	0	153.4
	CloudDrop	2336.5	2485.7	2644.4	2754.6	2916.4
15	TransTime(ms)	6.66	6.79	7.02	6.93	7.81
			UDC			
	SimDrop	0	0	0	0	184.1
	CloudDrop	2182.5	2465.7	2528.0	2644.7	3524.9
	TransTime(ms)	6.97	7.09	7.31	7.27	8.20

3.4.5.2 RQ3.2. (Scalability)

We address RQ3.2 by analyzing how the `offsetRange` parameter affects our simulator's scalability. We note that higher values of `offsetRange` mean that the edge devices' offsets can vary within a larger range. Thus, the edge devices' start times are more spread out within a time step. This temporal distribution helps mitigate sudden spikes in outgoing traffic that may otherwise overload the system. The experiments for RQ3.2 are performed for 15 to 19 simulation nodes that correspond to simulating $15,000$ to $19,000$ IoT devices. The results, presented in Table 3.7, show that by increasing the value of `offsetRange`, the simulator is able to send more messages successfully. Specifically, as shown in the table, SimDrop becomes zero when the `offsetRange` exceeds 20%. In other words, when we force all edge devices to start at the beginning of the time step, i.e., all offset values are zero, our simulator incurs some drop for all the simulation nodes shown in Table 3.7. However, when the offset values can vary within a range, the simulator successfully simulates up to $19,000$ IoT devices within half a second. This is approximately 1.3 times higher than the number of IoT devices our simulator could handle in RQ3.1 when the start time of edge devices is set to zero.

The answer to RQ3.2 is that by introducing randomness in the start time of edge devices, our simulator can scale to at least 1.3 times more IoT devices compared to situations where randomness is excluded.

Table 3.8: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.3 (Baselines).

#SN	Baselines	JMeter	Locust
12	SimDrop	11461.7	9985.5
	CloudDrop	1.8	1.8
	TransTime(ms)	0.67	0.42
11	SimDrop	4594.6	3984.4
	CloudDrop	4.4	4.4
	TransTime(ms)	0.46	0.30
10	SimDrop	2789.6	2421.2
	CloudDrop	0	0
	TransTime(ms)	0.40	0.36
9	SimDrop	687.6	594.9
	CloudDrop	0	0
	TransTime(ms)	0.33	0.36
8	SimDrop	559.4	484.1
	CloudDrop	0	0
	TransTime(ms)	0.35	0.33
...			
5	SimDrop	213.2	189.7
	CloudDrop	0	0
	TransTime(ms)	0.31	0.29
4	SimDrop	0	0
	CloudDrop	0	0
	TransTime(ms)	0.28	0.26

3.4.5.3 RQ3.3. (Comparison with Baselines)

To answer **RQ3.3**, we apply our baseline tools Apache JMeter and Locust to the UDP/TCP cloud using the settings discussed in Section 3.4.3.3. The average SimDrop, CloudDrop and packet-transmission time results obtained from the baseline tools are shown in Table 3.8. We compare the results obtained from the baselines, shown in Table 3.8, with the results from our simulator, shown in Table 3.6. The results were obtained by running the baselines

Table 3.9: Average packet drop and transmission time values obtained by ten runs of the experiments of RQ3.3 (Our Simulator).

#SN	Our Simulator	CDC	UDC
15	SimDrop	153.4	184.1
	CloudDrop	2916.4	3524.9
	TransTime(ms)	7.81	8.20
14	SimDrop	0	0
	CloudDrop	2834.1	3116.4
	TransTime(ms)	6.53	7.41

and our simulator under identical settings as detailed in Section 3.4.3.3. As Table 3.8 shows, there is significant packet drop on the simulator side in all the experiments conducted with JMeter and Locust. Compared to the results in Table 3.6, both JMeter and Locust exhibit lower SimDrop values than those of the Native platform. Although the baselines perform better than the Native platform, both perform significantly worse than the CDC, UCD, and VM platforms.

To determine the number of simulation nodes our simulator can handle without packet drops on the simulator side, we conduct extended experiments where we gradually increase the number of simulation nodes until we observe non-zero SimDrop. Table 3.9 shows that SimDrop occurs in our simulator when we have 15 or more simulation nodes. In other words, our simulator can successfully simulate up to 14 simulation nodes in one time step (i.e., 0.5s), which is equivalent to $28,000$ IoT devices per second. For JMeter and Locust, we decrease the number of simulation nodes until the value of SimDrop becomes zero or negligible. The part below the row of "..." in Table 3.8 shows that both JMeter and Locust lead to non-negligible SimDrop with 5 or more simulation nodes. This means that JMeter and Locust can successfully simulate up to 4 simulation nodes in a single time step,

corresponding to $8,000$ IoT devices per second. Therefore, our simulator can simulate 3.5 times more IoT devices than JMeter and Locust.

We note that the transmission-time and CloudDrop values are not used for comparison. These values are relatively low or negligible for the baselines. This is because, due to the high SimDrop, the baselines can transmit only a small portion of their packets. This leads to artificially lower transmission time and CloudDrop values when compared to the CDC and UDC platforms, which can transmit all their packets. Hence, these values cannot be used as a reliable basis for comparison.

The answer to **RQ3.3** is that our simulator, when utilizing the CDC and UDC platforms, outperforms the baseline tools, namely JMeter and Locust, by successfully simulating 3.5 times more IoT and edge devices compared to what can be achieved using the baseline tools.

3.4.5.4 RQ3.4. (Usefulness)

We answer RQ3.4 using both the UDP/TCP cloud and the MQTT cloud systems. Regarding the UDP/TCP cloud, we are interested in the the maximum number of IoT devices that our UDP/TCP cloud can serve without significant CloudDrop. To find this number, we recall the RQ3.1 results highlighted grey in Table 3.6. These results show that the UDP/TCP cloud reaches its limit, i.e., starts dropping packets, when the number of simulation nodes is 9 or higher for the CDC platform and 10 or higher for the UDC platform. Therefore, the threshold for the service capacity of our UDP/TCP cloud is 8 simulation nodes, since it is the highest number of simulations nodes that does not lead to significant

Table 3.10: Average packet drop and packet transmission time values obtained by ten runs of the experiments of RQ3.4 (stress testing MQTT system).

Metrics	#Vehicles=340	#Vehicles=330	#Vehicles=320	#Vehicles=310	#Vehicles=300	#Vehicles=290
			CDC			
CloudDrop	17050.8	16431.4	0	0	1253.1	0
TransTime(ms)	6.37	6.49	5.94	5.78	5.63	5.66
			UDC			
CloudDrop	16538.1	16153.5	0	0	970.2	0
TransTime(ms)	6.51	6.33	5.87	5.61	5.27	5.11

CloudDrop for both platforms. This means that our UDP/TCP cloud can handle — without packets being dropped by the cloud — $8,000$ IoT devices sending messages every 0.5s or $16,000$ IoT devices sending packets every second.

Table 3.10 presents the CloudDrop and transmission-time values obtained by applying our simulator to the MQTT cloud. Since the SimDrop values are consistently zero, these values are not included in the table. The results shown in Table 3.10 are obtained when the number of simulated vehicles is set to 290, 300, 310, 320, 330 and 340, respectively. Recall from Section 3.4.3 that for the MQTT cloud, each vehicle acts as one edge device with one associated IoT device, and further, each vehicle sends one packet with a payload of 166B every 100ms. As shown in Table 3.10, CloudDrop is significant when the number of vehicles is more than 320. Note that even for 300 vehicles, CloudDrop is non-zero, i.e., $1,253.1$ and 970.2 when we used our simulator with the CDC and UDC platforms, respectively. But this level of CloudDrop is 0.06% (negligible) considering that we send a total number of $1,800,000$ packets in the experiments with the MQTT cloud. Hence, CloudDrop is considered 0 or negligible when the number of vehicles is less than or equal to 320. The results in Table 3.10 show that the MQTT cloud can successfully handle

messages coming from up to 320 vehicles at every $100ms$ or messages coming from up to $3,200$ vehicles per second. We note that the packet payload for the MQTT case study is 20 times larger than the packet payload for our UDP/TCP case study. Due to the packet size differences and other differences between these two systems, the number of IoT devices that they can handle are different.

The answer to **RQ3.4** is that our simulator successfully determines the service capacity of our case studies — the UDP/TCP cloud and the MQTT cloud. Specifically, the UDP/TCP cloud has the ability to accommodate a load of $16,000$ IoT devices each sending one message every second, while the MQTT cloud can serve $3,200$ vehicles each sending one message every second.

3.4.6 Limitations and Validity Considerations

3.4.6.1 Limitations

Below, we outline the limitations of our approach, organizing them into four aspects.

Network protocols. IoTECS currently supports three network communication protocols: MQTT, TCP and UDP. MQTT is widely recognized as the predominant communication protocol for IoT systems [194], while TCP and UDP are among the most extensively employed transport layer protocols for networking, serving communication needs across the Internet [114]. We anticipate that these protocols will adequately cover the vast majority of communication scenarios between edge devices and cloud systems. If additional protocols are required in the future, updates to IoTECS and further empirical evaluation would

be necessary. The impact of incorporating new protocols on the language itself would be minor, with only Rule $\epsilon 7$ in Section 3.3.2 being affected. Nevertheless, it would also be necessary to develop appropriate new Xtend rules for code generation and conduct further experimentation with the new protocols.

Immutability. Currently, simulators derived from IoTECS specifications are immutable, meaning their configuration cannot be changed at runtime. To modify the simulator, one must alter the specification, then re-generate and re-execute the simulator. Introducing support for changing a simulator's parameters such as speed, offsetRange, duration and time step would enhance the usability and applicability of our simulators.

Analysis. The output metrics, such as SimDrop, CloudDrop, NetDrop, and TransTime, used in our evaluation of Section 3.4, are currently obtained only at the end of a simulation run. This approach to gathering metrics does not support online monitoring, especially during the typically long testing periods for IoT systems. Our work could be expanded to enable the collection of metrics in real-time during simulator execution through additional engineering effort, without affecting our overall approach.

Simulation. Currently, IoTECS specifications support the introduction of uniform latencies in packets sent by simulated edge devices. Incorporating a feature that enables custom latencies to be introduced in specific generated packets to further would increase the flexibility of defining communication patterns between edge devices and IoT cloud systems.

As evidenced from our industry partner, the above limitations have not impeded the adoption of IoTECS in practice, and IoTECS remains in active use by our industry partner.

That being said, prospective adopters of IoTECS need to take note of the following practical implications of the current limitations:

- Features: IoTECS currently only supports UDP, TCP, and MQTT. Extending support to other protocols necessitates language extensions. Similarly, enabling customization of latency patterns on a per-packet or per-group basis, if required, involves improvements to the language.
- Usability: The behaviour of a simulator remains fixed at runtime. As such, users must re-generate the simulator whenever they wish to adjust simulation parameters. Further, simulation results are generated only upon completion rather than being streamed progressively as they become available.

3.4.6.2 Validity Considerations

Construct and external validity are the most relevant dimensions of validity for our evaluation.

Construct Validity: The main concern for construct validity is the reliable measurement of our evaluation metrics as defined in Section 3.4.4. These metrics are based on common practices in network and performance engineering. To accurately measure our metrics of interest, we use Wireshark [210] — a standard and widely used network protocol analyzer.

External Validity: We have used IoTECS to generate simulators for stress testing two industrial case studies, i.e., the UDP/TCP and MQTT cloud systems outlined in Section 3.4.1. The results obtained for both case studies indicate that IoTECS can effectively

meet the stress testing needs of real-world IoT cloud systems. This provides confidence about the broader applicability of IoTECS. That said, conducting further evaluations with additional case studies would contribute to enhancing external validity.

3.5 Summary

Stress testing is a systematic way to verify the scalability of cloud-based systems. In this chapter, we presented a lean simulation framework specifically designed for IoT cloud stress testing. This framework enables efficient simulation of large arrays of devices on the user layer interacting with the cloud layer. To simplify simulation construction for practitioners, we proposed a domain-specific language, named IoTECS, tailored for generating simulators from high-level model-based specifications. The language has been implemented using Xtext and Xtend and is publicly available [188].

IoTECS is the result of a collaborative effort with an industry partner, Cheetah Networks. The language has been used independently by our partner in various testing and demonstration campaigns.

Our empirical evaluation revolved around two industrial case studies: one related to IoT connected vehicles and the other to the monitoring and generation of analytics for IoT networks. We systematically examined different platforms for running IoTECS simulators, and concluded that peak performance is achieved by containerizing groups of edge devices with Dockers, rather than using virtual machines or running (simulated) edge devices directly on host machines. Subsequently, we assessed the service capacity of our industrial

case studies using IoTECS simulators, and demonstrated that IoTECS simulators can handle 3.5 times more IoT and edge devices compared to two industrial stress-testing baseline tools, JMeter and Locust.

3.6 Implementation and Availability

We have implemented IoTECS using Xtext version 2.25.0 [64] and Xtend version 2.25.0 [63]. The IoTECS grammar and our code generation tool are publicly available [188]. Our online material include: (1) a complete description and implementation of IoTECS including an example of IoTECS specification, (2) scripts for running baseline tools (i.e., JMeter and Locust), and (3) the results of our experiments.

Chapter 4

A Self-Adaptive Framework to Scale the Network Layer

4.1 Introduction

Motivation. Enhancing the scalability of cloud-based systems to ensure they meet their Service Level Objectives (SLOs) and maintain reliability under varying workloads is a major challenge. Self-adaptation is a promising approach for addressing this challenge. The idea behind self-adaptation is that engineers take an existing system, specify its expected qualities and objectives as well as strategies to achieve these objectives, and build capabilities into the system in a way that enables the system to adjust itself to changes during operation [75].

When focusing on the scalability of the network layer in cloud-based systems, an adap-

tive approach to network management is essential to prevent anomalies and ensure efficient data flow. Many software-intensive systems can benefit from self-adaptivity. A particularly pertinent domain is software-defined networking (SDN) — a flexible network architecture that is prevalent in modern data centres and Internet of Things applications [62, 178, 190, 206]. A software-defined network provides centralized programmable control over distributed network resources, thereby allowing network operators to better manage network performance and to react in real time to events in the network.

Notably, many networks are prone to congestion when there is a burst in workload. For instance, in an emergency management system, such bursts may occur when a disaster situation, e.g., a flood, is unfolding. Taking advantage of the software programmability of SDN controllers, our ultimate goal in this chapter is to develop a self-adaptive approach for resolving network congestion, thereby enhancing network scalability to ensure that SLOs are met under high workloads.

Problem statement. Self-adaptation has been studied for many years [45, 100, 144, 163, 184]. The existing self-adaptation approaches include model-based [58, 179], control-based [68], requirements-based [10], and more recently, learning-based [77] solutions. At the heart of all self-adaptation solutions, there is a planning step that generates or determines an adaptation strategy to adapt the running system once an anomaly (i.e., an SLO violation) is detected [75]. Adaptation strategies may be composed of fixed and pre-defined actions identified based on domain knowledge, or they may be new behaviours or entities introduced at run-time [184]. Irrespective of the type of adaptation, the knowledge about how to generate adaptation strategies is often concentrated in the planner: The running system is merely modified by the planner to be able to handle an anomaly as seen in a

specific time and context; the running system is not necessarily improved in a way that it can better respond to future anomalies on its own. As we illustrate in our motivating example of Section 4.2, having to repeatedly invoke the planner for each adaptation can be both expensive and ineffective. The main problem that we address in this chapter is as follows: The self-adaptation planner should attempt to improve the control logic of the running system such that the system itself would learn how to steer clear of future anomalies, without triggering self-adaptation too frequently. The need for self-adaptation is never entirely eliminated, especially noting that the environment changes over time. Nonetheless, a system augmented with such a self-adaptation planner is likely to be more efficient, more robust to changes in the environment (e.g., varying network requests), and less in need of adaptation intervention. Furthermore, existing self-adaptation research focuses on modifying a running system via producing individual and concrete elements, e.g., configuration values [97]. In contrast, we take a generative approach [67], modifying the logic of the running system which in turn generates the concrete elements.

Contributions. We propose a generative self-adaptation framework, named GenAdapt, that uses genetic programming (GP) [109, 169] to realize the above-described vision of self-adaptation. GenAdapt builds on the well-known MAPE-K loop [105, 144] to incrementally enhance the control logic of the running systems. The idea is that by incremental adaptations (i.e., evolving a running system's logic), the running system becomes more robust to changes in the environment and requires less frequent adaptation. GenAdapt requires as input a context-free grammar capturing the language in which the configurable (adaptive) fragment of a system's control logic has been expressed. It then employs genetic programming to incrementally evolve the system's control logic so that the system responds better

to environment changes observed over time. To ensure the continuity of learning over time and to evolve the control logic in a way that better fits the environment in which the system is operating, at each round of self-adaptation, we maintain in the MAPE-K knowledge base a set of best solutions (alternative control-logic implementations) computed by GP. These solutions are used to partially bootstrap GP in the future rounds of self-adaptation, thereby informing these future rounds about the candidate solutions that best fit the environment uncertainty in the past.

In this chapter, we instantiate GenAdapt to address congestion control for SDNs. In an SDN, network control is transferred from local fixed-behaviour controllers distributed over a set of switches to a centralized and programmable software controller [85]. More specifically, we address the self-adaptation of SDN data-forwarding algorithms in order to resolve network congestion in real time. We implement GenAdapt to enhance the programmable controller of an SDN. Noting that GenAdapt relies on the MAPE-K loop, it performs the following tasks periodically: (i) monitoring the SDN to check if it is congested and generating a model of the SDN to be used for congestion resolution; (ii) applying GP to evolve the logic of the SDN data-forwarding algorithm such that congestion is resolved and, further, the transmission delay and the changes introduced in the existing data-transmission routes are minimized; and (iii) modifying existing transmission routes to resolve the current congestion and updating the logic of SDN data forwarding to mitigate future congestion. Exploiting the global network view provided by an SDN for modifying the controller and resolving congestion in real time is not new. However, existing approaches adapt the network management logic using pre-defined rules [78]. In contrast, our approach uses GP to modify the data-forwarding control constructs in an evolving manner and without reliance

on fixed rules.

We have implemented GenAdapt into a tool, which we make publicly available [1]. As we describe in more detail in Section 4.5, the GenAdapt tool interacts with (i) the (software-programmable) controller of an SDN, (ii) a network emulator that simulates the network infrastructure, and (iii) a traffic generator that emulates different types of requests from devices and network users.

We evaluate GenAdapt on 26 synthetic and one industrial IoT network. The industrial network, which is the backbone of an IoT-based emergency management system [190], is prone to congestion when the volume of demand increases during emergencies. We compare our framework with two baselines: (1) a self-adaptive technique from the SEAMS community that attempts to resolve congestion by generating individual adaptations (i.e., individual data-transmission routes) without optimizing the SDN routing algorithms [190]; and (2) a standard data-forwarding algorithm from the network community that uses pre-defined rules (heuristics) to optimize SDN control at runtime and resolve congestion [49,51].

Our results indicate that GenAdapt successfully resolves congestion in all the 26 networks considered while the adaptive, but non-generative baseline fails to resolve congestion in four of the networks with probabilities ranging from 10% to 66%. In addition, compared to this baseline, GenAdapt reduces the average number of congestion occurrences, and hence, the number of adaptation rounds necessary. Compared to the baseline from the network community, GenAdapt is able to significantly reduce packet loss for the industrial network.

In addition, we empirically assess the impact of transferring to larger networks the

best control logic learned on smaller networks, when both the smaller and larger networks have the same topology (e.g., both are full graphs). We observe that for a given topology, bootstrapping GenAdapt with the best logic learned on a smaller network can significantly improve performance on a larger network in terms of the number of adaptation invocations, the amount of packet loss and overall duration the network remains congested. These results suggest the transferability of the learned logic over networks that share the same topology but have different numbers of nodes. Finally, we empirically demonstrate that the execution time of GenAdapt is linear in network size and the amount of data traffic over time, thus making it suitable for online adaptation.

The contributions of this chapter support the thesis contributions as follows:

Con2.1 I propose a generative self-adaptation framework, named GenAdapt, that uses genetic programming (GP) to adapt the control logic of the running system in such a way that the system itself would learn how to steer clear of future SLO violations, without triggering self-adaptation too frequently.

Con2.2 I apply GenAdapt to address network congestion of Software-Defined Networks (SDNs). SDN is widely utilized in cloud-based systems, particularly in modern data centers and IoT systems, due to its centralized, programmable control over distributed network resources [103]. My approach continuously learns and updates SDN data-forwarding logic to address network congestion while reducing the frequency of adaptation interventions over time.

Structure. The rest of this chapter is organized as follows: Section 4.2 motivates the chapter using an example. Section 4.3 provides an overview of our self-adaptation framework,

GenAdapt. Section 4.4 presents an instantiation of GenAdapt in the context of SDN data forwarding. Section 4.5 presents our tool support. Section 4.6 describes our evaluation. Section 4.7 summarizes the chapter.

4.2 Motivating Example

We motivate our approach through an example. Network-traffic management is about assigning flow paths to network requests such that the entire network is optimally utilized [3,9,28,152]. A flow path is a directed path of network links. Many network systems use a standard, lightweight data-forwarding algorithm, known as Open Shortest Path First (OSPF) [49,51]. OSPF generates a flow path for a data-transmission request by computing the shortest weighted path between the source and destination nodes of the request.

Figure 4.1(a) shows a network with five nodes (switches) and six links. Suppose that there are six requests, $r_1, \ldots, r_6$ from source node s_1 to destination node s_2, and that they arrive in sequence (first r_1, then r_2, ...) with a few seconds in between. These requests keep running after arrival. Further, all link weights are set to one. OSPF creates the flow path $s_1 \rightarrow s_2$ for every request. Assume that each flow utilizes 30% of the bandwidth of each link. Further, assume that we have a threshold of 80% for link utilization, above which we consider a link to be congested. Each link in our example then has enough bandwidth to transmit two flows before it is considered congested. Consequently, link $s_1 \rightarrow s_2$ will be congested after the arrival of r_3; see Figure 4.1(b). This leads to an invocation of the self-adaptation planning step to resolve the congestion. A common approach for congestion resolution is through combinatorial optimization, where

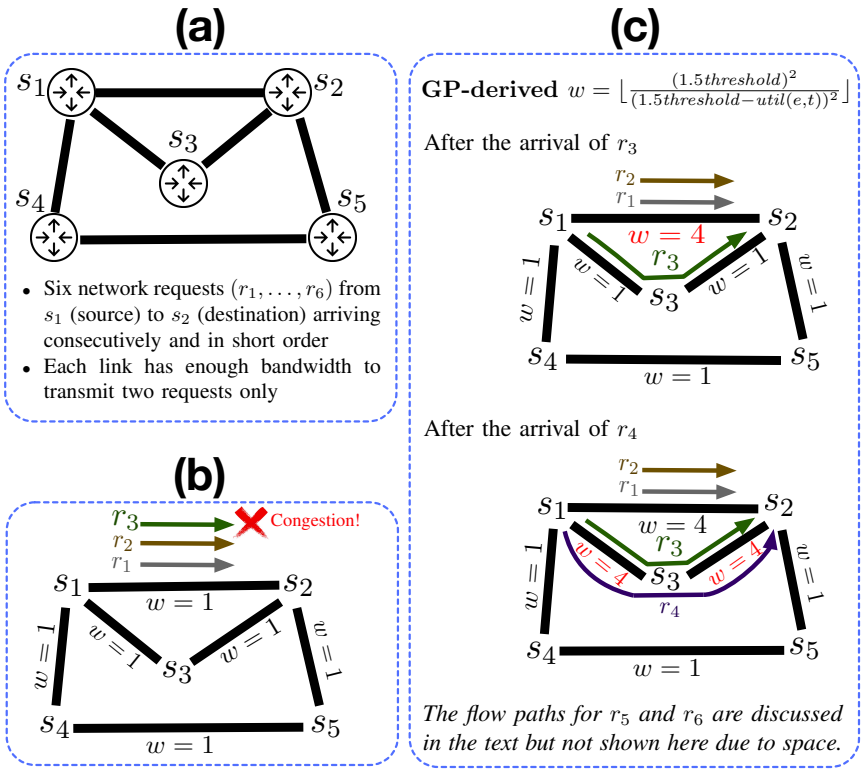

Figure 4.1: (a) A simple network alongside the description of network requests; (b) Output of a baseline data-forwarding algorithm; and (c) Output of data-forwarding after adaptation by our approach using genetic programming.

some flow paths are re-routed such that the data stream passing through each link remains below the utilization threshold [71, 83, 123]. This optimization can be done using various methods, e.g., graph optimization [71], mixed-integer linear programming [3], local search [70, 72], and genetic algorithms [190].

Optimization by re-routing flow paths has a major drawback. While optimizing at the level of individual flow paths may solve the currently observed congestion, doing so does not improve the logic of OSPF and thus does not contribute to congestion avoidance in the

future. For instance, in Figure 4.1(b), the arrival of r_3 will cause congestion. Re-routing r_3 over $s_1 \rightarrow s_3 \rightarrow s_2$ will resolve this congestion. But, upon the arrival of r_4, OSPF will yet again select the flow path $s_1 \rightarrow s_2$ for r_4 (since this is the shortest path between s_1 and s_2) just to find the $s_1 \rightarrow s_2$ link congested again. This in turn necessitates the self-adaptation planning step to be invoked for r_4 as well. In a similar vein, the arrival of r_5 and r_6 will cause congestion, prompting further calls to self-adaptation planning.

Our proposal is that, at each round of self-adaptation where a congestion is detected, one should focus on optimizing the logic of OSPF instead of optimizing flow paths. In this chapter, we optimize and update the link-weight formula that OSPF uses. The hypothesis here is that an optimized link-weight formula not only can resolve the existing congestion, but can simultaneously also make OSPF more intelligent towards congestion avoidance in the future, thus reducing the number of times that the self-adaptation planning step has to be invoked. Below, we illustrate our approach using the example of Figure 4.1.

We use genetic programming (GP) for self-adaptation planning, whereby we dynamically learn link-weight functions that not only help resolve an existing congestion but also help steer clear of future ones. Initially, and for requests r_1 and r_2, our approach does exactly as the standard OSPF would do, since there is no congestion. Upon the arrival of r_3 and the detection of congestion, i.e., the situation in Figure 4.1(b), our self-adaptation approach kicks in and automatically computes a link-weight formula to reroute some of the requests in order to resolve the congestion. For example, one possible formula that our approach could generate is : $\frac{(1.5threshold)^2}{(1.5threshold - util(e,t))^2}$. This formula would be stored in memory as a parse tree and called to compute the weight for each link when a new request arrives. In this formula, $util(e,t)$ is the utilization percentage of link e at time t, and *threshold*

is a constant parameter describing the utilization threshold above which the network is considered to be congested. The weight values generated for the links are first taken as absolute values and then rounded down to integers. This ensures that only positive integer weights are assigned to the links. In our example, $threshold = 0.8$, and for each link, $util(e, t)$ is 0.3 if one flow passes through e, and is 0.6 if two flows pass through e. Given that r_1 and r_2 are already routed through $s_1 \rightarrow s_2$, the weight of $s_1 \rightarrow s_2$ computed by this formula becomes $w = 4$; see Figure 4.1(c) after r_3's arrival. As a result, OSPF selects path $s_1 \rightarrow s_3 \rightarrow s_2$ for r_3 which successfully resolves the current congestion observed in Figure 4.1(b). This, however, does not increase the weights for $s_1 \rightarrow s_3$ and $s_3 \rightarrow s_2$, since these links are utilized at 30%, and their weights remain at 1. Once r_4 arrives, OSPF directs r_4 to $s_1 \rightarrow s_3 \rightarrow s_2$ as it is the shortest weighted path between s_1 and s_4. This will not cause any congestion, but the weights for $s_1 \rightarrow s_3$ and $s_3 \rightarrow s_2$ increase to $w = 4$ since these links are now utilized at 60%. Finally, OSPF will direct the last two requests r_5 and r_6 to the longer path $s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_2$, since, now, this path is the shortest weighted path between s_1 and s_2.

As shown above, using our GP-learned link-weight formula, OSPF is now able to manage flow paths without causing any congestion and without having to re-invoke the self-adaptation planning step beyond the single invocation after the arrival of r_3.

We note that there are several techniques that modify network parameters at runtime to resolve congestion in SDN (e.g., [78, 174]). These techniques, however, rely on pre-defined rules. For example, to make OSPF — discussed above — adaptive, a typical approach is to define a network-weight function [182]. This function can involve parameters that change at runtime and based on the state of the network. However, the structure of the function

is fixed. Consequently, the function may not be suitable for all networks with different characteristics and inputs. To address this limitation, we use GP to learn and evolve the function structure instead of relying on a fixed, manually crafted function. Our approach is generative and seeks to identify a higher-order structure, specifically a link-weight formula, that results in optimal adaptation solutions. The generated link-weight formula is used to determine the flow paths for incoming requests. In Section 4.6, we compare our GP-based approach with OSPF configured using an optimized weight function suggested by Cisco standards [49, 51]. As we show there, OSPF's optimized weight function cannot address the congestion caused by the network-request bursts in our industrial case study.

In this chapter, we focus on situations where congestion can be resolved by re-routing — in other words, when the network topology is such that alternative flow paths can be created for some requests. Given our focus, OSPF is a natural comparison baseline, noting that, in OSPF, congestion is resolved by re-routing. When alternative flow paths do not exist, congestion resolution can be addressed only via traffic shaping [78], i.e., via modifying the network traffic. Our approach does not alter the network traffic. We therefore do not compare against congestion-resolution baselines that use traffic shaping.

4.3 Framework Overview

Figure 4.2 shows an overview of GenAdapt — our generative self-adaptation framework. The main goal of GenAdapt is to adapt a running system by dynamically improving its control logic (e.g., a formula or a logical condition based on which the system makes a decision). GenAdapt aims to improve the designated logic in response to environmental

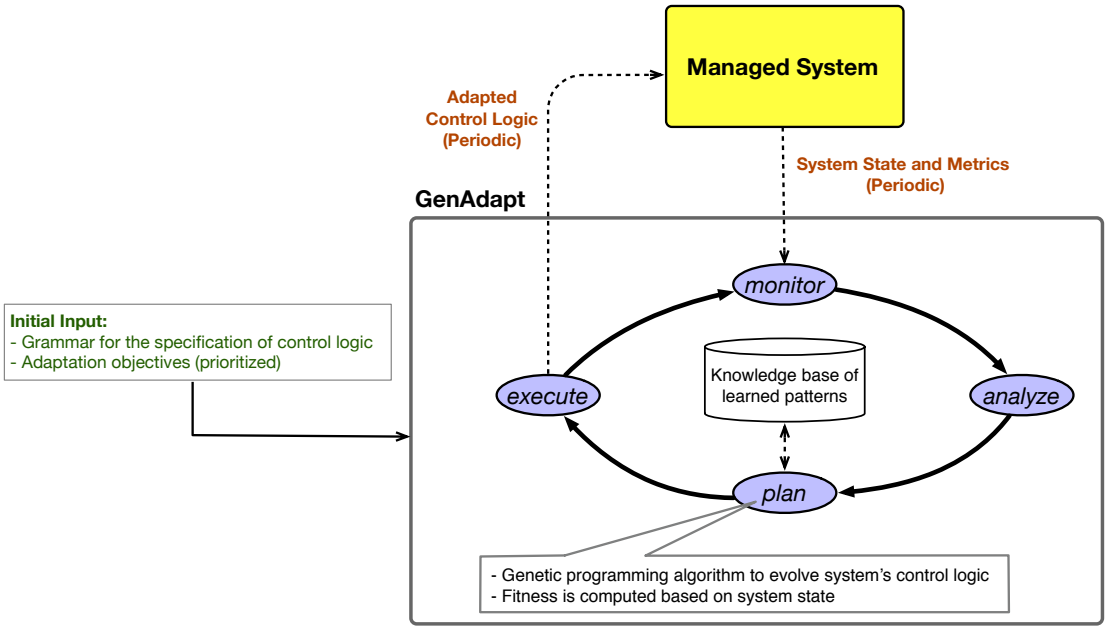

Figure 4.2: Overview of the GenAdapt Framework.

changes and find a trade-off between self-adaptation objectives. For example, in the context of SDN data forwarding, the control logic that GenAdapt is improving is the link-weight function used by the SDN's data-forwarding algorithm, as illustrated in Section 4.2. At each round of self-adaptation, GenAdapt evolves the link-weight function with the goal of satisfying a trade-off between three objectives related to the network's quality of service; these objectives will be defined and formalized in Section 4.4.2.

GenAdapt leverages MAPE-K [105,144] — the well-known self-adaptation control-loop. As shown in Figure 4.2, MAPE-K has four main steps:

(1) Monitoring the system and its environment: At this step, GenAdapt periodically receives the following information from the running system: (I) the state of the system at the current time, and (II) certain metrics that enable GenAdapt to monitor for the

occurrence of anomalies (i.e., violations of self-adaptation goals).

(2) Analyzing the information collected from the system and its environment and deciding whether adaptation is needed. At this stage, GenAdapt uses the system state and the metrics collected at the monitoring step to decide if some system goal has been violated, in which case the planning step needs to be triggered.

(3) Planning to adapt the system to address any violations observed during the previous step. This step employs a genetic programming (GP) algorithm to evolve the control logic of the running system such that when the system uses that evolved logic, the observed violation is no longer present, or its severity has been reduced. The GP algorithm employed in the planning step requires, in addition to the inputs received periodically from the running system, two further inputs: First, GP requires a context-free grammar specifying the language (template) for the control logic that is subject to modification. For example, we use a grammar defining valid mathematical formulas consisting of basic arithmetic operations and network parameters to specify a template for candidate link-weight functions in SDN data forwarding. Second, GP requires a set of prioritized adaptation objectives. The objectives should be defined so that optimizing them eliminates or reduces the violation that triggered adaptation. The GP algorithm aims to satisfy the objectives based on their order of priority. For objectives with the same priority, the planning step aims to find a trade-off solution that equally satisfies them. The grammar and the prioritized set of objectives should be provided at the outset and prior to launching GenAdapt.

(4) Executing the adaptation by applying it to the system. This step applies to the running system the adaptation (i.e., the adapted control logic) computed by the planning

step. The desired outcome here is that adaptation should not only address the current observed anomaly but should further make the running system more robust so that the system itself can steer clear of anomalous situations that may arise again.

Our main focus is to improve the planning step of self-adaptation through the application of GP. In designing our GP algorithm for the planning step, we consider two important factors:

First, for GP to assess the adaptation objectives for candidate solutions, the system should be executed for each candidate. Since we cannot execute the system in the adaptation loop to compute objectives for every candidate generated during search, we use the system state collected from the running system at the monitoring step to compute the objectives. The underlying assumption here is that the main parameters of the system remain unchanged for the duration of running the self-adaptation loop.

Second, GenAdapt uses a knowledge base to store the best control logic computed at each round of the planning step. Best solutions generated by previous invocations of GenAdapt are reused in the initial population of GP whenever the planning step is re-invoked. Reusing as bootstrap some of the best solutions from the previous round ensures continuity and incrementality in learning adaptation solutions across multiple rounds of self-adaptation. Reusing the best solutions further aims to ensure that computed adaptations can effectively address uncertainties observed over a long period instead of being focused or over-fitted to the uncertainties observed in a short period.

For example, the formula in Figure 4.1(c) is generated as one of the best solutions at the adaptation round invoked upon the arrival of r_3 and is stored in the knowledge

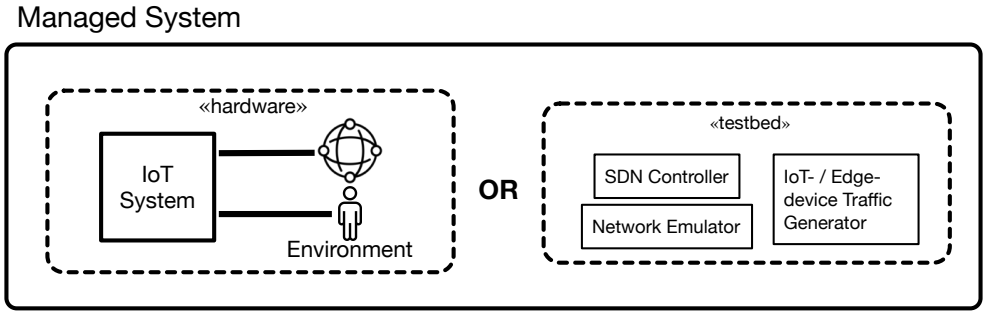

Figure 4.3: The managed system interacting with GenAdapt may be a real or a testbed system.

base. In future rounds of self-adaptation, we will use this formula to bootstrap the initial population of GP. Our conjecture is that this formula will likely solve future occurrences of congestion with minor mutations. We test this conjecture in our empirical evaluation for the SDN domain (see RQ4.1 and RQ4.2 in Section 4.6). We note that certain elements of GenAdapt might have SDN-specific characteristics, and it is important to recognize that our approach might not be applicable across all domains.

4.4 Self-Adaptation for SDN

We apply GenAdapt to make the data-forwarding logic of SDN controllers self-adaptive. For this, we implement GenAdapt as an add-on to the programmable SDN control layer. In an SDN, the control layer has a global view of the entire network and can dynamically modify the network at runtime. The routing logic of an SDN is centralized, decoupled from data and network hardware, and expressed using software code at the control layer. Given the separation of control logic from data planes and forwarding hardware, changing link

weights can be programmed at the control layer in a way that the changes do not affect the existing network flows and instead are used only for routing new flows [17]. Dynamically modifying link weights therefore does not jeopardize network stability.

Figure 4.3 shows a schematic view of an SDN (managed system) interacting with GenAdapt. The managed system can be either a system involving actual hardware, or a realistic testbed. In our work, GenAdapt interacts with a high-fidelity testbed that includes an actual, carrier-grade SDN controller and a simulated network. Using a simulated network rather than an actual network infrastructure is a common approach when designing and evaluating self-adaptation techniques. This enables us to experiment with a multitude of network systems rather than merely one fixed network setup [32, 81, 97, 197].

As shown on the right side of Figure 4.3, our testbed combines three components: An SDN controller capturing the software-defined controller of a network system [27]; a network emulator that simulates the network infrastructure including links, nodes and their properties [115]; and a traffic generator [33] that emulates different types of requests generated by IoT devices and sensors.

To assess the fitness values, i.e., adaptation objectives, we cannot use the actual testbed and its environment for each candidate solution, i.e., each candidate link-weight function, since this system is real-time (wall-clock-time) and requires a few milliseconds to compute the behaviour of a network for each candidate link-weight formula. Our GP algorithm, when invoked, needs to explore a large number of candidate formulas. To do so efficiently, we use surrogate computations that approximate the fitness values [137, 140, 149]. Specifically, as we detail in Section 4.4.2, GenAdapt computes the fitness for each candidate

solution using a snapshot of the system and its environment from the testbed, assuming that the system does not change during a small time period. We set the maximum time budget for adaptation to one second, the smallest monitoring interval permitted by our SDN controller as we discuss in Section 4.5. With regard to the general applicability of this assumption, we note that determining the time budget for self-adaptation requires careful consideration of the particular characteristics of the application domain; one needs to strike a balance where the allocated time is neither too short, impeding genetic improvement, nor too long, resulting in an outdated system state during the monitoring step.

Figure 4.4 shows a domain model consisting of two packages: One package, discussed in Section 4.4.1, captures the main elements of an SDN and its environment, and the other, discussed in Section 4.4.2, specifies the structure of GenAdapt. As indicated in the figure, the data-forwarding logic of SDN controllers is captured using link-weight functions.

4.4.1 Domain Model for Self-Adaptive SDN

The SDN and Environment package captures the static structure of a network as well as its dynamic behaviour over time. The environment of an SDN is comprised of IoT and edge devices which are connected to the network and which generate data-transmission requests (requests, for short) that the network needs to fulfil.

Definition 4.4.1 (Data-transmission Request)**.** A data-transmission request r specifies a data stream sent by a network node s to a network node d. We denote the source node of r by $r.s$ and the destination node of r by $r.d$. Let $[0..T]$ be a time interval. We denote the (data) bandwidth of r at time $t \in [0..T]$ by $r.bd(t)$. The bandwidth is the amount of

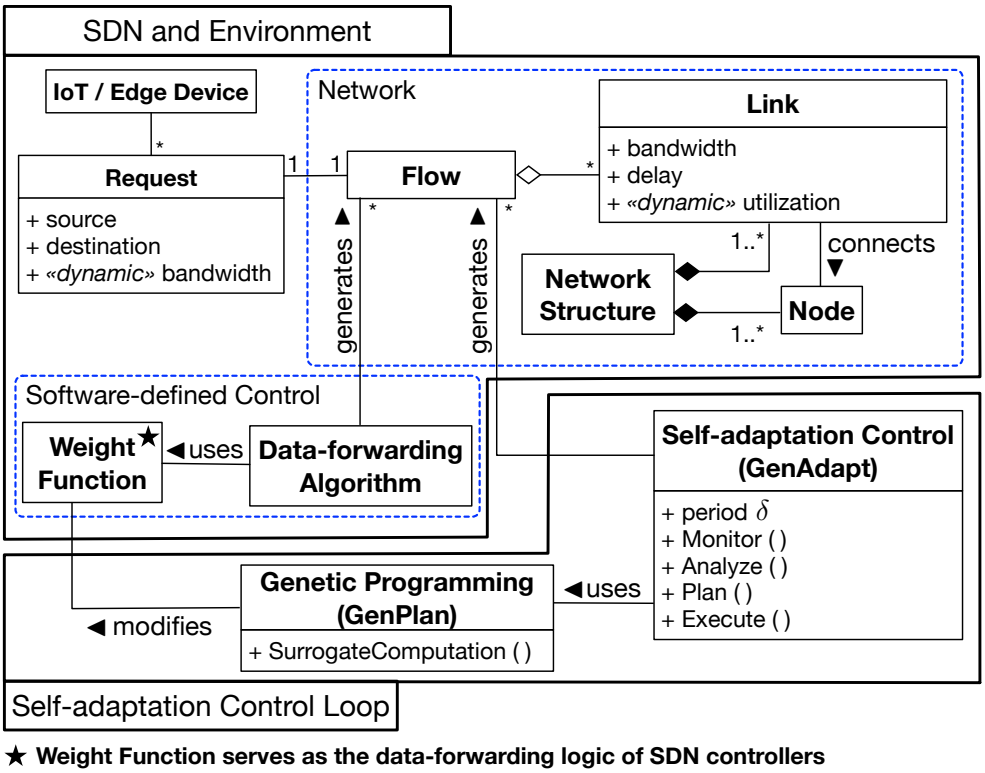

Figure 4.4: Domain model for self-adaptive SDN.

data transmitted from $r.s$ to $r.d$ over time. We measure bandwidth in megabits per second (Mbps). The request bandwidth may vary over time (marked as "dynamic" in Figure 4.4).

In Figure 4.4, the network part of an SDN is delineated with a dashed rectangle labelled Network and includes the Network Structure, Link, Node and Flow entities. Specifically, a network is a tuple $G = (V, E)$, where V is a set of nodes, and $E \subseteq V \times V$ is a set of directed links between nodes. Note that each (undirected) link in Figure 4.1 represents two directed links. Network links have a nominal maximum bandwidth and maximum transmission delay assigned to them based on their physical features and types. The bandwidth of a link is the maximum capacity of the link for transmitting data per second,

and the maximum delay specifies the maximum time it takes for data transmission over a link.

Definition 4.4.2 (Static Properties of a Link)**.** Let $G = (V, E)$ be a network structure. Each link $e \in E$ has a bandwidth $bw(e)$ and a nominal delay $dl(e)$. We denote by $StaticProp(e)$ the tuple $(bw(e), dl(e))$, indicating the static properties of e.

As discussed in Section 4.2, a network handles requests by identifying a directed path (or a flow) in the network to transmit them. Upon the arrival of each request r, a network flow (path) f is established to transmit the data stream of r from the requested source $r.s$ to the requested destination $r.d$. Each flow f is a directed path of links that connects $r.s$ to $r.d$. As shown in Figure 4.4, one flow is created per request. The bandwidth of each flow is equal to that of its corresponding request. Since requests have dynamic bandwidths (Definition 4.4.1), flows have dynamic bandwidths too. We define the throughput of link e at time t, denoted by $throughput(e, t)$, as the total of the bandwidths of the flows going through e at time t.

Definition 4.4.3 (Dynamic Utilization of a Link)**.** Let $G = (V, E)$ be a network, and let $[0..T]$ be a time interval during which the network is being monitored. At each time instance $t \in [0..T]$, each network link $e \in E$ has a utilization $util(e, t)$ computed as follows: $util(e, t) = throughput(e, t)/bw(e)$.

The software-defined control entities in Figure 4.4 include Weight Function and Data-forwarding Algorithm. Data-forwarding algorithms are event-driven and handle requests upon arrival. As discussed in Section 4.2, data forwarding typically generates flows based

on shortest weighted paths between the source and the destination of a request. Thanks to the SDN architecture, link weights are programmable and can be computed by a weight function that accounts for both the dynamic and the static properties of networks. For example, the GP-derived link-weight function in Figure 4.1 (c) uses the dynamic link utilization $util(e, t)$ and the static parameter *threshold* to compute link weights.

4.4.2 Generative Self-Adaptation

In this section, we describe *GenAdapt*, our self-adaptation control loop, and *GenPlan*, the genetic algorithm used in the planning step of GenAdapt. As discussed earlier, self-adaptation control has four main steps; these are specified as methods in the self-adaptation control entity in Figure 4.4. GenAdapt, i.e., our self-adaptation loop, runs in parallel with the data-forwarding algorithm. In contrast to the data-forwarding algorithm which is event-driven, GenAdapt is executed periodically with a period δ, indicated as an attribute of GenAdapt in Figure 4.4. The self-adaptation loop periodically monitors the network for congestion as the environment changes, e.g., due to the arrival of new requests. There is a trade-off between the execution time of GenAdapt and the period δ. In particular, δ should be small enough so that GenAdapt is executed frequently to detect and handle congestion promptly. At the same time, δ should be large enough so that frequent executions of GenAdapt do not interfere with other SDN algorithms and applications [190].

GenAdapt, shown in Algorithm 4.1, executes at every time step $i{\cdot}\delta$ ($i = 0, 1, \ldots$). The monitor step (line 2) fetches the set F_i of flows at time $i \cdot \delta$ and the utilization $util(e, i \cdot \delta)$ of every link e at $i \cdot \delta$. The analyze step (lines 3-4) determines whether the network

Algorithm 4.1 Self-adaptation loop to resolve congestion by learning a new weight function for SDN data-forwarding — GenAdapt.

Input:
G: Network structure
$\cup_{e \in E}$StaticProp(e): Static properties of links
BestSol: Best Solutions from the previous round;
Output: F: Optimized flows
Output: W: Optimized weight function

```
1 for every time step i ∈ {1, ..., n} do
2     F_i, ∪_{e∈E} util(e, i·δ) ← Monitor()        ▷ Dynamic data from SDN Sim
3     maxUtil = Max{util(e, i·δ)}_{e∈E}
4     if maxUtil > threshold then
5         F, W ← GenPlan(G, ∪_{e∈E} StaticProp(e), F_i, BestSol)
6         Apply F and W to the SDN data-forwarding algorithm
```

is congested, i.e., whether adaptation is needed. A network is congested if there is a link $e \in E$ which is utilized above a certain threshold [6, 121]. To detect congestion, the maximum utilization of all the links is compared with *threshold*, which is a fixed parameter of the SDN.

If the network is congested, GenAdapt calls *GenPlan* (line 5). GenPlan, shown in Algorithm 4.2, is our GP algorithm which we discuss momentarily. The output of GenPlan is a new link-weight function as well as a modified set of flows where a minimal number of flows have been re-routed. The new flows and the optimized link-weight function are then applied to the SDN under analysis (line 6). Note that the only change that needs to be done to the network is re-routing a typically small number of flows to eliminate congestion.

GenPlan (Algorithm 4.2) generates link-weight functions that optimize a fitness function characterizing the desired network-flow properties. The link-weight functions are specified in terms of static link properties ($bw(e)$ and $dl(e)$), dynamic link utilization ($util(e,t)$) and utilization threshold. Note that GenPlan always reuses in its initial population half of

the best solutions (candidate weight functions) generated by its previous invocation and stored in a variable named BestSol. Reusing as bootstrap some of the best solutions from the previous round ensures continuity and incrementality in learning the weight functions across multiple rounds of self-adaptation.

Before we describe the GenPlan algorithm in detail, we define three sets, OldFlows, BadFlows, and NewFlows, that are utilized by GenPlan. Specifically, OldFlows is the set of all flows when a congestion is detected; BadFlows is a subset of OldFlows that should be re-routed to resolve network congestion; and, NewFlows is the set of the new flows computed after re-routing those from BadFlows as well as the original flows that did not cause congestion (i.e., those in OldFlows $\setminus$ BadFlows). OldFlows is an input to GenPlan, and BadFlows and NewFlows are computed by GenPlan in order to resolve congestion, as we describe below.

GenPlan starts by selecting a number of flows, BadFlows, from a congested set of flows, OldFlows, using the FindFlowsCausingCongestion algorithm shown in Algorithm 4.3. Following the standard steps of GP, GenPlan creates an initial population P_0 (line 6) containing a set of possible weight functions (individuals). For every individual $\omega \in P_0$, we call ComputeSurrogate, shown in Algorithm 4.4, to re-route the flows in BadFlows when ω is used to compute weights of the network links (line 11 of GenPlan). Specifically, for each ω, ComputeSurrogate generates the set NewFlows which includes (1) the flows corresponding to those in BadFlows, but re-routed based on ω; and (2) the original flows that did not cause congestion (i.e., OldFlows $\setminus$ BadFlows). The set NewFlows is needed to compute the fitness value for ω (line 12). The fitness function aims to determine how close an individual is to resolving congestion. Our fitness function combines three criteria

as we elaborate momentarily. GenPlan evolves the population by breeding and generating a new offspring population (line 9). The breeding and evaluation steps are repeated until a stop condition is satisfied. Then, GenPlan returns the link-weight function with the lowest fitness (BestW) and its corresponding flows (line 17).

The ComputeSurrogate algorithm (Algorithm 4.4), which is called by GenPlan, estimates the flows for each candidate weight function ω, assuming that the flow bandwidths obtained at the monitoring step are constant over the duration of δ. ComputeSurrogate first updates the utilization and weight values for each link, assuming that BadFlows are absent (lines 2-4). The new weights are then used to re-route each flow f in BadFlows by identifying a flow f' as the new shortest path (line 6). After creating each new flow f', the utilization and weight values for each link on f' are updated (lines 7-9). Note that, because of the assumption that flow bandwidths remain fixed during δ, the utilization-per-link $util(e)$ in ComputeSurrogate is not indexed by time.

Following standard practice for expressing meta-heuristic search problems [87], we define the representation, the fitness function, and the genetic operators underlying GenPlan.

Representation of the Individuals. An individual represents some weight function induced by the following grammar rule:

$$\text{exp} ::= \text{exp} + \text{exp} \mid \text{exp} - \text{exp} \mid \text{exp} * \text{exp} \mid \text{exp} \,/\, \text{exp} \mid \text{const} \mid \text{StaticProp} \mid \text{DynamicVar} \mid \text{param}$$

In the above, the symbol | separates alternatives, `const` is an ephemeral random constant generator [205], `StaticProp` are static link properties (Definition 4.4.2), `DynamicVar`

Algorithm 4.2 Generating a new link-weight function and a congestion-free set of flows using Genetic Programming — GenPlan.

Input:
G: Network structure
$\cup_{e \in E}$StaticProp(e): Static properties of links
OldFlows: Current network flows
BestSol: Best Solutions from the previous round
Output: W: New weight function
Output: NewFlows: New network flows

```
1  t=0
2  BadFlows = FindFlowsCausingCongestion(OldFlows, G)
3  Flows = OldFlows \ BadFlows
4  while not(stop_condition) do
5      if t==0 then
6          P_0 = InitialPopOfWeightFormulas() ∪ BestSol
7          OffSprings = P_0
8      else
9          OffSprings = Breed(P_t)
10     for ω ∈ OffSprings do
11         NewFlows = ComputeSurrogate(Flows, BadFlows, G, ω)
12         ω.Fit = Evaluate (G, NewFlows, OldFlows)
13     P_{t+1} = OffSprings
14     t = t + 1
15     BestW = BestSolution(P_0, ..., P_t)
16     bestFlows = ComputeSurrogate(Flows, BadFlows, G, BestW)
17     return BestW, bestFlows
```

is the link utilization (Definition 4.4.3), and `param` is some network parameter. The formula in Figure 4.1(c) can be generated by this grammar rule and is thus an example individual in GenPlan. Note that the SDN data-forwarding algorithm assumes that link values are integers.

Each individual is constructed and manipulated as a parse tree. For example, Figure 4.5 shows the parse tree corresponding to an example link-weight formula, $\frac{th^2}{(th-u)^2}$, where u is a shorthand for $util(e,t)$ and th is a shorthand for threshold. The initial population of GenPlan is generated by randomly building parse trees using the grow method [169]

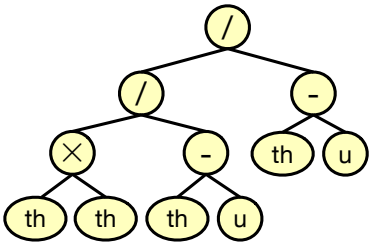

Figure 4.5: A parse tree for an example link-weight formula.

Algorithm 4.3 Selecting a subset of flows whose removal will resolve congestion — FindFlowsCausingCongestion.

Input:
G: Network structure
Flows: Current flows
Output: BadFlows: A subset of Flows causing congestion

```
1 BadFlows = ∅
2 while G is congested do
3     Let e be the most congested link
4     Let f ∈ Flows such that e ∈ f                ▷ selected randomly
5     BadFlows = BadFlows∪{f}
6     Flows = Flows\{f}
7 return BadFlows
```

(i.e., the root and inner nodes are labelled by the mathematical operations, and the leaves are labelled by variables, constants or parameters specified in the grammar). As discussed earlier when GenPlan is called for the first time, the initial population is generated randomly. For subsequent calls to GenPlan, half of the initial population is generated and the other half is reused from the best elements in the last population generated by the previous invocation of GenPlan.

Fitness Function. We propose a fitness function for resolving network congestion in SDNs. Our fitness function is hybrid and combines the following three metrics: (1) Maximum link utilization across all the network links (*Fit1*). For each individual weight function ω, GenPlan computes the set NewFlows of flows based on link weights generated by ω. The

Algorithm 4.4 Re-routing congested flows (BadFlows) by computing shortest paths based on a candidate weight function (ω) — ComputeSurrogate.

Input:
G: Network structure
Flows: Current congestion-free set of flows
BadFlows: Flows that have to be re-routed
ω: Candidate weight function
Output: NewFlows: BadFlows re-routed using ω

```
1  NewFlows = ∅
2  for e ∈ E do
3      util(e) ← sum of the bandwidths of f ∈ Flows s.t. e ∈ f
4      Compute weight for e using ω
5  for f ∈ BadFlows do
6      f′ ← shortest weighted path from source to destination of f
7      for e ∈ f′ do
8          util(e) = util(e) + bandwidth of f′
9          update the weight of e using ω
10     NewFlows = NewFlows∪{f′}
11 return NewFlows ∪ Flows
```

Fit1 metric computes the utilization value of the most utilized network link, considering the flows in NewFlows. If *Fit1* is higher than the threshold, then the network is congested. Hence, we are interested in individuals whose *Fit1* is less than the threshold. (2) The cost of re-routing network flows measured as the number of link updates, i.e., insertions and deletions, required to reconfigure the network flows (*Fit2*). In GenPlan, OldFlows is the current set of congested flows, and as mentioned above, NewFlows is the set of flows computed for each individual ω. We compute *Fit2* as the edit distance between each pair of flows $f \in$ OldFlows and $f' \in$ NewFlows such that f and f' are both related to the same request. Specifically, the distance between two flows f and f' is measured as the longest common subsequence (LCS) distance of two paths [53] by counting the number of link insertions and link deletions required to transform f into f'. We note that this metric has previously been used as a proxy for the reconfiguration cost of network flows [190].

(3) The total data transmission delay generated by the new flows (*Fit3*). The *Fit3* metric is computed as the sum of all the delay values, i.e., $dl(e)$, of the links that are utilized by the new flows (i.e., the NewFlow set). The larger this value, the higher the overall transmission delay induced by NewFlows. The last metric allows us to penalize candidates that generate longer flows compared to those that generate shorter ones.

The *Fit1*, *Fit2* and *Fit3* metrics have different units of measure and ranges. Thus, before combining them, we normalize them using the well-known rational function $\bar{x} = x/(x+1)$ [19]. This function provides good guidance to the search for minimization problems compared to other alternatives.

Among the three metrics, lowering *Fit1* below the congestion threshold takes priority; if a candidate solution is unable to resolve congestion, then we are not interested in the other two metrics. Once *Fit1* is below the threshold, we do not want to lower *Fit1* any further, since we want the network optimally utilized but not congested. Instead, we are interested in lowering the cost and delay metrics. We denote the normalized forms of our three metrics by $\overline{Fit1}$, $\overline{Fit2}$ and $\overline{Fit3}$, respectively, and define the following overall fitness function to combine the three:

$$Fit = \begin{cases} \overline{Fit1} + 2 & (1) \text{ If } Fit1 \geq threshold \\ \overline{Fit2} + \overline{Fit3} & (2) \text{ If } Fit1 < threshold \end{cases}$$

Given a candidate weight function ω, evaluating $Fit(\omega)$ always yields a value in [0..3]: $Fit(\omega) \geq 2$ indicates that ω is not able to resolve congestion; and $Fit(\omega) < 2$ indicates that ω can resolve congestion, and its fitness determines how well ω is doing in reducing cost and delay. Note that in our fitness function defined above, cost and delay are equally

important; we do not prioritize either one. If desired, one can modify the above function by adding coefficients to $\overline{Fit2}$ and $\overline{Fit3}$ to prioritize cost over delay or vice versa.

Genetic Operators. We use one-point crossover [167]. It randomly selects two parent individuals. It then randomly selects one sub-tree in each parent, and swaps the selected sub-trees resulting in two children. For the mutation operator, we use one-point mutation [168] that mutates a child individual by randomly selecting one sub-tree and replacing it with a randomly generated tree, which is generated using the initialization procedure. For the parent selection operator, we use tournament selection [127].

Symbolic Complexity of GenPlan. The most computationally expensive task when computing our fitness function is finding the shortest weighted path from the source switch to the destination switch for each candidate individual. The complexity of finding the shortest weighted path is $O((E + V)\log V)$, where E is the number of links and V is the number of switches in the network [53].

4.5 Tool Support

In this section, we describe the tool that we have developed to build self-adaptivity into SDN data forwarding. Figure 4.6 shows the architecture of our tool support. The tool is composed of four main modules: A traffic generation component (D-ITG) [33], a network emulator (Mininet) [115], an SDN controller (ONOS) [27] and an implementation of our self-adaptation framework (GenAdapt). Below, we discuss each of these modules.

We use Distributed Internet Traffic Generator (D-ITG) [33] — a network traffic gener-

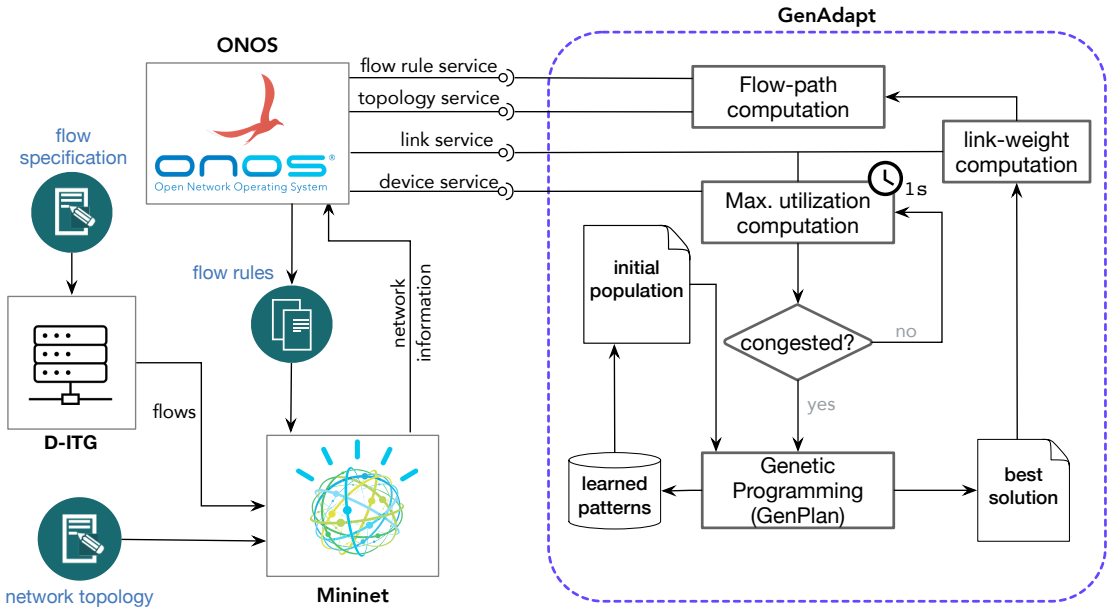

Figure 4.6: Tool architecture.

ator platform — to generate flows based on a specification provided by users (e.g., network administrators). These flows are meant for simulating real-world traffic in real networks. D-ITG can generate various flows with different protocols, bandwidth sizes, and temporal features communicated between the nodes in a network. D-ITG keeps track of various metrics related to the generated data flows. These metrics include, among others, packet loss, delay and jitter.

Mininet [115] is a network emulator which is used for creating a realistic virtual network consisting of virtual hosts, switches and links according to a user-provided network topology. The switch type that we are interested in among the alternative choices available in Mininet is the "Open virtual Switch (vSwitch)". This switch type is Mininet's default and runs in OpenFlow mode. This makes it possible to control switches of this type using

an OpenFlow controller such as ONOS (discussed in the next paragraph). Flow tables are installed on switches for packet forwarding and can be updated based on the instructions sent by the OpenFlow controller which the switches are connected to.

Open Network Operating System (ONOS) [27] is a popular, carrier-grade SDN controllers that can be used with either real or simulated networks. We use ONOS as the OpenFlow controller for the virtual networks created by Mininet. Using the OpenFlow protocol [139], ONOS communicates with the Open vSwitches in a Mininet network. This enables ONOS (i) to obtain a global view of the network by collecting network information and (ii) to manipulate the flow rules as needed in order to improve network performance. ONOS provides a set of services including flow rule service, topology service, link service and device service. All these services can be accessed and programmed via ONOS's API.

Our adaptation framework, GenAdapt, is realized as an add-on for ONOS. As discussed in Section 4.3, GenAdapt consists of four steps: monitor, analyze, plan and execute.

In the first step, our GenAdapt implementation uses ONOS's device service to collect network metrics from the switches. In the second step, which is triggered with a periodicity of one second, GenAdapt uses ONOS's link service alongside the metrics collected in the previous step to compute the utilization of each network link and obtain the maximum link utilization. As discussed in Section 4.4, the network is considered congested if the maximum utilization is greater than a certain threshold. Once a congestion is detected, GenAdapt enters the third step and calls GenPlan (Algorithm 4.2) to generate a new link-weight function (best solution) as well as a set of new link weights calculated by the best solution. In the fourth and final step, using the link weights calculated in the third

step, ONOS's flow rule service computes new shortest weighted paths and updates the flow rules for the switches according to these paths. This step resolves congestion by rerouting some of the flows (specifically, BadFlows in algorithm shown in Algorithm 4.3) based on the new flow rules.

The best solution derived by an invocation of GenPlan (step three) is retained and applied for as long as it has not been replaced by a newer solution (by GenPlan) in response to a future congestion occurrence. The candidate functions created in the last iteration of GP in GenPlan are sorted by their fitness values. The top half of these candidate functions are kept as patterns learned, to be reused by the next invocation of GenPlan. Upon its next invocation, GenPlan creates its initial population as follows: 50% of the initial population are the patterns learned from the previous invocation and the other 50% are randomly generated candidate functions. As we argue in Section 4.3, reusing the patterns learned helps ensure continuity across multiple rounds of self-adaptation.

4.6 Empirical Evaluation

In this section, we investigate the following research questions (RQs) using open-source synthetic and industrial networks. To answer RQ4.1, we use the industrial network and eighteen synthetic networks. To answer RQ4.2, we use six synthetic networks from RQ4.1. To answer RQ4.3, we use eight more synthetic networks alongside two of the synthetic networks from RQ4.1. In total, our evaluation involves one industrial network and 26 synthetic ones.

RQ4.1 (Effectiveness): How effective is GenAdapt in modifying the logic of the SDN data-forwarding algorithm to avoid future congestions? The main novelty of GenAdapt is in attempting to adapt the logic of SDN data forwarding (i.e., the link-weight functions) instead of adapting its output (i.e., the individual flow paths). Through RQ4.1, we compare GenAdapt with two baseline techniques: (i) an approach, named DICES [190], which, similar to GenAdapt, uses MAPE-K self-adaptation, but optimizes individual flow paths; and (ii) OSPF configured using a standard heuristic for setting optimized link weights [49, 51]. By comparing GenAdapt with these baselines, we investigate whether GenAdapt, when called to resolve an existing congestion, is able to reduce the number of occurrences of congestion in the future, without incurring additional overhead.

RQ4.2 (Transferability): Can we improve GenAdapt's performance for a larger network by bootstrapping GenAdapt with best solutions computed on a smaller network that has the same topology as the larger one? The GenAdapt algorithm (Algorithm 4.4) requires computing shortest weighted paths between different nodes in a network. Hence, the algorithm's execution time increases with network size, i.e., the number of links and nodes. Yet, link-weight functions do not contain information that is specific to the underlying network size. We therefore hypothesize that, for networks with similar topology but different sizes, the link-weight functions are likely to be similar, thus making it possible to enhance the performance of GenAdapt by transferring to larger networks the best control logic learned on smaller networks. To validate this hypothesis, we consider a set of networks with the same topology but different sizes. Before applying GenAdapt on the larger networks in this set, we initialize (part of) its population using the best control logic that GenAdapt has learned on the smaller networks in this set. We then compare the results

with those obtained when GenAdapt's first population is initialized randomly.

RQ4.3 (Scalability): Can GenAdapt resolve congestion efficiently as the size of the network and the number of requests increase? To assess scalability, we evaluate the execution time of GenAdapt as the size of the network and the number of requests increase.

Experimental Setup. All experiments were performed on a machine with a 2.5 GHz Intel Core i9 CPU and 64 GB of memory. All our experimental material is available online [1].

4.6.1 RQ4.1 — Effectiveness

Before answering RQ4.1, we present the baselines, the study subjects, the configuration of GenAadapt and the setup of our experiments.

Baselines. Our first baseline, DICES, is from the SEAMS literature. Similar to GenAadapt, DICES is self-adaptive, but unlike GenAadapt, it uses a genetic algorithm to modify the individual flows generated by the SDN data-forwarding algorithm. The second baseline is OSPF [49, 51] configured by setting the link weights to be inversely proportional to the bandwidths of the links as suggested by Cisco [49]. The OSPF heuristic link weights are meant to induce optimal flows that eliminate or minimize the likelihood of congestion. OSPF is widely used in real-world systems [71] and as a baseline in the literature [3, 12, 30, 31, 43, 171, 181].

Study Subjects. For RQ4.1, we use: (1) eighteen synthetic networks, and (2) an industrial SDN-based IoT network published in earlier work [190]. For the synthetic networks, we consider two network topologies: (i) complete graphs, and (ii) multiple non-overlapping

paths between node pairs. The network in Figure 4.1 is an example of the latter topology, where s_1 and s_2 are connected by three non-overlapping paths. Since our approach works by changing flow paths, we naturally focus our evaluation on topologies with multiple paths from a source to a destination, thus excluding topologies where adaptation through re-routing is not possible. The topologies that we experiment with, i.e., (i) and (ii) above, are the two extreme ends of the spectrum in terms of path overlaps between node pairs: In the first case, we have complete graphs, where there are as many overlapping paths as can be between node pairs; and, in the second case, we have no overlapping paths at all. We consider seven complete graphs — referred to as FULL hereafter — with five, six, seven, ten, fifteen, twenty two and thirty two nodes, and consider four multiple-non-overlapping path graphs — referred to as MNP hereafter — with five, eight, twelve and seventeen nodes. More precisely, one MNP graph has five nodes connecting the designated source and destination with three non-overlapping paths (Figure 4.1); others have eight, twelve and seventeen nodes doing the same with four, five, and six (non-overlapping) paths. Following the suggested parameter values in the existing literature for such experiments [190], we set the static properties of the links, namely bandwidth and delay, to 100Mbps and 25ms, respectively.

To instigate changes in the SDN environment, we generate data requests over time and not at once. We space the requests 10s apart to ensure that all the requests are properly generated and that the network has some time to stabilize, i.e., we send requests at 0s, 10s, 20s, 30s, and so on. For each FULL network, we fix a source and a destination node, and generate either three or four requests every 10s. For each MNP network, we generate every 10s two requests between the end-nodes connected by multiple paths. The reason why we

generate fewer requests in the MNP networks is because there are fewer paths compared to the FULL networks. For a given network (FULL or MNP), the generated requests have the same bandwidth. The request bandwidth for each network is selected such that some congestion is created starting from 10s. The request bandwidths are provided online [1]. We denote our synthetic networks by FULL(x, y) and MNP(x, y), where x is the number of nodes and y is the number of requests generated every 10s. The eighteen synthetic networks that we use in RQ4.1 are fourteen FULL networks, once with three requests and once with four requests generated every 10s, and four MNP networks with two requests generated every 10s.

Our industrial subject is an emergency management system (EMS) from the literature [190]. EMS represents a real-world application of SDN in a complex IoT system. This subject contains seven switches and 30 links with different values for the links' static properties. The EMS subject includes a traffic profile characterizing anticipated traffic at the time of a natural disaster (e.g., flood), leading to congestion in the network of the monitoring system. In particular, the network is used for transmitting 28 requests capturing different data stream types such as audio, video and sensor and map data. The static properties of the network links and the data-request sizes are available online [1].[1]

Experiments and metrics. RQ4.1 has two goals: (G1) determine whether, by properly modifying the link-weight function, GenAdapt is able to reduce the number of times congestion happens, and (G2) determine whether GenAdapt can respond quickly enough so that prolonged periods of congestion can be avoided.

[1]Due to platform differences, we could not reproduce the congestion reported in [190]. So, we increased the request bandwidths by 30% to reproduce congestion.

The synthetic subjects are best used for achieving G1, since the requests in these subjects are generated with time gaps as opposed to all at once, which is the case in EMS (industry subject). This characteristic of the synthetic subjects creates the potential for congestion to occur multiple times, in turn allowing us to achieve G1. For G1, we compare GenAdapt with DICES; comparison with the OSPF baseline does not apply, since OSPF's heuristic cannot avoid congestion for our synthetic subjects, and neither can it resolve congestion. To compare GenAdapt with DICES, we keep track of how many times congestion occurs during our simulation, the execution time of each technique to resolve each congestion occurrence, and the total time during which the network is in a congested state. In addition, we measure packet loss, which is a standard metric for detecting congestion in network systems. Packet loss is measured as the number of dropped packets divided by the total number of packets in transit across the entire network during simulation. The simulation time for each synthetic subject was set to end 10s after the generation of the last request. Recall that, in our synthetic subjects, the requests are sent at intervals of 10s and as long as the number of paths in the underlying network is sufficient to fulfill the incoming requests.

To achieve G2, we use EMS in order to compare GenAdapt with DICES and OSPF in terms of handling the congestion caused by requests arriving at once. Since DICES and GenAdapt are self-adaptive, they monitor EMS and attempt to resolve congestion when it occurs. In the case of OSPF, however, the heuristic link weights are meant to reduce the likelihood of congestion and packet loss. For this comparison, we report the total packet loss recorded by each of the three technique over a fixed simulation interval of 5min, and also whether, or not, GenAdapt and DICES were able to resolve the congestion in EMS

Table 4.1: Parameters of GenAdapt.

Mutation rate	0.1	Utilization threshold	80%
Crossover rate	0.7	Minimum of the constant in the GP grammar	0
Maximum depth of GP tree	15	Maximum of the constant in the GP grammar	100
Tournament size	7	Interval between invocations of GenAdapt (δ)	1s
Population size	10		

within the simulation time.

Configuring GenAdapt. Table 4.1 shows the configuration parameters used for GenAdapt. For the mutation and crossover rates, the maximum tree depth and the tournament size, we chose recommendations from either the GP literature [128, 169] or ECJ's documentation [187]. We set δ to 1s since this is the smallest monitoring period permitted by our simulator. We use the utilization threshold given in the literature [6, 121, 190]. Since the range for link-utilization values is [0% .. 100%], we set the minimum and maximum of the constants in GP individuals to 0 and 100, respectively.

To determine the population size and the number of generations for GP, we note that, ideally, GenAdapt should not take longer than δ to execute. The execution time of GenAdapt is likely shorter when it is called in the first rounds of request generation than in the later rounds, since there are fewer flows in the early rounds of our synthetic subjects. Hence, using the fixed time limit of 1s to stop GenAdapt is not optimal. Instead of using a time limit, we performed some preliminary experiments on our synthetic subjects to configure population size and the number of generations for the two topologies of FULL and MNP. For both topologies, we opt to use a small population size (i.e., 10). As for the number of generations, for the FULL networks, we stop GenAdapt when the

fitness function falls below two (i.e., when congestion is resolved but the solution is not necessarily optimized for delay and cost) or when 200 generations is reached. This will ensure that GenAdapt's execution time does not exceed 1s for our FULL networks. The MNP networks are, however, sparser and we are able to increase the number of generations while keeping the execution time below 1s. Specifically, for MNP networks with five nodes, we use 300 generations; and, for the ones with eight, twelve and seventeen nodes we use 500 generations. For EMS, the topology is more similar to a full graph, and as such, we use the same configuration for EMS as that for FULL networks.

Formulas Generated by GenPlan. We characterize the structure and shape of the formulas generated by GenPlan using the following metrics: (1) the number of operators, which represents the number of "+", "−", "∗" and "/" symbols within a formula; (2) the number of variables, which refers to the number of const, StaticProp, DynamicVar, and param elements within a formula; and, (3) the depth of the parse tree corresponding to a formula. As each formula is represented by a parse tree, the number of operators corresponds to the number of non-leaf nodes, and the number of variables corresponds to the number of leaf nodes in the tree. For example, the link-weight formula $\frac{th^2}{(th-u)^2}$, with its corresponding parse tree illustrated in Figure 4.5, contains 5 operators, 6 variables and has a parse-tree depth of 4. To prevent excessively large trees, GenPlan uses a maximum tree depth limit (set 15 in this evaluation), and ECJ's built-in simplification features help reduce redundant or overly complex expressions. We randomly selected 20 best individuals resulting from 20 runs of GenPlan. In these individuals, the number of operators ranges from 0 to 28, with an average of 6.6 and a median of 4. The number of variables, which also represents the number of leaves in the parse tree, ranges from 1 to 29, with an average

of 7.6 and a median of 5. The depth of the parse trees varies from 1 to 15, with an average depth of 4.3 and a median of 4. In GenPlan, individuals that result in a divide by zero are dropped, since they are invalid.

Results. As an example, Figure 4.7 shows the network utilization over time when GenAdapt and DICES are used to resolve congestion for the synthetic network MNP(8, 2). Two simultaneous network requests are sent at 0s, 10s, 20s and 30s, leading to congestion at 10s, 20s, and 30s. The network is congested when the utilization value is above 0.8 (i.e., the utilization threshold). Out of 30 runs for DICES, 18 runs record three congestion occurrences and 12 runs record four. In contrast, out of 30 runs for GenAdapt, six runs record only one congestion occurrence, 16 runs record two, seven runs record three, and only one run records four congestion occurrences. Note that three congestion occurrences at 10s, 20s, and 30s are visible in the figure. However, in the simulated environment, similar to the physical world, there are some small time gaps between the arrivals of the two requests generated each time, and in addition, there are small fluctuations in flow bandwidths over time. Hence, the monitoring step of GenAdapt or DICES may detect congestion twice (i.e., once per request arrival), instead of only once and after the arrival of both requests. In total, for the example in Figure 4.7, the average number of congestion occurrences is 2.1 with GenAdapt and 3.4 with DICES. In addition, with DICES, the average of the total time that the network remains congested is 7.6s, while with GenAdapt, this is reduced to 5.57s.

With the analysis for RQ4.1 intuitively explained over one subject, namely MNP(8, 2), we now present the complete results for this RQ. Table 4.2 compares GenAdapt against DICES for our eighteen synthetic subjects (i.e., seven FULL(n, 3), seven FULL (n, 4) and

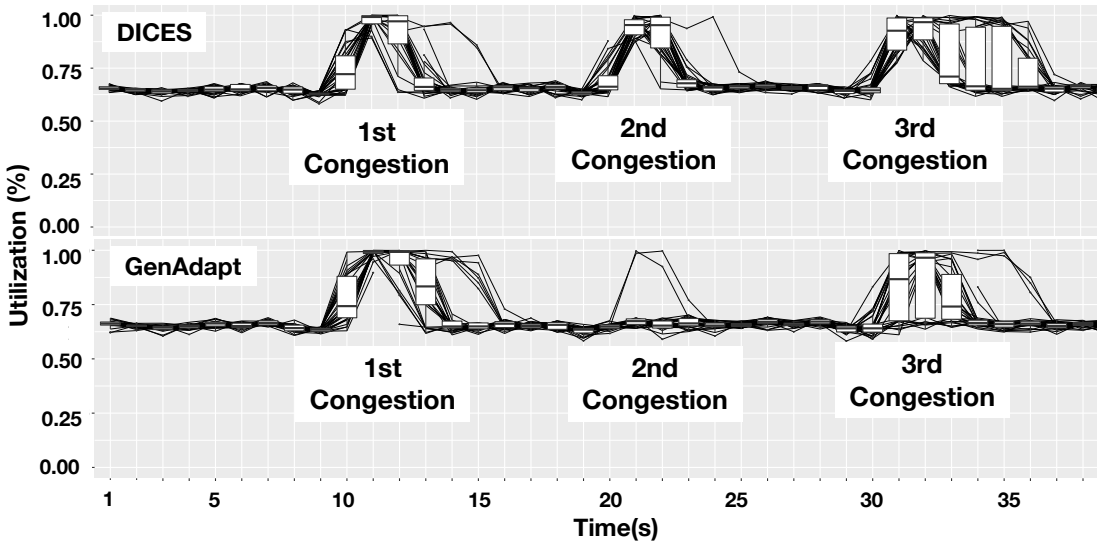

Figure 4.7: Comparing network utilization values over time obtained from 30 runs of DICES and GenAdapt for resolving congestion in an example synthetic subject: MNP(8, 2).

four MNP(n, 2) where n is the number of network nodes) by reporting the average number of congestion occurrences, the average total duration that the network is congested, the average execution time, and the average packet-loss values obtained by 30 runs of DICES and GenAdapt. For four subjects, some runs of DICES failed to resolve the last congestion within the simulation time. Specifically, for FULL(6, 3), 12 runs; for FULL(6, 4), 10 runs; for FULL(7, 3), 20 runs; and for FULL(10, 4), 3 runs of DICES failed to resolve congestion. For these subjects, the reported average number of congestion occurrences and execution times capture only the successful runs of DICES. In contrast, all runs of GenAdapt for all the subjects successfully resolved congestion within the simulation time.

We compare the results of Table 4.2 through statistical testing. We use the non-parametric pairwise Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [202]. We first focus on the following three metrics: number of congestion occurrences,

Table 4.2: Statistical-test results comparing the average number of congestion occurrences, the duration of congestion, the execution-time and the packet-loss values obtained by 30 runs of GenAdapt versus DICES for our eighteen synthetic subjects.

	FULL(5,3)			FULL(5,4)			FULL(6,3) – 40% of DICES runs failed		
	avg(G-D)*	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$
# Congestion	3.1 – 3.2	0.98	0.5(N)	3.8 – 4.4	0.193	0.59(S)	3.5 – 4.5	0.003	0.74(L)
Congestion Duration (s)	6.2 – 7.7	0.0919	0.62(S)	9.53 – 12	0.145	0.61(S)	7.93 – 12	1.69E-06	0.86(L)
Packet Loss (%)	32.1 – 32.33	0.001	0.75(L)	24.86 – 24.99	0.535	0.45(N)	24.94 – 31.73	7.96E-11	0.94(L)
Exec Time (ms)	140.86 – 392.96	<2.2E-16	0.98(L)	271.95 – 440.87	<2.2E-16	0.87(L)	302.21 – 701.06	<2.2E-16	0.9(L)
	FULL(6,4) – 33.3% of DICES runs failed			FULL(7,3) – 66.6% of DICES runs failed			FULL(7,4)		
	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$
# Congestion	3.8 – 5.4	5.39E-06	0.87(L)	4.1 – 6	9.70E-08	1(L)	2.9 – 3.1	0.063	0.59(S)
Congestion Duration (s)	10.03 – 14.47	5.98E-05	0.8(L)	8.03 – 16.63	1.35E-11	1(L)	5 – 5.97	0.0157	0.68(M)
Packet Loss (%)	28.4 – 31.64	1.53E-05	0.83(L)	16.6 – 19.7	0.145	0.61(S)	10.13 – 19.31	8.08E-07	0.87(L)
Exec Time (ms)	956.47 – 802.48	<2.2E-16	0.85(L)	295.94 – 949.2	<2.2E-16	0.97(L)	226.40 – 576.05	<2.2E-16	0.99(L)
	FULL(10,3)			FULL(10,4) – 10% of DICES runs failed			FULL(15,3)		
	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$
# Congestion	3.7 – 4	0.464	0.58(S)	4 – 5.9	9.21E-11	0.96(L)	1.87 – 5	2.757e-09	0.9 (L)
Congestion Duration (s)	7.5 – 8.47	0.029	0.65(S)	8.9 – 16.23	9.53E-11	0.98(L)	2.73 – 5.77	1.017E-06	0.86 (L)
Packet Loss (%)	8.75 – 13.02	0.006	0.71(M)	30.17 – 34.41	1.61E-10	0.98(L)	14.16 – 15.29	3.077E-08	0.92 (L)
Exec Time (ms)	564.88 – 814.8	4.65E-16	0.81(L)	596.31 – 1365.28	<2.2E-16	0.86(L)	732.92 – 1574.83	< 2.2E-16	0.89(L)
	FULL(15,4)			FULL(22,3)			FULL(22,4)		
	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$
# Congestion	2.53 – 3	0.0002	0.73 (M)	2.1 –5	1.681E-09	0.92(L)	2.3 – 3	3.94E-08	0.85(L)
Congestion Duration (s)	3.7 – 3.77	0.2308	0.59 (S)	3.73 – 5.5	0.0005	0.76 (L)	2.9 – 3.43	0.0037	0.71(M)
Packet Loss (%)	0.03 – 25.38	2.065E-11	1 (L)	14.33 – 17.88	6.108E-10	0.97(L)	0.03 – 24.56	1.893e-11	1(L)
Exec Time (ms)	550.8 – 1006.84	< 2.2E-16	0.88 (L)	1237.16 – 2927.74	< 2.2E-16	0.93(L)	588.86 – 2594.24	< 2.2E-16	1(L)
	FULL(32,3)			FULL(32,4)			MNP(5,2)		
	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$
# Congestion	3.7 – 5	0.0065	0.68(M)	3 – 3.03	0.8456	0.51(N)	1.8 – 2.2	0.002	0.68(M)
Congestion Duration (s)	5 – 5.57	0.0647	0.64(S)	3.73 – 3.33	0.1643	0.40(S)	3.87 – 5.23	0.0029	0.71(M)
Packet Loss (%)	14.16 – 20.93	1.382E-06	0.86(L)	20.14 – 24.85	0.0160	0.68(M)	32.39 – 32.65	0.004	0.72(M)
Exec Time (ms)	2654.62 – 3230.97	< 2.2E-16	0.80(L)	2756.13 – 2967.30	8.188E-12	0.79(L)	381.07 – 469.13	0.099	0.59(S)
	MNP(8,2)			MNP(12,2)			MNP(17,2)		
	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$	avg(G-D)	p-value	$\hat{A}_{12}$
# Congestion	2.1 – 3.4	1.55E-08	0.9(L)	2.6 – 4.03	2.985E-09	0.89(L)	2.43 – 5.7	1.479E-11	0.99(L)
Congestion Duration (s)	5.57 – 7.6	0.0002	0.77(L)	3.53–4.33	0.0090	0.68(M)	3.6 – 6.67	5.509E-07	0.87(L)
Packet Loss (%)	32.6 – 33.6	3.43E-06	0.85(L)	18.24 – 20.73	0.0029	0.72(M)	13.98 – 16.52	0.1153	0.62(S)
Exec Time (ms)	787.06 – 392.62	<2.2E-16	0.08(L)	468.97 – 370.20	0.4623	0.53(N)	485.26 – 619.83	1.28E-12	0.79(L)

* The avg(G-D) column shows the average (avg) metrics for GenAdapt (G) versus those for DICES (D).

congestion duration, and packet loss. For nine subjects — MNP(12, 2), MNP(8, 2), MNP(5, 2), FULL(22, 4), FULL(22, 3), FULL(15, 3), FULL(10, 4), FULL(6, 4), and FULL(6, 3) — the p-values for all the comparisons of the above three metrics are lower than 0.05 and the $\hat{A}_{12}$ statistics show large or medium effect sizes, indicating that GenAdapt significantly improves over DICES with respect to the three metrics on the nine subjects. For the seven other subjects — FULL(5, 3), FULL(7, 3), FULL(7, 4), FULL(10, 3), FULL(15, 4), FULL(32, 3) and MNP(17, 2) — GenAdapt significantly improves over DICES with

large or medium effect sizes with respect to at least one of these three metrics. For all the three metrics of the all the subjects except for the congestion duration metric of FULL(32, 4), the averages obtained by GenAdapt are better than those obtained by DICES. For FULL(32, 4), we note that while the average congestion duration for GenAdapt is slightly lower than that for DICES, the statistical test shows no difference between the congestion duration values obtained from DICES and GenAdapt (i.e., the p-value is higher than 0.05). We observe that, for FULL(32, 4), the packet loss and the execution time of GenAdapt are significantly better than those of DICES.

Figure 4.8 shows the packet-loss values obtained by 30 runs of GenAdapt, DICES and OSPF applied to the industry subject (EMS). All 30 runs of GenAdapt and DICES could resolve the congestion in EMS within the simulation time. However, OSPF incurs high packet loss (avg. 36.5%), since its heuristic link weights cannot prevent congestion in EMS. Both GenAdapt and DICES start with high packet loss, but since they can resolve the congestion, packet loss drops quickly, yielding low averages over the duration of simulation. The differences between the packet-loss values of GenAdapt and DICES are neither statistically significant (p-value = 0.44) nor practically significant — a packet-loss difference of 0.4% is negligible in practice. The results of Figure 4.8 signify the need for runtime adaptation in EMS. Further, the results in both Table 4.2 and Figure 4.8 show that while GenAdapt can reduce congestion occurrences over time compared to DICES, doing so does not come at the cost of being less effective in resolving a one-off congestion in EMS caused by several requests arriving almost at once.

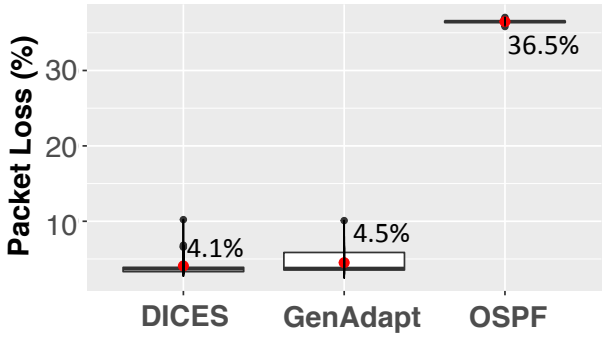

Figure 4.8: Comparing packet-loss values obtained by 30 runs of DICES, GenAdapt and OSPF.

The answer to **RQ4.1** is that GenAdapt successfully resolves congestion in all the subjects. Compared to DICES, GenAdapt reduces the average number of congestion for thirteen subjects with a high statistical significance, while DICES never outperforms GenAdapt in any of the congestion resolution metrics. Further, for our industry subject, GenAdapt significantly outperforms the standard SDN data-forwarding algorithm, OSPF, in reducing packet loss.

4.6.2 RQ4.2 — Transferability

Recall that in its first invocation (i.e., the first time self-adaption is needed), GenAdapt uses a randomly generated initial population, and only in its subsequent invocations, it reuses in its initial population half of the best solutions computed previously. To answer RQ4.2, we modify the GenAdapt configuration used in RQ4.1 as follows: For each of the FULL and MNP graphs, we designate the smallest subject as BaseSubject. Specifically, FULL(5, 3), FULL(5, 4) and MNP(5, 2) are BaseSubjects for the FULL(_, 3), FULL(_, 4)

and MNP(_, 2) categories, respectively. For each category, we use half of the best solutions (ordered based on their fitness values) obtained from the BaseSubject of that category to initialize the population of GenAdapt for the two largest graphs in that category. More precisely, the best solutions obtained from FULL(5, 3) are reused to initialize the populations for FULL(22, 3) and FULL(32, 3), the best solutions obtained from FULL(5, 4) are reused to initialize the populations for FULL(22, 4) and FULL(32, 4), and the best solutions obtained from MNP(5, 2) are reused to initialize the populations for MNP(12, 2) and MNP(17, 2).

Table 4.3 shows the results for the two largest networks in each subject category (i.e., FULL(22, 3), FULL(32, 3), FULL(22, 4), FULL(32, 4), MNP(12, 2) and MNP(17, 2)) obtained from the modified GenAdapt, as described above, against the results obtained from the original GenAdapt configuration described in RQ4.1. We refer to the modified GenAdapt as GenAdapt_Reuse and abbreviate it to GR in Table 4.3; the original GenAdapt algorithm is abbreviated to G in the table.

We report in Table 4.3 the same four metrics as introduced in RQ4.1. Like in RQ4.1, we first consider the following three metrics: number of congestion occurrences, congestion duration, and packet loss. In all the comparisons in Table 4.3, the averages of these three metrics obtained by GenAdapt_Reuse are lower than those obtained by GenAdapt with the exception of the packet loss for FULL(22,4), where the averages for GenAdapt_Reuse and GenAdapt are the same. For five out of six subjects - FULL(22, 3), FULL(32, 3), FULL(32, 4), MNP(12, 2) and MNP(17, 2) - the p-values for at least one of the above three metrics are lower than 0.05 and the $\hat{A}_{12}$ statistics show large or medium effect sizes, indicating that GenAdapt_Reuse significantly improves GenAdapt with respect to at least one of these

three metrics on the five subjects. Notably, for the two MNP cases, GenAdapt_Reuse significantly outperforms GenAdapt for all three metrics with a large or medium effective size.

Table 4.3: Statistical-test results comparing the average number of congestion occurrences, the duration of congestion, the execution-time and the packet-loss values obtained by 30 runs of GenAdapt_Reuse (GR) versus GenAdapt (G).

	FULL(22,3)			**FULL(22,4)**			**FULL(32,3)**		
	avg(GR-G)	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-G)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-G)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
# Congestion	1.43 –2.1	0.0535	0.63(S)	2.16 – 2.3	0.3312	0.55(N)	2.4 – 3.7	0.0126	0.68(M)
Congestion Duration (s)	1.63 – 3.73	0.0001	0.78(L)	2.63 – 2.9	0.2251	0.58(S)	3.2 – 5	0.0480	0.65(S)
Packet Loss (%)	12.09 – 14.33	3.248E-07	0.88(L)	0.03 – 0.03	0.8531	0.51(N)	12.09 – 14.16	0.6467	0.47(N)
Exec Time (ms)	1468.23 – 1237.16	0.0008	0.31(M)	808.92 – 588.86	4.601E-13	0.14(L)	2017.63.04 – 2654.62	0.0454	0.59(S)
	FULL(32,4)			**MNP(12,2)**			**MNP(17,2)**		
	avg(GR-G)	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-G)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-G)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
# Congestion	2.6 – 3	0.223	0.59(S)	1.1 – 2.6	1.819E-11	0.97(L)	1.43 – 2.43	4.941E-06	0.82(L)
Congestion Duration (s)	3.57 – 3.73	0.844	0.52(N)	1.5 – 3.53	7.106E-08	0.89(L)	1.87 – 3.6	3.096E-05	0.81(L)
Packet Loss (%)	3.69 – 20.14	4.271E-06	0.85(L)	16.15 – 18.24	0.0004	0.77(L)	12.56 – 13.98	0.0127	0.69(M)
Exec Time (ms)	1989.06 – 2756.13	0.0097	0.62(S)	229.61 – 468.97	0.0004	0.71(M)	498.05 – 485.26	0.0798	0.60(S)

* The avg(GR-G) column shows the average (avg) metrics for GR versus G.

In relation to execution time, we observe from Table 4.3 that the average execution time of GenAdapt_Reuse is sometimes higher than that of GenAdapt. This increase in execution time is likely attributable to the fact that some of the link-weight functions ranked as best solutions and reused in the initial population may have complex and redundant structures. This causes an overhead in the execution time of GenAdapt_Reuse. We note that the execution time metric refers to the average execution time of a single invocation of a self-adaptation technique. The number of times a self-adaptation technique is required to be called to solve a congestion is not included in this metric, but is represented through the number of congestion occurrences and the total congestion duration metrics. In all the cases where the average of a single execution time of GenAdapt_Reuse takes longer than that of GenAdapt (i.e., FULL(22, 3), FULL(22, 4), MNP(17, 2)), GenAdapt_Reuse still outperforms GenAdapt with respect to the number of congestion occurrences and total

congestion duration. This shows that GenAdapt_Reuse is overall more effective than GenAdapt in resolving congestion occurrences observed over a time period. In cases where the transferred solutions are not well suited to the target topologies, such solutions are gradually corrected or eliminated over successive generations in GP.

In addition, we compare in Table 4.4 the results from GenAdapt_Reuse against DICES and perform statistical tests for our four metrics. For five synthetic subjects - FULL(22, 3), FULL(22, 3), FULL(32, 3), MNP(12, 2) and MNP(17, 2)- GenAdapt_Reuse significantly outperforms DICES with respect to all the four metrics. For FULL(32, 4), GenAdapt_Reuse significantly improves over DICES for packet loss and execution time, and for the two other metrics, i.e., number of congestion occurrences and congestion duration, GenAdapt_Reuse yields higher averages than DICES.

Table 4.4: Statistical-test results comparing the average number of congestion occurrences, the duration of congestion, the execution-time and the packet-loss values obtained by 30 runs of GenAdapt_Reuse (GR) versus Dices (D).

	FULL(22,3)			**FULL(22,4)**			**FULL(32,3)**		
	avg(GR-D)	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-D)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-D)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
# Congestion	1.43 – 5	2.57E-13	1(L)	2.16 – 3	8.986E-11	0.92(L)	2.4 – 5	2.054E-09	0.92(L)
Congestion Duration (s)	1.63 – 5.5	7.248E-11	0.97(L)	2.63 – 3.43	1.546E-05	0.81(L)	3.2 – 5.57	8.265E-06	0.83(L)
Packet Loss (%)	12.09 – 17.88	< 2.2E-16	1(L)	0.03 – 24.56	1.956E-11	1(L)	12.09 – 20.93	3.33E-11	1(L)
Exec Time (ms)	1468.23 – 2927.74	< 2.2E-16	0.95(L)	808.92 – 2594.24	< 2.2E-16	1(L)	2017.63.04 – 3230.97	< 2.2E-16	0.90(L)

	FULL(32,4)			**MNP(12,2)**			**MNP(17,2)**		
	avg(GR-D)	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-D)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg(GR-D)**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
# Congestion	2.6 – 3.03	0.0650	0.61(S)	1.1 – 4.03	9.448E-14	1(L)	1.43 – 5.7	5.247E-12	1(L)
Congestion Duration (s)	3.57 – 3.33	0.2156	0.41(S)	1.5 – 4.33	1.404E-12	1(L)	1.87 – 6.67	1.311E-10	0.98(L)
Packet Loss (%)	3.69 – 24.85	1.362E-08	0.93(L)	16.15 – 20.73	4.19E-10	0.97(L)	12.56 – 16.52	0.0083	0.70(M)
Exec Time (ms)	1989.06 – 2967.30	7.102E-13	0.82(L)	229.61 – 370.20	7.76E-12	0.89(L)	498.05 – 619.83	8.675E-11	0.82(L)

* The avg(GR-D) column shows the average (avg) metrics for GR versus D.

The answer to **RQ4.2** is that bootstrapping GenAdapt with the best solutions it computes on a smaller network improves the algorithm's effectiveness for larger networks with the same topology. In particular, in all our study subjects, bootstrapping reduced either the average number of congestion occurrences or packet loss (or both). For the MNP topology, reductions in the average number of congestion occurrences and packet loss are statically significant.

4.6.3 RQ4.3 — Scalability

To answer RQ4.3, we perform two sets of studies. In the first set, we increase the network size and in the second set — the number of requests. Specifically, first, we create ten synthetic networks with the FULL topology and having 5, 10, ..., 50 nodes; and, for each network, we generate five requests with the same bandwidth at once to reach a total bandwidth of 150Mbps. Second, we create a five-node network with the FULL topology and execute ten different experiments by subsequently generating 5, 10, ..., 50 requests at once such that for each experiment, the requests have the same bandwidth and the sum of the requests' bandwidths is 150Mbps. For both experimental sets, we record the execution time of the configuration of GenAdapt used for the FULL topology as described in Section 4.6.1. Our experimental setup in RQ4.3 follows that used by DICES for scalability analysis. We focus on the FULL topology for scalability analysis because networks with this topology have considerably more links and pose a bigger challenge for self-adaptation planning, as evidenced by the failure of DICES over several networks with the FULL topology (see

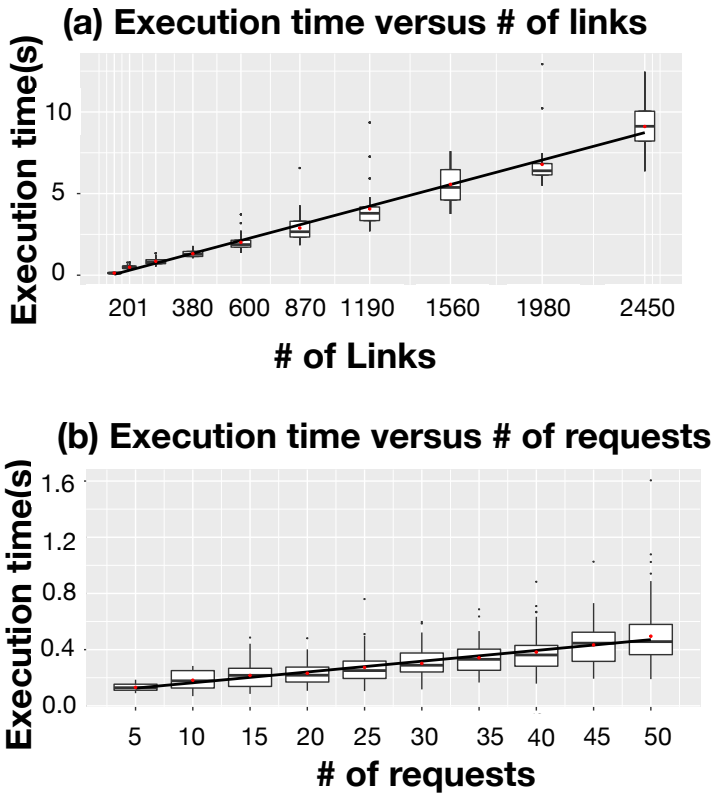

Figure 4.9: Execution times of GenAdapt versus (a) number of links and (b) number of requests.

Table 4.2).

Figure 4.9 shows the execution time of GenAdapt versus the number of network links (Figures 4.9(a)), and versus the number of requests (Figure 4.9(b)). For each network size and for each number of requests, GenAdapt is executed 30 times. For each diagram in Figures 4.9, we have fitted a linear regression line ($time = -0.019 + 0.0036 \times links$ in Figure 4.9(a); and $time = 0.087 + 0.0077 \times requests$ in Figure 4.9(b)). In both cases, we

obtain a p-value of $< 2.2e-16$, indicating that the models fit the data well.

The answer to **RQ4.3** is that the execution time of GenAdapt is linear in the number of requests and in the size of the network.

4.6.4 Threats to Validity

Internal, construct and external validity are the validity aspects most pertinent to our evaluation.

Internal validity. In our approach, we treat the threshold that determines network congestion as a static and fixed parameter. A fixed threshold may not be suitable for all situations as it assumes that network congestion will always occur when a certain threshold is exceeded. This may lead to false alarms or missed opportunities for adaptation as network conditions can vary and different factors may contribute to congestion. A more flexible approach, such as proactively predicting network congestion using machine learning, may be more effective in certain situations. Changing the fixed threshold parameter of an SDN to a dynamic parameter would not affect our approach in the planning step, which is our main contribution. Specifically, the grammar of the link-weight functions would remain unchanged, and the only difference would be the use of a dynamic threshold obtained from the analyzing step instead of a static one.

Construct validity. Our experiments are based on a testbed; the extent to which the measurements obtained through the testbed are reflective of the real world is therefore an important factor to consider. We note that our testbed uses ONOS, an actual carrier-grade SDN controller, which can be used with real networks as well [22, 27, 31, 216]. The network

emulator that our testbed builds on, i.e., Mininet, is widely considered to be a high-fidelity network emulator.

External validity. The centralized architecture adopted in our work may face challenges if deployed in very large networks. We note three ways to mitigate potential scalability issues: (1) Transferring to a larger network the best control logic learned on a smaller network that has the same topology as the larger one. As demonstrated by RQ4.2, GenAdapt's performance improves for a larger network by bootstrapping GenAdapt with best solutions computed on a smaller network with the same topology. (2) Localizing congestion to a subset of the network and applying GenAdapt to that subset. In this case, one would need to account for only the topology and traffic information of the subset, thus improving scalability. (3) Deploying the adaptation component on a cluster, as commonly done in service-oriented architectures [162]. By doing so, GenAdapt can leverage more computational resources, which may be necessary when dealing with very large networks. At the same time, we note that, based on the results of RQ4.3, our approach is scalable to networks with up to 2500 links, which corresponds to a full network with 50 nodes. The number, size and complexity of the networks in our experiments are comparable to those in the literature, e.g., [3, 22, 123, 190, 216].

We scoped our experiments to two network topologies only. As discussed in Section 4.6.1, the two topologies are the two extreme ends of the spectrum in terms of path overlaps between node pairs — the main determinant of complexity for GenAdapt. We expect that, for other topologies lending themselves to congestion resolution via re-routing, our approach will behave within the same range as seen in our experiments. To ensure adequate coverage, our evaluation examined 26 networks based on the two topologies con-

sidered. In addition, our evaluation included a real industrial IoT network with its own topology. The number, size and complexity of the networks in our experiments are comparable to those in the literature, e.g., [3, 22, 123, 190, 216]. To show the effectiveness of our approach (i.e., RQ4.1), we present results from 18 synthetic subjects. In comparison, the paper that introduces DICES [190] uses only one synthetic subject to show effectiveness, and the earlier version of this work [118] uses ten synthetic subjects. As an additional external validity measure for effectiveness, we examine (through RQ4.2) the reuse of adaptation solutions across networks and demonstrate that such reuse improves effectiveness. A different consideration related to external validity is the tuning of our approach. The only parameter of our approach that requires tuning by the user is the number of generations of GenAdapt. As discussed in Section 4.6.1, tuning this parameter is straightforward based on the easily measurable objective of keeping the execution time of GenAdapt less than δ (i.e., the time interval between consecutive invocations of GenAdapt). In practice, engineers can use our simulator to tune this parameter for their specific network topology and traffic profile.

4.7 Summary

In this chapter, we presented GenAdapt — a self-adaptive framework that uses genetic programming for resolving network congestion in SDN-based network layers in cloud-based systems. While existing self-adaptation research focuses on modifying a running system via producing individual and concrete elements, GenAdapt is generative and modifies the logic of the running system so that the system itself can generate the concrete elements

without needing frequent adaptations. We used 18 synthetic and one industrial networks to compare GenAdapt against two baseline techniques: DICES [190] and OSPF [49, 51]. GenAdapt successfully resolved all congestion occurrences in our experimental networks, while DICES failed to do so in four of them. Further, compared to DICES, GenAdapt reduced the number of congestion occurrences and outperformed OSPF in reducing packet loss. Finally, our results show the transferability of the logic learned over networks that share the same topology but have different numbers of nodes. Specifically, we observe that for a given topology, bootstrapping GenAdapt with the best logic learned on a smaller network can significantly improve performance on a larger network in terms of the number of adaptation invocations, the amount of packet loss and the duration the network remains congested.

4.8 Implementation and Availability

We implemented GenAdapt based on the open-source version of DICES [190], a self-adaptive packet forwarding application, which itself builds on ONOS's default reactive forwarding application (called org.onosproject.fwd). For the GenPlan component, we use the genetic programming module provided by ECJ (version 27) [187]. GenAdapt is available online [1].

Chapter 5

A Self-Adaptation Framework to Scale the Cloud Layer

5.1 Introduction

Motivation. In the cloud layer of cloud-based systems, applications are deployed on cloud infrastructure to provide services to users. Instead of relying on monolithic architectures and deploying applications directly onto operating systems, cloud-native architectures adopt principles such as operation through automation platforms, softwarization of infrastructure and networks, and support for migration and interoperability across diverse cloud infrastructures and platforms. These characteristics enable applications to achieve scalability, elasticity and resiliency [112].

As discussed in Chapter 1, cloud-native architectures decompose traditional monolith

applications into multiple small, interdependent services known as microservices. These microservices are deployed on physical or virtual machines in the cloud and collaborate to accomplish specific tasks. Each microservice is encapsulated in lightweight virtualization technologies, such as containers. Upon receiving requests from users, these requests are processed by traversing through several or all of the microservices. The microservices then generate responses which are subsequently sent back to the users through the network.

Given the numerous microservices in cloud-native applications, orchestration is challenging and essential for managing these containerized microservices [5]. Kubernetes (K8S) [113] has emerged as the de-facto standard tool for container orchestration. As shown in Figure 5.1, K8S employs a master-slave architecture. The microservice containers are deployed on a cluster of worker nodes managed by a master node. Each node represents a physical or virtual machine in the cloud. Users send requests to a front-end microservice. These requests are then processed through multiple microservices to generate responses.

By utilizing K8S for orchestration, applications in the cloud layer can efficiently manage containerized microsevices through automated deployment, scaling, and operations. The provided scaling methods focus primarily on CPU and memory consumption [57, 150, 155]. Operators are required to set thresholds for CPU or memory usage, and the number of microservice instances increases or decreases based on the difference between the current usage and the threshold. As a result, optimizing the threshold becomes critical: if the threshold is set too low, the number of microservice instances may be over-provisioned, resulting in unnecessary resource usage; conversely, if the threshold is set too high, the number of microservice instances may be under-provisioned, with too few instances to

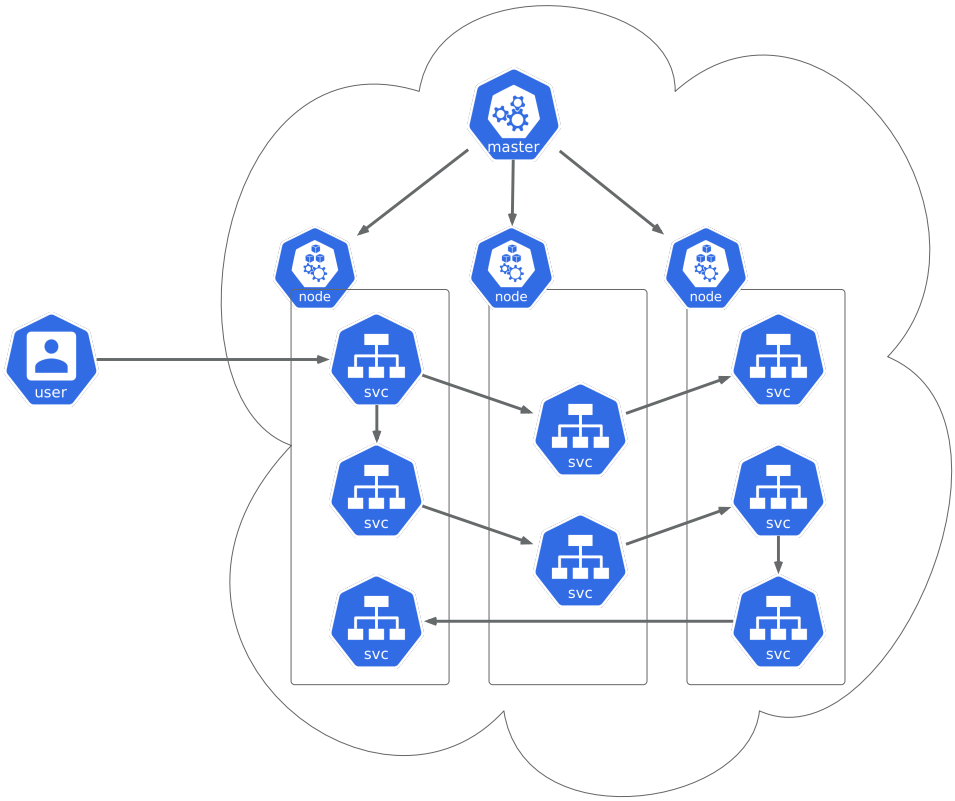

Figure 5.1: A schematic overview of a cloud-native application deployed on K8S.

handle the incoming workload effectively.

However, users of cloud-based applications are typically uninterested in the applications' CPU and memory consumption. Their primary concern lies with the service-level objectives (SLOs) that define the quality of service they experience [59]. Notably, there is no direct link between CPU or memory thresholds and the SLOs. Moreover, different applications may define distinct types of SLOs that measure various metrics. The following are two examples of SLOs used to verify the quality of cloud applications:

(1) 90% of the requests shall be responded to within 500ms.

(2) The success rate of the requests shall be more than 95%.

In the first SLO example, we monitor the actual response times and calculate the 90th percentile (P90), which represents the time within which 90% of the requests are responded to. This P90 value is then compared against the SLO threshold of 500ms to determine if a violation has occurred. In the second SLO example, we track the actual success rate of incoming requests and compare it against the defined SLO threshold of 95%. An SLO violation is detected if the observed success rate falls below this threshold. To satisfy such SLOs, cloud applications must be capable of automatically scaling in response to SLO violations. This requires continuously monitoring SLO-specific indicators, such as the P90 response time and success rate, which directly reflect whether SLOs are being met. Scaling decisions should be made based on these indicators as well as operational metrics, which are quantifiable data that reflect the performance of an application and the application's resource consumption and availability during its operation. These operational metrics include incoming query load (measured in queries per second, or QPS), CPU usage, memory consumption, network traffic, etc.

Problem statement. Automatically scaling a cloud-native application to meet SLOs is a complex challenge. As cloud-native applications usually have tens of different microservices, deciding the appropriate scaling strategy for each microservice under high workload conditions is complex. Some existing approaches [155, 173] focus on metrics for a particular microservice, such as scaling based on the response time of that microservice. However, these approaches cannot ensure that the entire application meets a given SLO and do not account for the interconnections among different microservices. Also, certain approaches [176] focus on specific types of SLOs, such as request latency, or address SLO

violations without considering the detection and elimination of overprovisioning when the SLOs are met. As many service providers (i.e., organizations offering consumer-facing services like online banking, shopping, or entertainment) purchase resources, such as computing power, storage, and specialized hardware like GPUs, from cloud service providers (e.g., Amazon Web Services, Google Cloud, Microsoft Azure, OVHcloud) to deploy their cloud applications, improper management of these resources can lead to significant financial burden.

The main problem that we address in this chapter is as follows: A scaling approach for cloud-native applications should dynamically adapt the number of microservice instances to meet SLOs, while explicitly considering interdependencies among microservices. The approach must handle two key situations: (1) when SLOs are violated due to insufficient resource allocation, and (2) when SLOs are satisfied but with excessive resource usage that could be reduced while still ensuring SLO compliance.

Contributions. We propose RanSLOScale, an auto-scaling framework designed for cloud-native applications orchestrated by K8S. RanSLOScale dynamically scales microservices by increasing or decreasing the number of microservice instances to meet SLOs in real time, while optimizing computational resources. Our RanSLOScale architecture employs random search with the well-known MAPE-K self-adaptation loop [100]. This approach is evaluated on two case studies, a traditional cloud-native online shopping application and a chatbot based on Large Language Models (LLM) [215]. The contributions of this chapter support the thesis contributions as follows:

Con3.1 I propose an auto-scaling framework for cloud-native applications in the cloud

Algorithm 5.1 RanSLOScale.

Input:
operationalMetrics: The quantifiable data that reflect the real-time behavior, resource consumption and availability of a cloud-native application during operation.
SLOThreshold: The specific target defined in a SLO that an application must meet to be considered compliant with its performance or reliability goals.

```
1 DetectingBottlenecks()
2 OfflineTraining()
3 In every period:
4     SLOValue=Monitor()
5     SLOStatus=Compare(SLOValue, SLOThreshold)
6     if SLOStatus==violated or SLOStatus==exceeded then
7         BestList = RanPlan(operationalMetrics)
8         updatePodsNum(BestList)
```

layer of cloud-based systems. It uses random search to dynamically generate scaling decisions based on the operational metrics. This framework ensures compliance with various SLOs, while optimizing computational resources usage by adjusting the number of microservice instances in real time.

Structure. The remainder of this chapter is structured as follows. Section 5.2 introduces our approach. Section 5.3 provides evaluation results. Finally, Section 5.4 summarizes the chapter.

5.2 Approach

RanSLOScale, shown in Algorithm 5.1, leverages the well-known MAPE-K loop to adapt cloud-native application, ensuring that SLOs are met while minimizing computational resource usage. RanSLOScale takes two inputs: `operationalMetrics` and `SLOThreshold`. The `operationalMetrics` input refers to quantifiable data that reflect the application's

real-time behavior, its resource consumption and the availability of resources during operation, such as QPS, CPU usage and memory consumption. `SLOThreshold` specifies the target defined in an SLO that the application must meet to be considered compliant with its performance or reliability goals. For example, in the SLO "90% of the requests shall be responded to within 500ms", `SLOThreshold` is $500ms$. This value is compared with `SLOValue` which represents the actual measured performance of the application relative to the defined SLO. In the same example, if 90% of requests are responded to within $480ms$, then $480ms$ is the `SLOValue`. By comparing `SLOValue` with `SLOThreshold` (line 6 in Algorithm 5.1), RanSLOScale determines the current `SLOStatus`, which categorizes the application's SLO compliance into three states: `met`, `exceeded`, or `violated`. We define these three states as follows:

(1) *SLO is `violated`:* The `violated` state indicates that the SLO is not met and the application requires scaling up. Whether an SLO is violated depends on the nature of the SLO: for latency-based SLOs, a violation occurs when `SLOValue` > `SLOThreshold`, whereas for success-rate-based SLOs, a violation occurs when `SLOValue` < `SLOThreshold`. For example, in the SLO "90% of requests shall be responded to within 500ms", if the response time for 90% of requests is greater than 500ms, then `SLOValue` exceeds `SLOThreshold`, indicating an SLO violation. Similarly, in a success rate SLO of "99.9%", if the actual success rate drops below 99.9%, the SLO is also considered violated. In addition, when out-of-memory (OOM) events are detected in the application, the application is classified in the `violated` state, as theses events signify that the application has exhausted its allocated memory resources, potentially leading to crashes or service disruptions that prevent it from fulfilling its performance or reliability objectives.

(2) *SLO is* `exceeded`*:* This state indicates that the application performs significantly better than the target defined in the SLO, potentially implying inefficient resource usage (i.e., more resources are allocated than necessary). The state is determined by evaluating the exceeding percentage, calculated as either $\frac{\texttt{SLOValue}}{\texttt{SLOThreshold}}$ or $\frac{\texttt{SLOThreshold}}{\texttt{SLOValue}}$, depending on whether a lower or higher value indicates better performance. If this ratio falls below a predefined threshold (e.g., 80%), the SLO is considered `exceeded`. For example, in the SLO "90% of requests shall be responded to within 500ms", if the `SLOValue` is $300ms$, then $\frac{300}{500} = 0.6$, which is below 0.8, indicating the SLO is `exceeded`. We also treat ideal performance as exceeded—for instance, in a success rate SLO of "above 95%", if the actual rate is 100%, then $\frac{95}{100} = 0.95$, which exceeds 0.8, yet we still classify it as `exceeded` because the application is clearly overperforming relative to its target.

(3) *SLO is* `met`*:* This state indicates that the application satisfies the SLO without exceeding it.

Both the `met` and `exceeded` states indicate that the SLO is satisfied. However, the `exceeded` state may signal opportunities for downscaling to reduce potential resource waste.

As shown in Algorithm 5.1, RanSLOScale comprises a bottleneck analysis step (line 1), an offline training step (line 2), followed by a loop containing the four primary steps of the MAPE-K loop visualized in Figure 5.2. We explain each step in the following:

Detecting bottlenecks: In cloud-native applications, certain microservices are more prone to becoming overloaded under high workloads, degrading the overall application performance. These microservices are referred to as bottleneck microservices. Scaling non-bottleneck microservices has limited impact on mitigating SLO violations. Therefore,

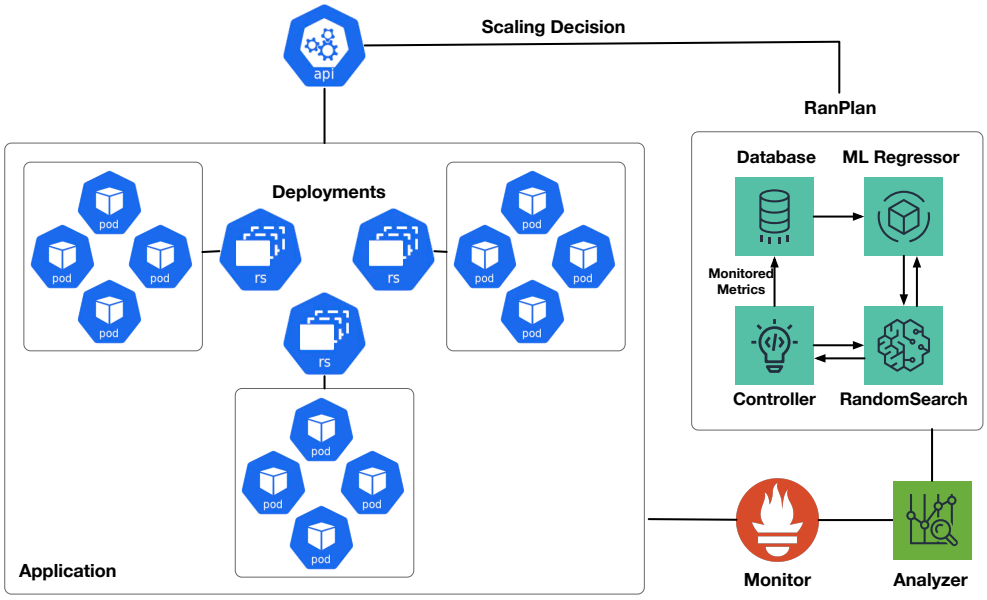

Figure 5.2: Self-adaptation control loop of RanSLOScale.

focusing on autoscaling bottleneck microservices can reduce the search space and accelerate the identification of the optimal scaling strategies [212], particularly in applications with a large number of microservices. To identify bottlenecks, we adopt PBA [212], a bottleneck analysis tool that detects performance degradation and resource wastage under varying workloads. Once the bottleneck microservices are identified, we obtain a candidate list of microservice for applying scaling strategies.

Offline Training: The most accurate way to evaluate a scaling strategy is to apply it to the running application and observe the resulting SLO status. However, this is often impractical due to the associated cost and risk. Instead, in this step, we collect operational metrics, the number of instances for each microservice, and corresponding `SLOValue` to form the training dataset. A machine learning regression model (i.e., ML Regressor) is then trained using this data. The ML Regressor takes as input the `operationalMetrics`

and the current number of instances for each microservice, and predicts the corresponding `SLOValue`, such as response time or success rate. Several ML regressors are trained, and the one with the highest R^2 score and lowest mean absolute error (MAE) is selected for use in the subsequent MAPE-K loop.

Monitoring: In this step (line 4 in Algorithm 5.1), RanSLOScale periodically retrieves `SLOValue` and `operationalMetrics` from the application via Prometheus [21], an open-source monitoring toolkit designed for microservices. It pulls metrics from the application and stores them as time-series data. For metrics not natively collected by Prometheus, such as success rate, response time, and QPS, we employ Istio [20], an open source service mesh that integrates transparently with existing applications. By deploying sidecar proxies alongside application containers, Istio facilitates comprehensive metric collection and enhances observability across the application.

Analyzing: In this step (line 5 in Algorithm 5.1), RanSLOScale evaluates the current `SLOStatus` of the application, by comparing `SLOValue` collected during the most recent monitoring period against the `SLOThreshold`.

To avoid triggering new adaptations before previous ones take effect, RanSLOScale first checks the readiness state of all microservice instances using K8S readiness probes. The analyzing step proceeds only if all microservice instances are in the "Ready" state (i.e., fully initialized and capable of processing requests). If any instance is not ready (e.g., still initializing or recovering), RanSLOScale skips the planning step and continues monitoring.

If the application is in either the `violated` or `exceeded` state (line 6 in Algorithm 5.1), the planning step is triggered. Otherwise, RanSLOScale continues monitoring without

Algorithm 5.2 RanPlan.

Input: SLOStatus.
SLOThreshold
BN: Bottleneck microservices.
podNum_max: Maximum allowable number of instances for a microservice.
podNum_min: Minimum allowable number of instances for a microservice.
podNum_current,d: Current number of instances for microservice d.
operationalMetrics.
Output: BestList

1 $t = 0$
2 **for** not (stop_condition) **do**
3 **if** SLOStatus==exceeded **then**
4 $List_t = [(random(podNum_{min}, podNum_{current,d} - 1)$ for $d \in BN]$
5 **else if** SLOStatus==violated **then**
6 $List_t = [(random(podNum_{current,d} + 1, podNum_{max})$ for $d \in BN]$
7 $SLOValue_{predict} \leftarrow$ MLRegressor(operationalMetrics, $List_t$)
8 $List_t.fitness$=Fitness(SLOThreshold, $SLOValue_{predict}$)
9 t=t+1
10 BestList=BestSolution($List_0, ..., List_t$) ▷ Calculated by comparing the fitness values of the lists

making scaling adjustments.

Planning: In this step (line 7 in Algorithm 5.1), RanSLOScale uses the collected `operationalMetrics` from the monitoring step as input and invokes the `RanPlan` function to generate a scaling decision for the application using random search. The output of this step is `BestList`, which specifies the recommended number of instance adjustments for each bottleneck microservice.

RanPlan, shown in Algorithm 5.2, begins by retrieving the `operationalMetrics` collected during the monitoring step. Following standard random search procedures, RanPlan explores different scaling configurations depending on the current `SLOStatus`: (1) If the SLO is `exceeded`, the objective is to reduce the number of microservice instances for bottleneck microservices to avoid over-provisioning. The candidate instance counts range from $podNum_{min}$ to one less than the current number of instances for each bottleneck service,

where $podNum_{min}$ indicates the minimum allowable number of instances for a microservice. We refer to the number of instances as "podNum" because, in K8S, a `Pod` typically represents a single instance of a microservice. (2) If the SLO is `violated`, the goal is to increase instance counts of bottleneck microservices to improve performance. In this case, candidate values range from one more than the current number to $podNum_{max}$, where $podNum_{max}$ denotes the maximum allowable number of instances for a microservice. For each candidate configuration (i.e., $List_t$), the pre-trained ML Regressor model is used to predict the resulting `SLOValue` (i.e., $SLOValue_{predict}$)(line 7). A fitness function (line 8) is applied to evaluate how well the configuration satisfies the SLO while minimizing number of microservice instances as more instances consume more computational resources (line 8). This fitness function combines two criteria, which we will elaborate momentarily. The search process continues until a stopping condition is met. The stopping condition can be defined either as reaching a maximum number of iterations t, or achieving a fitness score that meets a predefined threshold. Finally, RanPlan returns the optimal configuration (i.e., `BestList`) which specifies the recommended number of instances for each bottleneck service (line 10).

Our fitness function is hybrid and combines two key metrics: (1) the first metric, denoted as $Fit1$, represents the percentage by which the SLOValue violates the SLOThreshold: $Fit1 = \frac{|SLOThreshold - SLOValue|}{SLOThreshold}$. A lower $Fit1$ indicates closer alignment with the target SLO, with zero meaning the predicted value exactly meets the threshold. (2) the second metric, denoted as $Fit2$, quantifies the total number of microservice instances among all bottleneck services, representing the cost of computational resources. It is defined as: $Fit2 = \sum_{n=1}^{N} podNum_n$ where N is the number of bottleneck microservices, as only bottle-

neck microservices are considered for scaling.

Among these two metrics, minimizing $Fit1$ is the top priority. If a candidate solution fails to satisfy the SLO, we focus solely on reducing $Fit1$ without concern for $Fit2$, as satisfying the SLO is the primary objective. Once the SLO is satisfied, further minimizing $Fit1$ becomes unnecessary, and our focus shifts to reducing computational resource usage. In this case, we aim to lower $Fit2$. To reflect this prioritization, we define the overall fitness function as follows:

$$Fitness = \begin{cases} \text{Fit1+1} & \text{(1) If } SLO \text{ is violated} \\ \dfrac{Fit2}{podNum_{max} \times N} & \text{(2) If } SLO \text{ is satisfied} \end{cases}$$

Given a candidate list of instance modifications for bottleneck microservices, denoted as w, the evaluation function $Fit(w)$ always yields a value in the range [0..2]. If $Fit(w) > 1$, it indicates that the candidate configuration fails to meet the SLO. Conversely, if $Fit(w) \leq 1$, the SLO is satisfied, and the fitness value reflects the efficiency of w in reducing the instance numbers, thus minimizing computational resource usage. In scenarios where a pay-as-you-go public cloud service is used, and the SLO is met, the fitness function can be replaced with the actual cost, calculated based on the pricing models of cloud providers such as Amazon EC2 or Google Cloud.

Regardless of whether an SLO violation occurs, the `operationalMetics`, `SLOValue`, and `podNum` values for all microservices collected from the monitoring step are stored in a database, as shown in Figure 5.2. To ensure the ML regressor remains accurate and adapts to potential changes in workload patterns, it is periodically retrained and updated

based on the accumulated data in the database. The list of bottleneck microservices is also periodically updated by reapplying PBA to account for potential shifts in bottlenecks caused by changing workload patterns.

Executing: In this step, RanSLOScale applies the autoscaling strategy (i.e., the list of modifications to the `podNum` for each bottleneck microservices) generated in the planning step to the application via the K8S API. Specifically, the number of instances for each bottleneck microservice is adjusted by modifying the `replica` field in its `Deployment`, which automatically manages the associated `ReplicaSet` to scale the instances (pods) accordingly, as illustrated in Figure 5.2.

Considering the changing environment, the `operationalMetrics` observed during the monitoring phase may differ from those at the time of execution. To ensure that the scaling action does not lead to an SLO violation, RanSLOScale uses the ML Regressor to re-evaluate the proposed scaling strategy based on the most recent metrics. If the model predicts that the strategy will satisfy the SLO under current conditions, the changes are applied to the application. Otherwise, the strategy is discarded and not executed.

5.3 Evaluation

In this section, we evaluate our RanSLOScale through two case studies: (1) a cloud-native boutique shop and (2) an LLM-based chatbot application. Our case study applications are deployed on a K8S cluster consisting of one master node and two server nodes. All three machines run Ubuntu and collectively have a 2x Intel Xeon Gold 6338 CPU, 512GB

of memory and one A40 GPU. Our analysis is guided by the following research questions (RQs).

RQ5.1: (Effectiveness): *How effective is RanSLOScale in scaling cloud applications to meet SLOs while optimizing computational resource usage under dynamic workloads?* To evaluate RQ5.1, we compare RanSLOScale with HPA (Horizontal Pod Autoscaling, the default K8S autoscaling method).

RQ5.2: (Usefulness): *Can RanSLOScale effectively scale different types of cloud applications with varying SLOs?* We access how well RanSLOScale scales the number of microservice instances for a GPU-intensive application with a different SLO under changing workloads.

5.3.1 RQ5.1 - Effectiveness

Before answering RQ5.1, we present the baseline, the case study application, the configuration of RanSLOScale and the experimental setup.

Baseline. Our baseline, HPA, is the default K8S autoscaling method and is widely used in both real-world systems and as a baseline in the literature. It updates the number of microservice instances, using the following formula [150]:

$$desiredReplicas = ceil[currentReplicas * \frac{currentMetricValue}{desiredMetricValue}]$$

where the currentReplicas represents the current number of instances ($podNum_{current}$) for a particular microservice, and the metric value is either CPU usage or memory consumption

per microservice instance (pod). Unlike our approach, which operates at the application level and scales based on SLOs, HPA focuses on scaling individual microservices using these metrics.

Case Study. For RQ5.1, we use Online Boutique, which is an open-source, traditional cloud-native demo application developed by Google [82]. This web-based e-commerce platform consists of eleven microservices that collaborate to enable users to browse items, add them to carts, and complete purchases.

To simulate the communication between users and the application on the cloud, the traffic pattern alternates between high and normal workloads. During normal workload periods, the requests frequency is low enough to satisfy the SLO. During high workload periods, the requests frequency increases to a level where the SLO is violated but remains below the threshold for OOM events or pods crashes, which require extra actions to be taken. The high workload occurs at 10m-20m, 30m-40m and 50-60m, while the remaining time runs at normal workload levels. After 10 minutes of normal workload, the workload increases to a level where the SLO is violated, followed by a 10-minute period before returning to the normal level. We develop our traffic generator based on the open source version of IoTECS [117]. The experiment runs for a total of 70 minutes, resulting in three workload increases and three corresponding decreases.

Experiments and metrics. RQ5.1 evaluates whether RanSLOScale effectively mitigate SLO violations while minimizing computational resource usage. We measure resources consumption by tracking the number of running pods in K8S, as more pods require more computational resources.

For online shopping applications such as online boutique, response time is critical. In our evaluation, we adopt an SLO defined as "90% of requests must have a response time below 500ms" (i.e., SLOValue $\leq$ SLOThreshold, SLOThreshold = 500ms). The SLOValue is measured over a one-minute time window to reflect the response time within which 90% of requests are responded to.

To compare RanSLOScale with HPA, we monitor the SLOValue and number of instances every minute, tracking the frequency of SLO violations and the execution time of RanSLOScale in resolving each violation.

For all microservices, we set the maximum number of instances to 10 (i.e., $podNum_{max} = 10$) and the minimum to 1 (i.e., $podNum_{min} = 1$). Each experiment begins with every microservice running a single instance. We configure the HPA with a CPU usage threshold of 80% and deploy one HPA per microservice, resulting in 11 HPAs for the 11 microservices in online boutique. For RanSLOScale, the candidate pool size is set to 100, which serves as the stopping condition.

In the bottleneck detecting step, we identify bottleneck services in Online Boutique: `frondend` and `productcatalogservice`.

For offline training, we collect data over a 10-hour period by generating random traffic and dynamically adjusting the number of instances for each bottleneck microservice. During this period, the number of instances for each bottleneck service is updated every 5 minutes based on the following rule: if the SLO is met, the system increases the likelihood of reducing the number of instances; if the SLO is violated, it is more likely to increase them. We record the number of instances for bottleneck services, the metrics included in

Table 5.1: Performance comparison of ML Regressor Models trained for Online Boutique shop case study.

Model	R^2	MAE
DecisionTreeRegressor	0.81	134.46
LinearRegression	0.35	571.84
KNeighborsRegressor	0.67	242.03
RandomForestRegressor	0.87	126.94
AdaBoostRegressor	0.63	370.30
GradientBoostingRegressor	0.85	170.57
BaggingRegressor	0.85	140.80
ExtraTreeRegressor	0.80	137.71

`operationalMetrics`, and the `SLOValue` every 5 seconds.

After collecting the training data, we train several machine learning regression models and select the Random Forest Regressor (RFR) as the best-performing model, as it achieves the highest R^2 score and the lowest mean absolute error (MAE), as shown in Table 5.1. To account for the prediction error of RFR, we incorporate the MAE into the predicted SLO value ($SLOValue_{predict}$) when comparing with SLOThreshold or computing fitness scores. Specifically, we use $SLOValue_{predict} + MAE$ or $SLOValue_{predict} - MAE$ depending on the type of SLO. For latency-related SLOs, we use $SLOValue_{predict} + MAE$ to avoid underestimating response time. For success-rate-based SLOs, we use $SLOValue_{predict} - MAE$ to avoid overestimating performance. This adjustment helps prevent misclassification of SLO violations due to prediction errors.

Results: Figures 5.3 and 5.4 illustrate the `SLOValue` (P90 in this case study, representing the time within which 90% of requests are served) and the number of instances monitored under different scaling strategies: no scaling (Do Nothing), HPA, and RanSLOScale. In

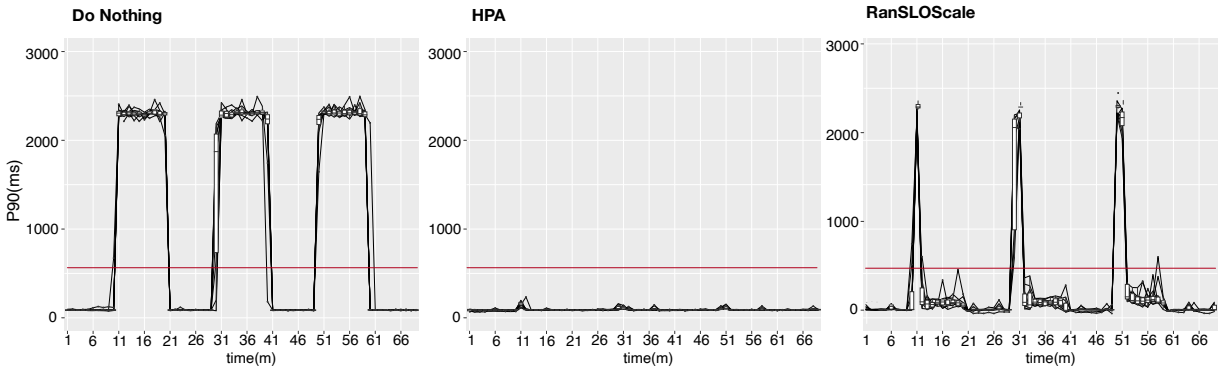

Figure 5.3: SLOValues over time for the Online Boutique shop case study.

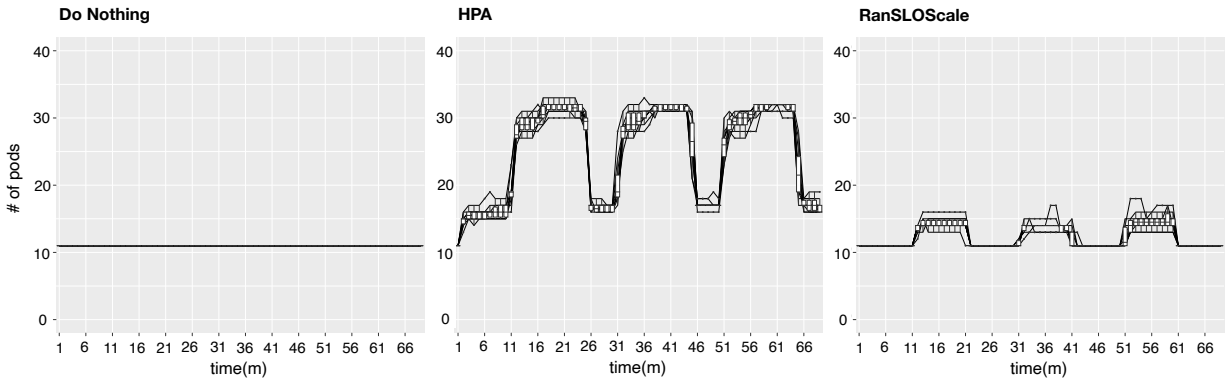

Figure 5.4: # of pods over time for the Online Boutique shop case study.

Figure 5.3, the Do Nothing shows SLO violations occurring at approximately 10m, 30m, and 49m, with exceedances at 20m, 40m, and 60m. When using HPA, no violation is observed throughout the experiment. With RanSLOScale, violations are detected during high workload periods and resolved within 2 minutes each time by increasing the number of instances for bottleneck microservices. When the workload drops from high to low, the number of instances of bottleneck microservices decreases. Figure 5.4 shows that HPA results in a significantly higher number of pods compared to RanSLOScale. This is confirmed by the statistical test results comparing the number of pods in Table 5.2.

Table 5.2: Statistical-test results comparing the SLOValue, the average number of violation occurrences, the duration of congestion and the number of pods obtained by 10 runs of RanSLOScale versus baselines for the Online Boutique case study.

	RanSLOScale vs DoNothing			**RanSLOScale vs HPA**		
	avg	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
SLOValue	366.47 – 1076.46	0.37	0.51(N)	366.47 – 91.62	< 2.2e-16	0.22(L)
# of pods	12.35 – 11	< 2.2e-16	0.29(M)	12.35 – 25.05	< 2.2e-16	0.98(L)
# of Violation	3 – 3	NA	0.5(N)	3 – 0	1.594e-05	0(L)
Violation Duration(m)	1.9 – 10.3	3.25e-12	1(L)	1.9 – 0	3.121e-13	0(L)

Although RanSLOScale results in a higher average SLOValue and number of violations compared to HPA, it conserves computational resources by maintaining fewer pods during low-workload periods while quickly resolving SLO violations once detected. This balance of efficiency and responsiveness is crucial for applications with limited or costly computational resources.

The answer to **RQ5.1** is that RanSLOScale successfully resolves SLO violations and exceedances in a traditional cloud-native application. Compared to HPA, RanSLOScale significantly reduces computational resource usage by requiring fewer pods.

5.3.2 RQ5.2 - Usefulness

In this research question, we evaluate RanSLOScale on a GPU-intensive chatbot application.

Case Study. The application under evaluation is a chatbot system based on large language models (LLMs). This application is GPU-intensive, and GPU resources are both limited and expensive. To successfully adjust the number of chatbot instances, we employ time

slicing on a Nvidia A40 GPU and restrict the maximum GPU memory each pod can request. Based on our preliminary experiments, our cluster can support a maximum of three chatbot instances.

The traffic pattern follows a similar structure to that used in RQ5.1. Since response time is significantly affected by chatbot's answer length, we apply a simple query to the chatbot requiring a short answer each time to eliminate this variable's influence.

Experiments and metrics. RQ5.2 examines whether RanSLOScale can be applied to different types of cloud-native applications with different SLOs.

For chatbot applications, while response time is important, our preliminary experiments indicate that the 90th percentile response time (P90) remains stable under low workload conditions, when the number of instances for each bottleneck microservice is fixed at one. However, under high workload, P90 becomes unstable because the chatbot reaches its service limit, leading to HTTP 500 errors instead of successful responses. This behavior, shown in the second picture of Figure 5.5, suggests that defining SLOs similar to the first SLO example in Section 5.1 is not appropriate for our chatbot application. Instead, it may be more suitable to define alternative SLOs using different metrics or to consider the variability range of P90 as the SLO metric.

To better reflect the health of the chatbot, we adopt success rate as the SLO metric. As shown in the first picture of Figure 5.5, the success rate remains close to one under low workload but drops significantly when the workload increases. For the chatbot application, we define the SLO as "The success rate must be at least 95%." The SLOValue is measured over a one-minute time window.

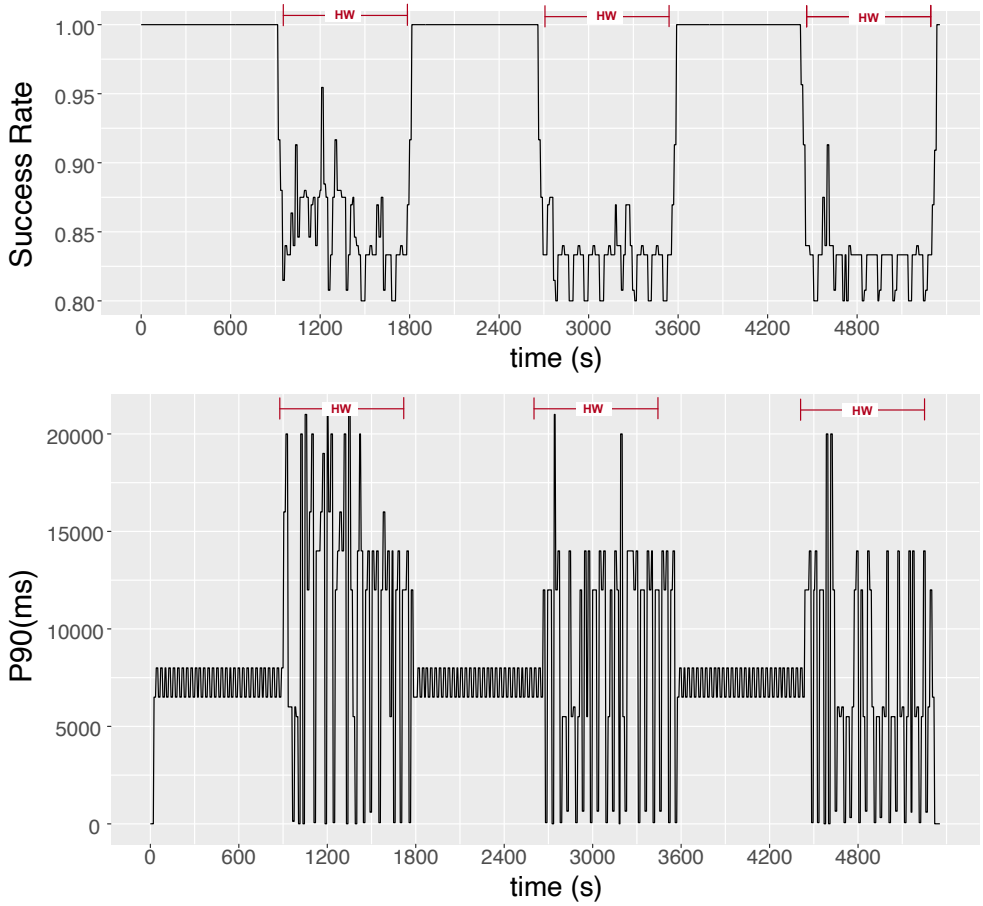

Figure 5.5: P90 and success rate under varying workloads for the Chatbot case study.

We monitor the SLOValue and the number of instances every minute, keeping track of how many times SLO violations occur during operation, as well as the execution time of RanPlan to resolve each SLO violation.

We set the maximum number of instances for microservices to 3 and the minimum number to 1, starting each experiment when all microservices have only one instance. The candidate pool size for RanPlan is also set to 100, which serves as the stopping condition. Since the chatbot has only one primary microservice, it is designated as the bottleneck

Table 5.3: Performance comparison of ML Regressor Models trained for the Chatbot case study.

Model	R^2	MAE
DecisionTreeRegressor	0.89	0.0067
LinearRegression	0.27	0.0369
KNeighborsRegressor	0.84	0.0129
RandomForestRegressor	0.93	0.0084
AdaBoostRegressor	0.61	0.0298
GradientBoostingRegressor	0.78	0.0196
BaggingRegressor	0.90	0.0094
ExtraTreeRegressor	0.86	0.0079

service.

In the offline training step, using only the number of instances and incoming QPS results in poor model performance ($R^2 < 0.72$, $MAE > 0.02$). To improve accuracy, we incorporate the CPU and memory usage per pod as `operationalMetrics`. The updated R^2 scores and MAEs are reported in Table 5.3.

We select the RFR model as the best-performing model based on both R^2 and MAE. Considering the MAE of the RFR, when comparing $SLOValue_{predict}$ with the SLOThreshold to get the predicted SLOStatus or computing the fitness values, we incorporate the MAE by using $SLOValue_{predict}$-MAE instead of $SLOValue_{predict}$ to prevent misclassifications of SLO violations due to prediction errors.

Results. Figures 5.6 presents the performance of the chatbot under three conditions: without any scaling method (Do Nothing), using HPA, and using RanSLOScale. As shown, the chatbot encounters SLO violations during periods of high workload and exceeds the SLO when the workload transitions from high to normal. Combined with the data in

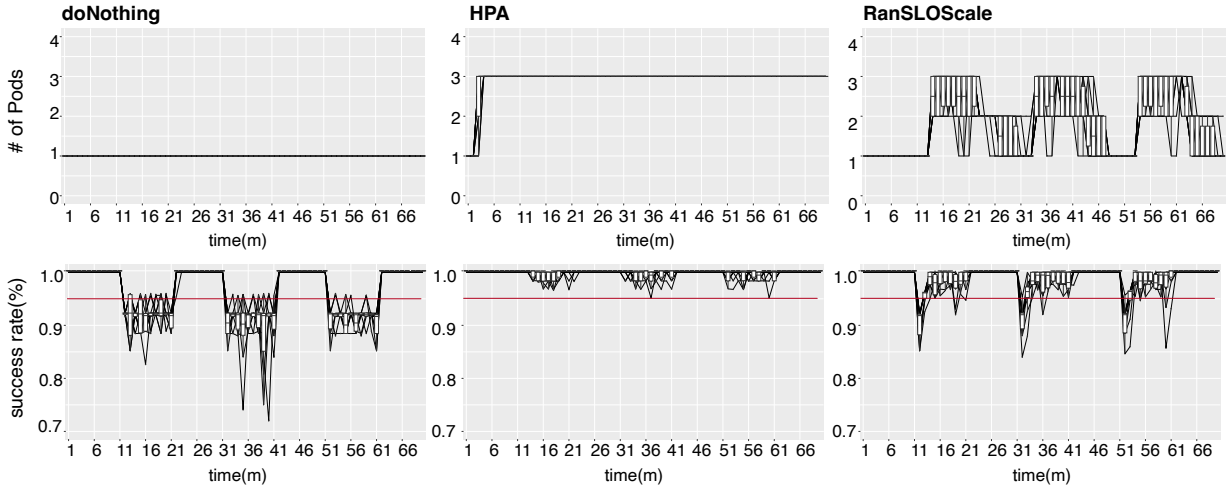

Figure 5.6: # of pods and SLOValues over time for the Chatbot case study.

Table 5.4, we observe that RanSLOScale effectively mitigates SLO violations and reduces the number of instances for bottleneck microservices when the SLO is exceeded. Although RanSLOScale cannot prevent SLO violations, it resolves them within a few minutes and maintains a significantly lower number of instances compared to HPA.

Table 5.4: Statistical-test results comparing the average number of violation occurrences, the duration of congestion, success rate and the number of pods values obtained by 10 runs of RanSLOScale versus other methods for the Chatbot case study.

	RanSLOScale vs DoNothing			**RanSLOScale vs HPA**		
	avg	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**avg**	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
SLOValue	98.86% – 96.26%	< 2.2e-16	0.37(S)	98.86% – 99.76%	1.738e-12	0.57(S)
# of pods	1.79 – 1	< 2.2e-16	0.24(L)	1.79 – 2.93	< 2.2e-16	0.85(L)
# of Violation	3 – 3	NA	0.5(N)	3 – 0	1.594e-05	0(L)
Violation Duration(m)	1.7 – 8.87	3.25e-12	1(L)	1.7 – 0	3.121e-13	0(L)

The answer to **RQ5.2** is that RanSLOScale successfully resolves SLO violations and exceedances in a GPU-intensive LLM-based chatbot application, where the success rate serves as the SLO value.

5.3.3 Threats and Validity

Internal and external validity are the validity aspects most pertinent to our evaluation.

Internal validity: In our approach, we train machine learning models to predict the SLOStatus. If the trained model is inaccurate, such as when the MAE is too large, RanSLOScale may fail to identify a scaling decision that satisfies the SLO. In such cases, even if a suitable scaling decision exists, the predicted SLOStatus may incorrectly indicate a violation, causing potentially valid scaling strategies to be discarded. To mitigate this issue, careful training of the ML models and periodic retraining based on collected data are essential.

Another consideration related to internal validity is the bottleneck changes. RanSLOScale currently does not account for bottleneck changes. If the bottleneck microservices shift, or if the number of bottleneck microservices increases or decreases, the current version of RanSLOScale is unable to adapt the scaling strategy accordingly. Addressing this limitation requires future work. One potential solution is to invoke PBScaler's [212] bottleneck detecting method either when RanSLOScale fails to scale the application or periodically updating bottlenecks.

External validity: We evaluated RanSLOScale using two cloud-native case studies: the Online Boutique (CPU-intensive) and the chatbot (GPU-intensive) in Section 5.3. The results demonstrate that RanSLOScale effectively mitigates SLO violations while optimizing resource utilization. Although these results build confidence in the broader applicability of RanSLOScale, further evaluations with diverse case studies, including applications with different workload characteristics, are necessary to further enhance external validity.

5.4 Summary

In this chapter, we introduced RanSLOScale, an auto-scaling framework designed for cloud-native applications orchestrated by Kubernetes (K8S). RanSLOScale dynamically scales microservices to meet diverse SLOs in real time while optimizing resource utilization and reducing the frequency of SLO violations. Unlike traditional scaling approaches that rely on CPU and memory thresholds, RanSLOScale focuses on application-level SLOs, providing more relevant and effective scaling decisions.

RanSLOScale leverages random search within the well-known MAPE-K self-adaptation loop to generate and evaluate scaling strategies. By incorporating machine learning (ML) models to verify scaling decisions in changing environments, RanSLOScale improves the applications' ability to handle fluctuating workloads and evolving operational conditions.

We evaluated RanSLOScale through two cloud-native case studies: (1) the Online Boutique, a CPU-intensive microservices application, and (2) a chatbot based on Large Language Models (LLM), which is GPU-intensive. The results demonstrate that RanSLOScale effectively mitigates and prevents SLO violations while optimizing computational resources in both scenarios.

5.5 Implementation and Availability

We implemented RanSLOScale based on the open-source version of PBScaler [212], a bottleneck-aware autoscaling framework designed to prevent performance degradation in cloud-native application. RanSLOScale is available online [2].

Chapter 6

Related Work

Cloud-based systems play a critical role in modern computing. Ensuring their reliability requires rigorous testing, while scalability techniques are essential for optimizing performance, allowing applications to scale dynamically in response to changing workloads and environments. Various methods have been proposed to verify system correctness and enhance scalability. In this chapter, we provide a comprehensive review of different methods for verifying and scaling cloud-based systems. Section 6.1 presents an overview of existing simulation and verification techniques, while Section 6.2 discusses approaches for scaling cloud-based systems. We summarize the chapter in Section 6.3.

6.1 Simulation and Verification

We compare our simulation-based stress testing approach proposed in Chapter 3 with the related work in the areas of simulation, testing, and domain-specific languages for cloud-

based systems in the IoT domain.

6.1.1 Simulation and Testing of Cloud-Based IoT systems

To better position our work proposed in Chapter 3 within the literature on IoT simulation and testing, we present in Table 6.1 a structured comparison with the research strands and online tools most closely related to our work. We use the following criteria for the comparison:

(1) Scope refers to the IoT layer(s) that an approach simulates or addresses in terms of the particular testing challenges of the layer(s). Common IoT layers addressed by IoT simulators include the following: device, edge, fog, network, and cloud. Each of these layers possesses specific characteristics and details, and entails different analytical needs. Due to the heterogeneity and intrinsic differences among these layers, IoT simulators and testbeds usually focus on just one or two of these layers. Since fog and edge layers often serve similar functions, they are sometimes used interchangeably in the IoT literature [46]. In our classification of IoT simulators and testbeds, we consider the terms "fog" and "edge" to be synonymous unless a clear distinction between fog and edge is made by the simulator or the testbed. In addition, some methods are focused exclusively on the analysis and synthesis of the data generated and transmitted between different layers, and do not explicitly capture any IoT layer. In Table 6.1, the focus of these methods is denoted as "data".

This classification of IoT layers aligns with the three-layer architecture of cloud-based systems proposed in Chapter 1, which consists of the user, network, and cloud layers. Specifically, the device, edge, and fog layers in IoT systems correspond to the user layer

Table 6.1: Comparison of our approaches proposed in Chapter 3 with relevant work strands in cloud-based IoT systems.

Approach(es)	**Scope**	**Protocol(s)**	**Virtualization**	**Scalability**
MobIoTSim [106] [165], AWS IoT Device Simulator [11]	IoT device	MQTT	×	×
NuvIot [125]	IoT device	REST, MQTT, AMQP, TCP, UDP	×	×
IoTSim-Edge [104]	IoT device, edge device	MQTT, AMQP, XMPP, CoAP	×	×
IoTCloudSamples [201]	IoT device, edge device	MQTT, AMQP, REST, CoAP	✓	×
EdgeCloudSim [196]	IoT device, edge device, network, cloud	Unspecified	✓	×
OMNet++ [156], NS-3 [153], Castalia [34]	network	MQTT, CoAP, TCP, UDP, etc.	×	✓
COOJA [158], Mininet [141], MaxiNet [207]	network	MQTT, CoAP, TCP, UDP, etc.	×	×
PIoT [69]	IoT device, network	Unspecified	×	×
Kaala [56]	IoT device, network, cloud	MQTT, CoAP, TCP, UDP, etc.	✓	×
IOTSim [214], GreenCloud [107], CloudSim [40], NetworkCloudSim [73]	cloud	Unspecified	×	×
Moawad et al. [143]	data	Unspecified	×	×
EMU-IoT [180]	IoT device, edge device, cloud	MQTT, HTTP	✓	×
IOTier [151]	IoT device, edge device, network	4G, Starlink, 802.11n, DSL, etc	✓	×
Fogify [199]	edge device	VXLAN	✓	×*
UiTiOt v3 [129]	IoT device	802.11a/b/g, ZigBee	✓	×*
Sendorek et al. [189]	IoT device, edge device	MQTT, GPIO, I2C, etc	✓	×
iFogSim [84]	IoT device, edge device	Unspecified	✓	×
FogTorch [36]	edge device	Unspecified	×	×
FogNetSim++ [175]	IoT device, edge device, network	TCP, UDP, FTP, HTTP, MQTT, CoAP, AMPQ	×	×
Fogbed [55]	edge device, fog device, network, cloud	Unspecified	✓	×
EmuFog [138]	fog device, network	Unspecified	✓	×
Chapter 3	edge device, cloud	MQTT, TCP, UDP	✓	✓

** These approaches provide partial support for scalability but cannot go beyond limits of physical computational resources.*

in cloud-based systems, as they represent data generation and initial processing before transmission through the network.

(2) Protocol(s) indicate the network protocols that an approach supports. As indicated in Table 6.1, IoT simulators that are focused on the network layer, e.g., [34,52,141,153,156], may support several protocols. IoT simulators and testbeds, which are focused on evaluating edge / fog computing without specifying a network protocol, are denoted "Unspecified" in Table 6.1.

(3) Virtualization indicates whether a simulation or testing approach uses any virtualization techniques, such as virtual machines and containers, to enhance its effectiveness and efficiency.

(4) Scalability denotes whether a simulation or testing approach has mechanisms to expand its operational capacity when exceeding the computational resource limits. In other words, this criterion determines if a simulator or a testbed offers configurable parameters or other methods for scaling up its capacity. Depending on the focus, scaling up the capacity may involve increasing the number of simulated IoT or edge devices, enlarging the simulated network, or raising the number of pod instances in a cloud native system.

Having set the stage with Table 6.1, we now discuss the specific work strands listed in the table. We begin with an overview of IoT simulations, followed by an outline of testbeds and techniques designed for testing IoT systems. We then contrast our work with existing IoT simulation and testing approaches.

IoT Simulation. Numerous simulators exist for different components of IoT systems, e.g., IoT sensors and actuators [11, 56, 106, 125, 165], IoT edge devices [104, 196, 201], IoT

networks [34,141,153,156,158,207], and IoT cloud infrastructures [40,73,107,214]. Among these, the closest ones to our work are IoT edge simulators.

IoTCloudSamples [201] provides a simluation framework aimed at facilitating the development and operation of edge software systems. IoTSim-Edge [104] is an IoT device-to-edge simulator enabling the specification of various IoT-device characteristics, e.g., network connectivity, mobility and energy consumption, and simulating the interactions of IoT devices with edge devices. EdgeCloudSim [196] is an edge computing simulator providing features such as network modelling, edge-server simulation, and packet generation for IoT devices. iFogSim [84] is an IoT device-to-edge simulator focusing on edge computing. It is used to evaluate resource management and application scheduling policies across edge devices. FogTorch [36] is a model designed for determining QoS-aware deployments of IoT applications in fog infrastructures. It accepts an infrastructure and an application as inputs and employs a heuristic search approach to identify suitable deployments on edge devices. FogNetSim++ [175] is a simulator designed for task execution in the fog or edge layer, featuring a centralized broker that facilitates task scheduling and mobility-driven handover for IoT devices. In addition to its system model, FogNetSim++ offers an energy model and a pricing model, enabling users to explore various request scheduling and resource handover algorithms in fog computing environments. Fogbed [55] enhances Mininet [141], a network simulator, by adding support for Docker containers as virtual nodes within its edge and fog layers. Combining Mininet's functions with user-provided Docker images for edge, fog, and cloud, FogBed enables the seamless integration of various computing elements, helping users in designing and testing fog applications. Emufog [138] is an emulation framework based on MaxiNet [207], a network simulator. This framework is designed for creating

network topologies, placing fog nodes within these topologies, and executing applications on the fog nodes. Emufog further provides a latency-based fog node placement policy, facilitating the selection of simulated routers for ideal fog node deployment.

IoT Testing. Testing IoT systems is a challenging task due to numerous factors, e.g., the sheer scale of IoT systems, the heterogeneity of IoT sensors and actuators, the diversity of network protocols and network topologies in IoT systems, and the tight integration of IoT systems with their environment [25].

Motivated by testing IoT systems, Moawad et al. [143] propose a conceptualization of IoT data at run-time. They then use time-series compression and polynomial segmentation algorithms to synthesize realistic sensor data based on their proposed conceptualization.

EMU-IoT [180] is an IoT testbed based on a microservice architecture. This testbed can emulate IoT devices through containers running on virtual machines, create virtualized gateways, and define orchestrators, monitors and load balancers. EMU-IoT has been applied for various testing purposes, e.g., testing the collection of information about resource consumption across an IoT system. IOTier [151] is a testbed that enables using resource-constrained containers as IoT components and grouping these components into a device tier, gateway tier and cloud tier. Using IOTier, one can simulate dynamic changes in the operating conditions of IoT systems and thereby test device capabilities, network performance and load-balancing strategies. Fogify [199] is an edge testbed that focuses on emulating edge topologies and deployment conditions using containerized descriptions. Fogify further provides a mechanism to alter network quality and inject entity and infrastructure downtime at run-time. UiTiOt (version 3) [129] provides a testbed for integrating real IoT

devices with emulated devices that run as containers on virtual machines to support large-scale experimentation. Sendorek et al. [189] propose the concept of a software-defined IoT test environment. Their proposed testbed can define virtual environments, virtual sensors and virtual communication devices as software, thereby allowing the combination of virtual and real equipment for testing IoT applications and communication protocols.

Comparison. Most IoT simulators and testbeds in Table 6.1 are not edge-to-cloud because they either concentrate mainly on the information flow between IoT devices and the edge, or they exclusively target the network or cloud layers. Only two simulators, Fogbed and EdgeCloudSim, along with one testbed, EMU-IoT, are edge-to-cloud. Among these three, EdgeCloudSim [196] focuses on evaluating the performance of different edge computing architectures, rather than assessing the performance of cloud systems as done by our approach. In EdgeCloudSim, data packets are communicated primarily between IoT devices and edge devices, with some packets from IoT devices directed to the cloud when the architecture includes a cloud system. Due to the differences in focus and function between EdgeCloudSim and our approach, we did not select EdgeCloudSim for a detailed design comparison in Chapter 3. On the other hand, EMU-IoT and Fogbed are comparable to our approach in terms of their function and are included in our detailed comparison in Chapter 3.

Depending on specific application scenarios, IoT simulation and testing approaches support different network communication protocols. Nevertheless, MQTT stands out as the most popular choice for IoT systems. Our approach, IoTECS, which focuses on simulating edge devices for stress testing cloud systems, supports MQTT, TCP, and UDP since these protocols are widely used in IoT systems.

Some existing IoT simulators and testbeds (11 out of 29) use virtualization techniques to improve the fidelity of the simulated devices or to efficiently manage simulated entities. However, none of these simulators or testbeds support the objective that motivates our work in Chapter 3, i.e., stress testing of cloud systems via a simulated edge layer. Furthermore, none of the approaches that use virtualization and containerization methods perform systematic experiments that compare containers versus virtual machines and native hosts for simulation. Hence, in addition to being novel in terms of the objective and focus, our evaluation in Chapter 3 is novel in its empirically grounded examination of alternative platforms for edge-device simulation.

The network simulators, OMNet++, NS-3, and Castalia [34, 153, 156], support scalability by providing mechanisms such as parallel and distributed simulation, along with dynamic allocation and deallocation of resources during simulation. These simulators are, however, only focused on the network layer. They neither tackle stress testing for IoT cloud systems nor offer support for capturing the IoT edge and cloud layers as done in our work. Some IoT simulation and testing techniques provide partial support for scalability such as Fogify [199] and UiTiOt v3 [129]. Specifically, Fogify [199] allows for the addition and removal of simulated nodes or entities, while UiTiOt v3 [129] can increase the number of simulated IoT devices using virtual machines. However, such scalability supports are not effective as the number of simulated devices and entities is restricted by the available physical computational resources. In contrast, our approach offers the flexibility to specify offset ranges and speed values. By increasing the offset range or decreasing the speed, users can increase the number of simulated IoT and edge devices, thereby enhancing the simulator's scalability without requiring additional physical resources.

6.1.2 Domain-Specific Languages (DSLs) for IoT

Model-driven engineering (MDE) is a well-established thrust in IoT [143, 147, 193]. DSLs are particularly common as a way to abstract away from the complexity of IoT systems and thereby simplify the construction and analysis of this type of systems.

Sneps-Sneppe and Namiot [192] propose a web-based DSL to simplify data access in IoT web applications. This DSL supports both synchronous and asynchronous data updates as well as communication between processes and sensors. Tichy et al. [200] present a DSL that supports the declarative specification and execution of IoT system components for quality assurance purposes. Gomes et al. [79] propose EL4IoT, a DSL that aims to simplify the development of IoT-device applications by refactoring the network stack. Amrani et al. [14] propose IoTDSL for specifying and assembling usage scenarios of IoT systems in a way that would be understandable for end-users without programming expertise. Åkesson et al. [4] develop a DSL for IoT service composition. This DSL supports live programming, drag-and-drop, and sequential, alternative and parallel events. Finally, Cacciagrano and Culmone [39] propose a DSL, named IRON, for programming smart environments based on Event-Condition-Action (ECA) rules [38]. The end goal here is to provide the ability to intercept specific program anomalies and application-specific safety issues.

These existing DSLs have helped us better shape our approach proposed in Chapter 3; nevertheless, none of these DSLs were designed to tackle our use case of interest, namely, stress testing of cloud-based systems. Specifically, the flexibility we provide for using different simulation platforms without requiring the users to have a deep understanding of virtualization and containerization technologies, and the mechanisms our DSL offers for

scaling up the number of edge devices are, to the best of our knowledge, novel.

6.2 Scaling and Self-Adaptation

We compare our approaches in Chapter 4 and Chapter 5 with related work on self-adaptation frameworks, learning-based self-adaptation, network congestion control and scaling methods for cloud-native applications.

6.2.1 Self-Adaptive Systems

To better position our approaches proposed in Chapter 4 and Chapter 5 within the software engineering literature on self-adaptive systems, we present in Table 6.2 a structured comparison with the research strands most closely related to our work. Our comparison takes the following criteria into consideration:

(1) Adaptation paradigm refers to the method used to engineer self-adaptive systems [208]. In alignment with Weyns' classifications of adaptation paradigms (or waves) [208], we categorize related research as architecture-based, model-based, requirement-based, control-based, or learning-based.

(2) Autonomic computing area is a classification introduced by IBM in the early 2000s [54], inspired by the human body's autonomic system. It refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users. The main autonomic computing areas are: self-configuring, self-healing, self-optimizing and self-protecting. Briefly, self-

Table 6.2: Comparison of our works in Chapter 4 and Chapter 5 with relevant work strands in the literature on self-adaptive systems.

Approach(es)	Adaptation Paradigm	Area of Autonomic Computing	Application Domain	Planning Strategy		
				Online/Offline	Generative	Transferable
Kramer et al. [111]	Architecture-based	General	Generic	Offline	×	×
Rainbow [74] C2 [157]	Architecture-based	General	Generic	Unspecified	×	×
Inverardi et al. [98]	Requirement-based	General	Generic	Unspecified	×	×
Morin et al. [145, 146]	Model-based	General	Generic	Unspecified	×	×
Bhuiyan et al. [29]	Architecture-based	Self-optimizing	Network	Offline	×	×
Stein et al. [198]	Model-based	Self-configuring	Network	Offline	×	×
Anaya et al. [15]	Model-based	Self-configuring	Network	Offline	×	×
DICES [190]	Learning-based	Self-healing	IoT	Online	×	×
FUSION [66], Rodrigues et al. [183]	Learning-based	Self-configuring	Generic	Online	×	×
Camara et al. [41]	Learning-based	Self-configuring	IoT	Online	×	×
Weyns et al. [209], Quin et al. [177]	Learning-based	Self-configuring	IoT	Online	×*	×
Jamshidi et al. [102]	Learning-based	Self-configuring	Robotics	Online	×	×
Chapter 4	Learning-based	Self-healing	IoT	Online	✓	✓
Chapter 5	Learning-based	Self-configuring	Cloud-native	Online	×	×

** These approaches use Deep Learning (DL). DL can be used for generative tasks as well. However, in the existing self-adaptation literature, DL has been used mainly for classification and regression use cases.*

configuring refers to the ability of the system to dynamically adapt to changes in the environment; self-healing describes the capability to discover, diagnose and act to prevent disruptions; self-optimizing entails the ability to optimize the system's resource utilization; and self-protecting involves anticipating, detecting, identifying and protecting against hostile behaviours. While some approaches are specific to a particular area of adaptation, some others propose architectures or models that can benefit several or all areas of autonomic computing. These latter approaches are labeled "General" in Table 6.2.

(3) Application domain refers to the area or field where a given approach is applied. As shown in Table 6.2, some approaches are "Generic" and not tied to any particular domain. The more recent approaches, e.g., [15, 29, 41, 102, 177, 190, 198, 209], are typically targeted at specific domains.

(4) Planning strategy relates to the planning step of the MAPE-K loop (Figure 1.6) and characterizes the strategy used in this step according to three factors: (i) whether the strategy is online or offline, (ii) whether the planning is generative, and (iii) whether the results of planning are transferable. We distinguish between online and offline planning as follows: Online planning strategies continuously learn and adapt while in operation, adjusting to new system behaviour "on the fly", while offline planning strategies lack the ability to dynamically learn and apply at runtime new knowledge gained during system execution. Offline strategies either employ pre-designed static policies or involve a distinct pre-processing training phase that is not updated based on new data from the system in operation. A generative planning approach is designed to update complex structures or formulas used by a system, enabling the system to produce effective planning solutions at runtime and reducing the need for the invocation of the planning step. A transferable

approach generalizes solutions obtained over one system to other systems that share certain common characteristics. Our contribution in Chapter 4 enhances the planning step of self-adaptation by making it generative and transferable.

Having set the stage with Table 6.2, we now discuss the specific work strands listed in the table. We begin with an overview of self-adaptation frameworks, followed by an outline of more recent developments where learning is used for self-adaptation.

Self-adaptation frameworks. Engineering self-adaptive systems, including the principles underlying the construction, maintenance and evolution of such systems, have been studied from different angles and for different domains [15, 29, 45, 50, 75, 198]. Kramer and Magee [111] propose a layered architecture model (component control, change management, and goal management) as an abstraction mechanism for modelling and reasoning about dynamic adaptations. Garlan et al. [74] develop the Rainbow framework — a reusable architecture-based framework for self-adaptive systems. Rainbow uses a rule-based language to implement self-adaptation rules, with the Rainbow architecture model monitoring and detecting the need for adaptation in a managed system.

Inverardi and Mori [98] propose a feature-based self-adaptation framework to tackle the evolution of requirements for high-variability systems. A feature is a dynamic unit representing the smallest part of a service that the user can recognize. The proposed framework handles the consistent evolution of component-based service-oriented systems by adapting their features. Oreizy et al. [157] propose the C2 architectural style to minimize inter-dependencies between components by communicating messages between them through connectors. The proposed architecture is used as a basis for architecture-based

runtime adaptation and evolution. Morin et al. [145, 146] propose a self-adaptation approach based on aspect-oriented modelling. This approach uses aspects as course-grained adaptation units to avoid combinatorial explosion in the adaptation space. The approach employs feature models to capture variability in the system and its context. The combination of aspects and feature models leads to a versatile platform for realizing self-adaption.

Bhuiyan et al. [29] propose an approach for autonomous adaptive sampling for network systems. The adaptive rate sampling is implemented as an embedded algorithm and evaluated using simulators . Stein et al. [198] present a network topology adaptation model and rule language, and demonstrate how the model and the adaptation rule language can be employed in a self-adaptation monitoring loop. In this work, network links are updated or removed in order to adapt to environmental changes. Anaya et al. [15] propose a framework for proactive adaptation, relying on predictive models and historical information about the environment. The predictive models are specifically used for improving the decisions generated by the reasoning engine of the framework.

The above approaches aim to improve the engineering of self-adaptive systems by proposing new architectures or by utilizing advancements in software modelling and requirements engineering. In our works, in order to design a self-adaptive framework, we use a centralized architecture, drawing inspiration from the MAPE-K loop. In that respect, our works have been informed by the above approaches, since they too are largely based on MAPE-K. Movahedi et al. [148] classify the proposed architectures for network autonomy into centralized, distributed and hybrid architectures. While distributed and hybrid architectures may offer better scalability, flexibility and fault tolerance compared to centralized architectures [148], they also introduce additional overhead. In the case of

distributed architectures, this overhead is due to the communication required for achieving consensus among distributed adaptation managers. For hybrid architectures, the overhead arises from the need for optimal allocation of responsibilities and management tasks between the central and distributed entities. Although we adopt a centralized architecture in our works, our contributions are not strictly tied to this decision. Our main contribution is in improving the planning step of self-adaptation, and this falls primarily under the umbrella of dynamic adaptive search-based software engineering [88], which uses a blend of artificial intelligence and optimization for adaptation. In this context, the closest work to ours is DICES [190], which serves as our baseline in Chapter 4 and also employs a centralized architecture. We leave to future work the exploration of deploying our approaches in decentralized and hybrid architectures. We anticipate that such architectures can be supported if a comprehensive view of the targeted system is available.

Learning-based self-adaptation. There is a growing interest in using machine learning techniques in self-adaptive systems [185]. FUSION [66] uses a learning-based approach where, instead of relying on static analytical models, a machine-learning technique, named model trees learning (MTL), is used for configuring the adaptive behaviors of a system. Like our approach, FUSION aims to mitigate environment uncertainty by gradually learning suitable adaptation solutions as the system operates in a new environment. FUSION improves adaptation decisions using analytical models that predict the impact of such decisions on quantifiable features, e.g., system response. Rodrigues et al. [183] use decision trees to predict the time it takes to effect a change in the environment based on the current state of the system, and utilize this information to configure adaptation policies. Camara et al. [41] use reinforcement learning and quantitative verification to reduce the decision

space in a self-adaptation framework for IoT systems. Jamshidi et al. [102] provide an adaptation technique that, in an offline mode, learns a set of Pareto-optimal configurations, and then uses these configurations in adaptation plans at runtime. To reduce the adaptation space during the "Analyze" step of the MAPE-K loop, Quin et al. [177] propose a machine-learning framework based on classification and regression. In a similar manner, Weyns et al. [209] employ deep learning for adaptation-space reduction, supporting three common classes of adaptation goals: threshold, optimization, and set-point goals.

Similar to the above approaches, we adopt a learning-based and online planning strategy for self-adaptation in Chapter 4 and Chapter 5. Some of the above approaches use techniques such as decision trees to predict configuration parameters [66, 183], while others improve planning and decision making by narrowing the adaptation space at runtime using classification / regression [41, 102, 177]. Recently, Weyns et al. [209] have used deep learning (DL) to reduce the adaptation space at runtime. DL can eliminate the necessity for domain engineers and further makes it possible to consider a combination of threshold, minimzation and set-point adaptation goals. To date and in the context of self-adaptation, learning models have been used in a discriminative way, where learned models are applied to candidate adaptation options to identify a subset satisfying the desired adaptation goals. In contrast, in Chapter 4, we adopt a generative approach: rather than using a prediction model to determine which link-weight values satisfy our adaptation goal, we use GP to generate a formula that dynamically produces link-weight values meeting our goal. In addition, we show that our planning strategy in Chapter 4 is transferable. That is, the best formulas learned on smaller networks can be reused for larger networks, when both the smaller and larger networks have the same topology.

We note that DL models can be used in a generative way to produce adaptation solutions, without applying the models to candidate solutions in the adaptation space. For the case studies used by Weyns et al. [209], the size of the adaptation space is a few thousand permutations, and hence, using DL models generatively was likely deemed unnecessary. In our work presented in Chapter 4, however, the search space for link-weight values is much larger. We thus opt to use a generative approach, specifically genetic programming (GP), due to its flexibility in learning formulas that adhere to our specific grammar for link-weight functions to optimize network routing. To our knowledge, we are the first to apply GP for adapting SDNs, and reducing the need for frequent adaptations in this domain. In our work presented in Chapter 5, the adaptation focuses solely on scaling the number of instances of bottleneck microservices, which significantly reduces the search space. As a result, we adopt a random search strategy, supported by machine learning models, to generate the adaptation plan online, keeping the planning step lightweight.

6.2.2 Congestion Control in SDNs

Network congestion resolution has been studied extensively [9, 28, 89, 136]. Some congestion resolution approaches work by adjusting data transmission rates [28]. Several others focus on traffic engineering to provide solutions for better traffic control, better traffic operation, and better traffic management, e.g., by multi-path routing [86], creating a new routing technique [133], and flow-based routing [13]. Chapter 4 is related to the research that utilizes the additional flexibility offered by the programmable control in the SDN architecture [3, 35, 47, 76, 78, 92, 95, 190]. Within this line of research, congestion control

is commonly cast as an optimization problem [47], to be solved using local search [76] or linear programming [3]. Furthermore, there are techniques, e.g., [78], which use pre-defined rules, and techniques, e.g., [217], which use machine learning (particularly deep learning) to dynamically mitigate SDN congestion at runtime.

Building on the OPPBG (OpenFlow-based path-planning with bandwidth guarantee) algorithm, Xiao et al. [211] propose an SDN traffic-routing approach which computes best routing paths for new flows, taking into account both bandwidth requirements and the number of link hops. Zuo et al. [217] use neural networks and build a sequence-to-sequence model to generate ideal routing paths. This approach uses attention mechanisms [213] to ensure reasonable order of the network nodes in a sequence and uses beam search to move out of local optima. Chiang et al. [47] formulate multicast traffic routing in SDN as an optimization problem and propose an algorithm for computing efficient routing paths, considering bandwidth consumption, scalability and rerouting overhead. Gay et al. [76] propose a minimal-time forwarding approach based on local search to lower link loads in SDN and thereby improve network response time to unexpected events such as significant traffic changes and network failures. Ghobadi et al. [78] propose OpenTCP — a rule-based adaptation framework for resolving network congestion in SDN-based data centers. In this approach, when a congestion is detected, the SDN controller generates (through rules) a set of congestion-resolution actions, and distributes these actions to end hosts for execution. Agarwal et al. [3] propose a graph-based algorithm for traffic engineering in cases where there is only a partial deployment of SDN capabilities and where SDNs co-exist with traditional networks. The authors show that network performance can be significantly improved even with a few strategically deployed SDN switches.

In contrast to our approach proposed in Chapter 4, none of the SDN congestion control techniques that we are aware of evolve the routing rules in response to feedback from SDN monitoring. Our approach is generative and its main purpose is, instead of directly predicting suitable solutions for adaptation, to learn a higher-order structure, i.e., a link-weight function, that generates optimal adaptations on demand. The generated link-weight function evolves the flow rules in the flow table by updating the link weights for incoming requests. While the routing algorithm itself is not changed, the flow rules are modified for the incoming requests to avoid future congestion. In this way, our approach automatically learns forwarding rules that reduce the likelihood of future congestion occurrences and iteratively improves these rules when a congestion occurs.

6.2.3 Auto-Scaling in Cloud-Native Applications

As a fundamental capability in cloud-native applications, auto-scaling has been widely studied, leading to various proposed approaches [57, 134, 160, 191]. Here, we review the approaches that close to our work in Chapter 5.

Pozdniakova et al. [172] proposed a high-level architecture for enabling auto-scaling in cloud native applications. Their work focuses on identifying the necessary layers and components for auto-scaling but does not propose a specific method for addressing SLO violations or exceedances. MS-RA [155] combines HPA and VPA, applying a rule-based (threshold-based) scaling based on the current environment, including SLO status, CPU usage and memory consumption. Pramesti et al. [173] proposed a rule-based (threshold-based) horizontal autoscaling method based on respond time as the SLO. Their approach

leverages a Support Vector Regression (SVR) model to predict response times, facilitating autoscaling decisions. However, rather than considering the cloud-native application as a whole and considering the interactions among the microservices, these approaches performs scaling at the level of individual services.

Rather than making adaptations based on SLOs, Liu et al. [122] propose a method that reallocates soft resources (e.g., server threads and database connections) to maximize resource utilization after scaling. While their approach is SLO-aware in the sense that it monitors SLOs, it does not directly use them as decision-making criteria for scaling. Chamulteon [24] is an auto-scaling method that integrates both reactive and proactive decision logic, along with a conflict resolution mechanism. It is rule-based and primarily responds to request arrivals but does not make scaling decisions based on SLOs. PEMA [93] focuses on downscaling cloud-native applications to optimize CPU resource allocation. It initially allocates abundant resources to keep response latency within the SLO, then iteratively reduces resources based on feedback. While it preserves SLO compliance, the method exclusively considers response time as the SLO and optimizes CPU resources through gradual adjustments, requiring multiple iterations to find an optimal solution for a given workload range.

HetSev [142] targets machine-learning inference-serving applications deployed on Kubernetes with GPU support. It introduces an auto-scaling method that minimizes GPU costs while maintaining latency SLOs as constrains. Based on multi-tenancy technology, the approach employs a heuristic algorithm to decide when to scale and selects the cheapest type among heterogeneous GPU instances. However, it lacks fine-grained mechanisms for managing the scalability of microservices. PBScaler [212] is a bottleneck-aware autoscal-

ing method that first detects bottleneck services using a random walk algorithm based on topological potential theory. It then applies genetic algorithms (GA) to scale these bottleneck services. The GA component utilizes an offline SLO violation predictor to estimate whether an SLO violation would occur for a given number of pod instances before applying scaling decisions.

Similar to PBScaler [212], our approach in Chapter 5 adopts a bottleneck-aware strategy for scaling cloud-native applications. In contrast to the methods mentioned above, our approach in Chapter 5 uses a lightweight random search, with the help of machine learning regressors, to select an optimal scaling strategy. This efficiency is enabled by bottleneck detection, as we focus only on scaling bottleneck microservices based on metrics collected from the live environment. The regressor model, initially trained offline, continuously evolves using data collected during application operation, allowing it to adapt to changes in the runtime environment. To the best of our knowledge, our approach is also the only one evaluated on both a traditional cloud-native application and an LLM-based application.

6.3 Summary

In this chapter, we review existing approaches for verifying correctness and optimizing the scalability of cloud-based systems, comparing them with the methods proposed in this thesis. Our verification approach, introduced in Chapter 3, uniquely integrates edge-to-cloud simulation to enable efficient and flexible stress testing of cloud-based IoT applications based on virtualization technology. For performance optimization, Chapters 4 and 5 focus on the SDN and cloud-native domains, respectively, proposing scaling approaches based

on GP and Random Search. These methods dynamically evolve adaptation strategies to ensure that SLOs remain satisfied.

In the next chapter, we conclude this thesis by summarizing its key contributions and discussing future research opportunities.

Chapter 7

Conclusion

In this chapter, I summarize the contributions presented in this thesis and briefly discuss potential directions for future research.

7.1 Thesis Contributions

Cloud-based systems have become an integral part of modern digital services, enabling scalable, flexible, and on-demand access to computing resources across diverse domains such as e-commerce, online gaming, IoT, and social media. As these systems grow in complexity and scale, ensuring scalability across the user, network, and cloud layers becomes a critical challenge. This thesis addresses these challenges by proposing simulation-based stress testing and self-adaptive frameworks that optimize the scalability of cloud-based systems while maintaining Service Level Objectives (SLOs) under fluctuating workloads.

The contributions of this thesis are organized around three high-level research questions that address scalability challenges in cloud-based systems.

First, to tackle the challenge of simulating large numbers of devices in the user layer for stress testing (RQ1), I proposed an efficient model-based simulation framework that employs symbolic representations to reduce resource consumption, as presented in Chapter 3. This approach enables practitioners to conduct large-scale stress tests on cloud-based systems with minimal resource requirements. In addition, I developed a domain-specific language (DSL) to facilitate simulator construction and demonstrated that grouping simulated devices further enhances simulation scalability.

Second, to efficiently address network congestion while preventing future congestion occurrences (RQ2), I introduced a generative self-adaptation framework, GenAdapt, which utilizes genetic programming to dynamically evolve the control logic of running systems, as detailed in Chapter 4. GenAdapt continuously learns and updates system behavior to mitigate SLO violations while reducing the need for frequent adaptation interventions. I applied GenAdapt to Software-Defined Networks (SDNs) in the network layer of cloud-based systems, demonstrating its effectiveness in resolving network congestion and improving overall system performance.

Third, to optimize the scalability of cloud-native applications (RQ3), I developed an auto-scaling framework, RanSLOScale that integrates random search with self-adaptation to dynamically manage the numbers of bottleneck microservice instances in a coordinated manner, as discussed in Chapter 5. RanSLOScale continuously learns from the current environment and generates scaling decisions, increasing the numbers of bottlenecked mi-

croservices instances when SLOs are violated, and releasing resources by scaling down the numbers of bottleneck microservices instances when SLOs are exceeded. This approach goes beyond traditional scaling methods by considering the interdependencies between microservices and addressing resource-intensive applications that are constrained by factors beyond CPU and memory usage. Simultaneously, it resolves various SLO violations, ensuring SLO compliance while minimizing resource waste.

In summary, this thesis presents three approaches aimed at enhancing the scalability of cloud-based systems across different layers. The first approach focuses on verifying scalability in the user layer through efficient stress testing, while the other two address autoscaling challenges in the network and cloud layers by resolving SLO violations, using optimized computational resource.

By advancing the state of the art in stress testing, self-adaptive network management, and auto-scaling for cloud-native applications, this research provides practical methodologies and tools that enable service providers to improve system scalability. In practice, symbolic simulation and device grouping offer an efficient approach for large-scale stress testing, while self-adaptive network control mechanisms like GenAdapt help resolve and prevent network congestion. At the cloud layer, SLO-driven autoscaling enables more effective resource allocation and ensures responsive behaviour to workload fluctuations. These approaches empower cloud-based systems to autonomously adapt to dynamic conditions while maintaining service quality. By adopting these techniques and leveraging the publicly available tools developed in this research, practitioners can build more efficient and scalable cloud-based architectures.

In addition to addressing various research questions related to scalability, this thesis opens the door to a wide range of future research opportunities, which are outlined in the following section.

7.2 Opportunities for Future Research

This section explores various technical and conceptual opportunities for future research related to the contributions of this thesis.

Enhancing Stress Testing DSL for Cloud-Based Systems. While our current verification approach (IoTECS) has been successfully adopted by our industry partner, addressing existing limitations can further improve its usefulness and applicability. Key advancements include introducing dynamic adjustments through high-level constructs for more flexible configurations, providing real-time access to system metrics during stress testing, and integration of SLOs to directly access stress impacts on service quality. Another promising direction is to make the DSL cost-aware. IoT solution providers often rely on externally hosted resources for stress testing, incurring costs based on CPU and memory usage for containers and virtual machines. Incorporating cost-awareness into the DSL will enable optimization of simulation parameters to reduce expenses. Optimization techniques, such as search algorithms, could be explored to strike an optimal balance between the number of simulated IoT devices and the associated simulation costs. In addition, IoTECS could be expanded to support stress testing in other domains of cloud-based systems, such as cloud-native applications, as demonstrated in Chapter 5. Although existing stress testing tools (e.g., JMeter) are available, our approach is distinguished by its

user-friendly design, symbolic simulation of users, and ability to leverage virtualization. Extending these capabilities to other domains will further expand IoTECS's applicability and impact.

Distributed Network Scaling. Another area for future work is transitioning GenAdapt's design from a centralized architecture to a distributed or hybrid one to enable scalability in larger networks. Scaling GenAdapt to more complex and larger-scale environments requires designing distributed control mechanisms that can coordinate adaptation decisions across multiple nodes while minimizing latency and communication overhead. A distributed approach can also enhance fault tolerance by preventing a single point of failure, ensuring that adaptation processes continue smoothly even if some nodes become unresponsive. In addition, extending self-adaptive logic to the user layer of the cloud-based systems presents an opportunity to improve traffic control. This approach allows cloud-based systems to achieve faster response times, reduce congestion in high-volume environments, and improve overall system efficiency by enabling localized decision-making and adaptation closer to the data sources.

Combining Horizontal Scaling and Vertical Scaling. Our proposed cloud-native autoscaling approach, RanSLOScale, has primarily focused on Horizontal Pod Autocaling (HPA) due to its suitability for responding to sudden workload changes. Vertical Pod Autoscaling (VPA) offers stable resource recommendations over longer periods, making it complementary to HPA [173]. While VPA is not suitable for the evaluations conducted in this thesis, future work aims to explore a hybrid scaling approach that combines HPA and VPA. By integrating HPA and VPA, cloud-based systems can dynamically respond to both short-term workload surges and long-term resource demands. This hybrid approach has

the potential to optimize resource allocation by rapidly scaling out during sudden workload changes while gradually refining resource configurations to maintain stability and reduce overhead. Investigating adaptive policies to switch between HPA and VPA, depending on workload characteristics, presents a promising avenue for enhancing performance and efficiency of cloud-based systems.

References

[1] Genadapt. https://figshare.com/s/de6eb6e61816401b5c9e, 2021.

[2] Ransloscale. https://github.com/JiaLi123456/RanSLOScale, 2025.

[3] Sugam Agarwal, Murali S. Kodialam, and T. V. Lakshman. Traffic engineering in software defined networks. In Proceedings of the 2013 Annual IEEE International Conference on Computer Communications (INFOCOM'13), pages 2211–2219, 2013.

[4] Alfred Åkesson, Görel Hedin, Mattias Nordahl, and Boris Magnusson. Compos: Composing oblivious services. In IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2019, Kyoto, Japan, March 11-15, 2019, pages 132–138, 2019.

[5] Isil Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdogan. Deployment and communication patterns in microservice architectures: A systematic literature review. J. Syst. Softw., 180:111014, 2021.

[6] Ian F. Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in SDN-OpenFlow networks. Computer Networks, 71:1–30, 2014.

[7] Ahmed Hazim Alhilali and Ahmadreza Montazerolghaem. Artificial intelligence based load balancing in SDN: A comprehensive survey. Internet Things, 22:100814, 2023.

[8] Ahmed Hussein Ali and Mahmood Zaki Abdullah. A survey on vertical and horizontal scaling platforms for big data analytics. International Journal of Integrated Engineering, 11(6), September 2019.

[9] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'10), pages 63–74, 2010.

[10] Dalal Alrajeh, Antoine Cailliau, and Axel van Lamsweerde. Adapting requirements models to varying environments. In Gregg Rothermel and Doo-Hwan Bae, editors, ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pages 50–61. ACM, 2020.

[11] Amazon. Amazon Web Services. IoT Core. https://aws.amazon.com/solutions/\implementations/iot-device-simulator/, 2021.

[12] Rashid Amin, Martin Reisslein, and Nadir Shah. Hybrid SDN networks: A survey of existing approaches. IEEE Communications Surveys and Tutorials, 20(4):3259–3306, 2018.

[13] Ahmed Amokrane, Rami Langar, Raouf Boutaba, and Guy Pujolle. Online flow-based energy efficient management in wireless mesh networks. In 2013 IEEE Global

Communications Conference, GLOBECOM 2013, Atlanta, GA, USA, December 9-13, 2013, pages 329–335. IEEE, 2013.

[14] Moussa Amrani, Fabian Gilson, Abdelmounaim Debieche, and Vincent Englebert. Towards user-centric DSLs to manage IoT systems. In Luís Ferreira Pires, Slimane Hammoudi, and Bran Selic, editors, Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017, pages 569–576, 2017.

[15] Ivan Dario Paez Anaya, Viliam Simko, Johann Bourcier, Noël Plouzeau, and Jean-Marc Jézéquel. A prediction-driven adaptation approach for self-adaptive sensor networks. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems SEAMS'14, pages 145–154, 2014.

[16] Apache JMeter. https://jmeter.apache.org/, 2023.

[17] G. Apostolopoulos, S. Kamat, D. Williams, R. Guérin, A. Orda, and T. Przygienda. QoS routing mechanisms and OSPF extensions. RFC, 2676:1–50, 1999.

[18] Victor Araujo, Karan Mitra, Saguna Saguna, and Christer Åhlund. Performance evaluation of FIWARE: A cloud-based IoT platform for smart cities. J. Parallel Distributed Comput., 132:250–261, 2019.

[19] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, pages 205–214. IEEE Computer Society, 2010.

[20] Istio Authors. Istio service mesh. https://istio.io, 2024.

[21] Prometheus Authors. https://prometheus.io, 2025.

[22] Abdelhadi Azzouni, Raouf Boutaba, and Guy Pujolle. Neuroute: Predictive dynamic routing for software-defined networks. In 13th International Conference on Network and Service Management, CNSM 2017, Tokyo, Japan, November 26-30, 2017, pages 1–6. IEEE Computer Society, 2017.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003, pages 164–177. ACM, 2003.

[24] André Bauer, Veronika Lesch, Laurens Versluis, Alexey Ilyushkin, Nikolas Herbst, and Samuel Kounev. Chamulteon: Coordinated auto-scaling of micro-services. In 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019, pages 2015–2025. IEEE, 2019.

[25] Jossekin Beilharz, Philipp Wiesner, Arne Boockmeyer, Lukas Pirl, Dirk Friedenberger, Florian Brokhausen, Ilja Behnke, Andreas Polze, and Lauritz Thamsen. Continuously testing distributed IoT systems: An overview of the state of the art. CoRR, abs/2112.09580, 2021.

[26] Ouafa Bentaleb, Adam S. Z. Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. Containerization technologies: taxonomies, applications and challenges. J. Supercomput., 78(1):1144–1181, 2022.

[27] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantzand Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN'14), pages 1–6, 2014.

[28] August Betzler, Carles Gomez, Ilker Demirkol, and Josep Paradells. CoAP congestion control for the internet of things. IEEE Communications Magazine, 54(7):154–160, 2016.

[29] Md Zakirul Alam Bhuiyan, Jie Wu, Guojun Wang, Tian Wang, and Mohammad Mehedi Hassan. e-Sampling: Event-sensitive autonomous adaptive sensing and low-cost monitoring in networked sensing systems. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 12(1):1:1–1:29, 2017.

[30] Andrea Bianco, Paolo Giaccone, Ahsan Mahmood, Mario Ullio, and Vinicio Vercellone. Evaluating the SDN control traffic in large ISP networks. In Proceedings of the 2015 IEEE International Conference on Communications (ICC'15), pages 5248–5253, 2015.

[31] Andrea Bianco, Paolo Giaccone, Reza Mashayekhi, Mario Ullio, and Vinicio Vercellone. Scalability of ONOS reactive forwarding applications in ISP networks. Computer Communications, 102:130–138, 2017.

[32] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, François-Xavier Jegeden, and Donghwan Shin. Digital twins are not monozygotic - cross-replicating ADAS testing in two industry-grade automotive simulators. In 14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, pages 383–393. IEEE, 2021.

[33] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. Computer Networks, 56(15):3531–3547, 2012.

[34] Athanassios Boulis. Castalia: A simulator for wireless sensor networks and body area networks. NICTA: National ICT Australia, 83:7, 2011.

[35] Sebastian Brandt, Klaus-Tycho Foerster, and Roger Wattenhofer. On consistent migration of flows in SDNs. In Proceedings of the 2016 Annual IEEE International Conference on Computer Communications (INFOCOM'16), pages 1–9, 2016.

[36] Antonio Brogi and Stefano Forti. QoS-Aware deployment of IoT applications through the fog. IEEE Internet Things J., 4(5):1185–1192, 2017.

[37] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. Cloud Computing Principles and Paradigms. Wiley Publishing, 2011.

[38] Diletta Romana Cacciagrano, Flavio Corradini, Rosario Culmone, Nikos Gorogiannis, Leonardo Mostarda, Franco Raimondi, and Claudia Vannucchi. Analysis and

verification of ECA rules in intelligent environments. Journal of Ambient Intelligence and Smart Environments, 10(3):261–273, 2018.

[39] Diletta Romana Cacciagrano and Rosario Culmone. IRON: reliable domain specific language for programming IoT devices. Internet Things, 9:100020, 2020.

[40] Rodrigo N. Calheiros, Rajiv Ranjan, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. CoRR, abs/0903.2525, 2009.

[41] Javier Cámara, Henry Muccini, and Karthik Vaidhyanathan. Quantitative verification-aided machine learning: A tandem approach for architecting self-adaptive IoT systems. In 2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020, pages 11–22, 2020.

[42] J. Anthony Capon. Elementary Statistics for the Social Sciences: Study Guide. Wadsworth Publishing Company, Belmont, CA, USA, 1991.

[43] Marcel Caria, Tamal Das, and Admela Jukan. Divide and conquer: Partitioning OSPF networks with SDN. In Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM'15), pages 467–474, 2015.

[44] H.A. Chan. Accelerated stress testing for both hardware and software. In Annual Symposium Reliability and Maintainability, 2004 - RAMS, pages 346–351, 2004.

[45] Betty H. C. Cheng, Andres J. Ramirez, and Philip K. McKinley. Harnessing evolutionary computation to enable dynamically adaptive systems to manage uncertainty.

In 1st International Workshop on Combining Modelling and Search-Based Software Engineering, CMSBSE@ICSE 2013, San Francisco, CA, USA, May 20, 2013, pages 1–6, 2013.

[46] Mung Chiang and Tao Zhang. Fog and IoT: An overview of research opportunities. IEEE Internet Things J., 3(6):854–864, 2016.

[47] Sheng-Hao Chiang, Jian-Jhih Kuo, Shan-Hsiang Shen, De-Nian Yang, and Wen-Tsuen Chen. Online multicast traffic engineering for software-defined networks. In Proceedings of the 2018 Annual IEEE International Conference on Computer Communications (INFOCOM'18), pages 414–422, 2018.

[48] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. Rpe Report, 142, 2005.

[49] Cisco. Cisco. OSPF design guide. Documentationathttps://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/7039-1.html, 2005.

[50] Zack Coker, David Garlan, and Claire Le Goues. SASS: Self-adaptation using stochastic search. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems SEAMS'15, pages 168–174, 2015.

[51] Rob Coltun, Dennis Ferguson, John Moy, and Acee Lindem. OSPF for IPv6. Internet Standard RFC 5340, Network Working Group, 2008.

[52] Cooja. Cooja simulator. https://anrg.usc.edu/contiki/index.php/Cooja_Simulator, 2016.

[53] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 3rd edition, 2009.

[54] IBM Corporation. An architectural blueprint for autonomic computing. Technical report, IBM Corporation, 2005.

[55] António Coutinho, Fabíola Greve, Cássio V. S. Prazeres, and João Cardoso. Fogbed: A rapid-prototyping emulation environment for fog computing. In 2018 IEEE International Conference on Communications, ICC 2018, Kansas City, MO, USA, May 20-24, 2018, pages 1–7. IEEE, 2018.

[56] Udhaya Kumar Dayalan, Rostand A. K. Fezeu, Timothy J. Salo, and Zhi-Li Zhang. Kaala: scalable, end-to-end, IoT system simulator. In NET4us '22: Proceedings of the ACM SIGCOMM Workshop on Networked Sensing Systems for a Sustainable Society, Amsterdam, The Netherlands, August 22, 2022, pages 33–38. ACM, 2022.

[57] Shuiguang Deng, Hailiang Zhao, Binbin Huang, Cheng Zhang, Feiyi Chen, Yinuo Deng, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Cloud-native computing: A survey from the perspective of services. Proceedings of the IEEE, 112(1):12–46, 2024.

[58] Byron DeVries and Betty H. C. Cheng. Using models at run time to detect incomplete and inconsistent requirements. In Proceedings of MODELS 2017 Satellite Events, Austin, TX, USA, September, 17, 2017, volume 2019 of CEUR Workshop Proceedings, pages 201–209. CEUR-WS.org, 2017.

[59] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing service level objectives for cloud services: Realities and myths. In 2019 IEEE International Conference on Autonomic Computing, ICAC 2019, Umeå, Sweden, June 16-20, 2019, pages 200–206. IEEE, 2019.

[60] Jasenka Dizdarevic, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. ACM Comput. Surv., 51(6):116:1–116:29, 2019.

[61] Docker Inc. https://www.docker.com, 2025.

[62] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. Scalable network virtualization in software-defined networks. IEEE Internet Comput., 17(2):20–27, 2013.

[63] Eclipse Foundation, Inc. Xtend v2.25.0. https://www.eclipse.org/Xtend/, 2021.

[64] Eclipse Foundation, Inc. Xtext v2.25.0. https://www.eclipse.org/Xtext/, 2021.

[65] Hanan Elazhary. Internet of things (IoT), mobile cloud, cloudlet, mobile IoT, IoT cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions. Journal of Network and Computer Applications, 128:105–140, 2019.

[66] Ahmed M. Elkhodary, Naeem Esfahani, and Sam Malek. FUSION: a framework for engineering self-tuning self-adaptive software systems. In Gruia-Catalin Roman and André van der Hoek, editors, Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, pages 7–16, 2010.

[67] Robert Feldt and Shin Yoo. Flexible probabilistic modeling for search based test data generation. In ICSE '20: 42nd International Conference on Software Engineering, SBST Workshop, Seoul, Republic of Korea, 27 June - 19 July, 2020, pages 537–540. ACM, 2020.

[68] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In Uwe Aßmann, Birgit Demuth, Thorsten Spitta, Georg Püschel, and Ronny Kaiser, editors, Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany, volume P-239 of LNI, pages 112–113, 2015.

[69] Abbas Dehghani Firouzabadi, Hakim Mellah, Orestes Manzanilla-Salazar, Reza Khalvandi, Vincent Therrien, Victor Boutin, and Brunilde Sansò. PIoT: A performance IoT simulation system for a large-scale city-wide assessment. IEEE Access, 11:56273–56286, 2023.

[70] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing OSPF weights. In Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000, pages 519–528. IEEE Computer Society, 2000.

[71] Bernard Fortz and Mikkel Thorup. Optimizing OSPF/IS-IS weights in a changing world. IEEE J. Sel. Areas Commun., 20(4):756–767, 2002.

[72] Bernard Fortz and Mikkel Thorup. Increasing internet capacity using local search. Comput. Optim. Appl., 29(1):13–48, 2004.

[73] Saurabh Kumar Garg and Rajkumar Buyya. Networkcloudsim: Modelling parallel applications in cloud simulations. In IEEE 4th International Conference on Utility and Cloud Computing, UCC 2011, Melbourne, Australia, December 5-8, 2011, pages 105–113, 2011.

[74] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer, 37(10):46–54, 2004.

[75] David Garlan, Bradley R. Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In Autonomic Computing and Networking, pages 31–55. Springer, 2009.

[76] Steven Gay, Renaud Hartert, and Stefano Vissicchio. Expect the unexpected: Sub-second optimization for segment routing. In Proceedings of the 2017 Annual IEEE International Conference on Computer Communications (INFOCOM'17), pages 1–9, 2017.

[77] Omid Gheibi, Danny Weyns, and Federico Quin. On the impact of applying machine learning in the decision-making of self-adaptive systems. In 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2021, Madrid, Spain, May 18-24, 2021, pages 104–110. IEEE, 2021.

[78] M. Ghobadi, S. Hassas Yeganeh, and Y. Ganjali. Rethinking end-to-end congestion control in software-defined networks. In Srikanth Kandula, Jitendra Padhye, Emin Gün Sirer, and Ramesh Govindan, editors, 11th ACM Workshop on Hot Topics in Networks, HotNets-XI, Redmond, WA, USA - October 29 - 30, 2012, pages 61–66. ACM, 2012.

[79] Tiago Gomes, P. Lopes, J. Alves, Pedro Mestre, Jorge Cabral, João L. Monteiro, and Adriano Tavares. A modeling domain-specific language for IoT-enabled operating systems. In IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society, Beijing, China, October 29 - November 1, 2017, pages 3945–3950, 2017.

[80] Oihane Gómez-Carmona, Diego Casado-Mansilla, Diego López-de-Ipiña, and Javier García-Zubía. Optimizing computational resources for edge intelligence through model cascade strategies. IEEE Internet Things J., 9(10):7404–7417, 2022.

[81] Carlos A González, Mojtaba Varmazyar, Shiva Nejati, Lionel C Briand, and Yago Isasi. Enabling model testing of cyber-physical systems. In International Conference on Model Driven Engineering Languages and Systems, pages 176–186. ACM, 2018.

[82] Google. Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo, 2024.

[83] Yingya Guo, Zhiliang Wang, Xia Yin, Xingang Shi, and Jianping Wu. Traffic engineering in SDN/OSPF hybrid network. In 2014 IEEE 22nd International Conference on Network Protocols, pages 563–568, 2014.

[84] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. Softw. Pract. Exp., 47(9):1275–1296, 2017.

[85] Evangelos Haleplidis, Kostas Pentikousis, Spyros G. Denazis, Jamal Hadi Salim, David Meyer, and Odysseas G. Koufopavlou. Software-defined networking (SDN): Layers and architecture terminology. Information RFC 7426, Internet Research Task Force (IRTF), 2015.

[86] Guangjie Han, Yuhui Dong, Hui Guo, Lei Shu, and Dapeng Wu. Cross-layer optimized routing in wireless sensor networks with duty cycle and energy harvesting. Wirel. Commun. Mob. Comput., 15(16):1957–1981, 2015.

[87] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In LASER Summer School, 2010.

[88] Mark Harman, Edmund Burke, John Clark, and Xin Yao. Dynamic adaptive search based software engineering. In Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'12), pages 1–8, 2012.

[89] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John B. Carter, and Aditya Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'16), pages 244–257, 2016.

[90] Baik Hoh, Marco Gruteser, Hui Xiong, and Ansaf Alrabady. Preserving privacy in GPS traces via uncertainty-aware path cloaking. In Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007, pages 161–171. ACM, 2007.

[91] Houshyar Honar Pajooh, Mohammed A. Rashid, Fakhrul Alam, and Serge Demidenko. IoT big data provenance scheme using blockchain on Hadoop ecosystem. Journal of Big Data, 8(1):114, 2021.

[92] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In Proceedings of the 2013 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'13), pages 15–26, 2013.

[93] Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. Practical efficient microservice autoscaling with QoS assurance. In HPDC '22: The 31st International Symposium on High-Performance Parallel and Distributed Computing, Minneapolis, MN, USA, 27 June 2022 - 1 July 2022, pages 240–252. ACM, 2022.

[94] Lu Hou, Shaohang Zhao, Xing Li, Periklis Chatzimisios, and Kan Zheng. Design and implementation of application programming interface for internet of things cloud. International Journal of Network Management, 27(3):e1936, 2017.

[95] Meitian Huang, Weifa Liang, Zichuan Xu, Wenzheng Xu, Song Guo, and Yinlong Xu. Dynamic routing for network throughput maximization in software-defined networks.

In Proceedings of the 2016 Annual IEEE International Conference on Computer Communications (INFOCOM'16), pages 1–9, 2016.

[96] Mark Hung. Leading the IoT. Documentation at https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf, 2017.

[97] Muhammad Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. DeltaIoT: A self-adaptive internet of things exemplar. In Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems SEAMS'17, pages 76–82, 2017.

[98] Paola Inverardi and Marco Mori. Feature oriented evolutions for context-aware adaptive systems. In Andrea Capiluppi, Anthony Cleve, and Naouel Moha, editors, Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), Antwerp, Belgium, September 20-21, 2010, pages 93–97, 2010.

[99] Ahmed A. Ismail, Haitham S. Hamza, and Amira M. Kotb. Performance evaluation of open source IoT platforms. In 2018 IEEE Global Conference on Internet of Things (GCIoT), 2018.

[100] Sharmin Jahan, Ian Riley, Charles Walter, Rose F. Gamble, Matt Pasco, Philip K. McKinley, and Betty H. C. Cheng. MAPE-K/MAPE-SAC: an interaction framework for adaptive systems with security assurance cases. Future Gener. Comput. Syst., 109:197–209, 2020.

[101] Manar Jammal, Taranpreet Singh, Abdallah Shami, Rasool Asal, and Yiming Li. Software defined networking: State of the art and research challenges. Computer Networks, 72:74–98, 2014.

[102] Pooyan Jamshidi, Javier Cámara, Bradley R. Schmerl, Christian Kästner, and David Garlan. Machine learning meets quantitative planning: enabling self-adaptation in autonomous robots. In Marin Litoiu, Siobhán Clarke, and Kenji Tei, editors, Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pages 39–50, 2019.

[103] Saeed Javanmardi, Mohammad Shojafar, Reza Mohammadi, Mamoun Alazab, and Antonio Caruso. An SDN perspective IoT-Fog security: A survey. Comput. Networks, 229:109732, 2023.

[104] Devki Nandan Jha, Khaled Alwasel, Areeb Alshoshan, Xianghua Huang, Ranesh Kumar Naha, Sudheer Kumar Battula, Saurabh Garg, Deepak Puthal, Philip James, Albert Y. Zomaya, Schahram Dustdar, and Rajiv Ranjan. IoTSim-Edge: A simulation framework for modeling the behavior of internet of things and edge computing environments. Software - Practice and Experience, 50:844–867, 2020.

[105] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. Computer, 36(1):41–50, 2003.

[106] Attila Kertész, Tamas Pflanzner, and Tibor Gyimóthy. A mobile IoT device simulator for IoT-Fog-Cloud systems. Journal of Grid Computing, 17(3):529–551, 2019.

[107] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. The Journal of Supercomputing, 62(3):1263–1283, 2012.

[108] John R. Koza, Martin A. Keane, Jessen Yu, Forrest H. Bennett, and William Mydlowec. Automatic creation of human-competitive programs and controllers by means of genetic programming. Genetic Programming and Evolvable Machines, 1(1):121–164, 2000.

[109] John R Koza and John R Koza. Genetic programming: on the programming of computers by means of natural selection, volume 1. MIT press, 1992.

[110] Heiko Koziolek, Sten Grüner, and Julius Rückert. A comparison of MQTT brokers for distributed IoT edge computing. In Software Architecture - 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14-18, 2020, Proceedings, volume 12292 of Lecture Notes in Computer Science, pages 352–368, 2020.

[111] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA, pages 259–268, 2007.

[112] Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. J. Syst. Softw., 126:1–16, 2017.

[113] Kubernetes. https://kubernetes.io, 2023.

[114] Santosh Kumar and Sonam Rai. Survey on transport layer protocols: TCP & UDP. International Journal of Computer Applications, 46(7):20–25, 2012.

[115] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets'10), pages 19:1–19:6, 2010.

[116] Apostolos Lazidis, Konstantinos Tsakos, and Euripides GM Petrakis. Publish–subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems. Internet of Things, 19:100538, 2022.

[117] Jia Li, Behrad Moeini, Shiva Nejati, Mehrdad Sabetzadeh, and Michael McCallen. A lean simulation framework for stress testing IoT cloud systems. IEEE Trans. Software Eng., 50(7):1827–1851, 2024.

[118] Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. Learning self-adaptations for IoT networks: A genetic programming approach. In SEAMS '22: IEEE/ACM 17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Pittsburgh, PA, USA, May 21 - 29, 2022, 2022.

[119] Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. Using genetic programming to build self-adaptivity into software-defined networks. ACM Trans. Auton. Adapt. Syst., 19(1):2:1–2:35, 2024.

[120] Jia Li, Shiva Nejati, Mehrdad Sabetzadeh, and Michael McCallen. A domain-specific language for simulation-based testing of IoT edge-to-cloud solutions. In Proceedings of the 25th International Conference on Model Driven Engineering Languages and

Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022, pages 367–378, 2022.

[121] Ying-Dar Lin, Hung-Yi Teng, Chia-Rong Hsu, Chun-Chieh Liao, and Yuan-Cheng Lai. Fast failover and switchover for link failures and congestion in software defined networks. In Proceedings of the 2016 IEEE International Conference on Communications (ICC'16), pages 1–6, 2016.

[122] Jianshu Liu, Shungeng Zhang, Qingyang Wang, and Jinpeng Wei. Coordinating fast concurrency adapting with autoscaling for SLO-oriented web applications. IEEE Trans. Parallel Distributed Syst., 33(12):3349–3362, 2022.

[123] Lin Liu, Jiantao Zhou, Xiaoyong Guo, and Rui-dong Qi. A method for calculating link weight dynamically by entropy of information in SDN. In 22nd IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD 2018, Nanjing, China, May 9-11, 2018, pages 535–540. IEEE, 2018.

[124] Locust. https://locust.io, 2023.

[125] Software Logistics. NuvIoT IoT simulator. https://www.nuviot.com, 2019.

[126] Manuel Ramírez López and Josef Spillner. Towards quantifiable boundaries for elastic horizontal scaling of microservices. In Companion Proceedings of The10th International Conference on Utility and Cloud Computing, page 35–40. Association for Computing Machinery, 2017.

[127] Sean Luke. Essentials of Metaheuristics. Lulu, second edition, 2013.

[128] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. Evol. Comput., 14(3):309–344, 2006.

[129] Nhan Ly-Trong, Chuong Dang-Le-Bao, Dang Huynh-Van, and Quan Le Trung. UiTiOt v3: A hybrid testbed for evaluation of large-scale IoT networks. In Proceedings of the Ninth International Symposium on Information and Communication Technology, SoICT 2018, Danang City, Vietnam, December 06-07, 2018, pages 155–162, 2018.

[130] Tyson Macaulay and Bryan L. Singer. RIoT Control: Understanding and Managing Risks and the Internet of Things. Elsevier, 2017.

[131] Yassine Maleh, Youssef Qasmaoui, Khalid El Gholami, Yassine Sadqi, and Soufyane Mounir. A comprehensive survey on SDN security: threats, mitigations, and future directions. J. Reliab. Intell. Environ., 9(2):201–239, 2023.

[132] Sathiya Kumaran Mani, Ramakrishnan Durairajan, Paul Barford, and Joel Sommers. An architecture for IoT clock synchronization. In Proceedings of the 8th International Conference on the Internet of Things, IOT 2018, Santa Barbara, CA, USA, October 15-18, 2018, pages 17:1–17:8, 2018.

[133] Bomin Mao, Zubair Md. Fadlullah, Fengxiao Tang, Nei Kato, Osamu Akashi, Takeru Inoue, and Kimihiro Mizutani. A tensor based deep learning technique for intelligent packet routing. In 2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017, pages 1–6. IEEE, 2017.

[134] Nicolas Marie-Magdelaine and Toufik Ahmed. Proactive autoscaling for cloud-native applications using machine learning. In GLOBECOM 2020 - 2020 IEEE Global Communications Conference, 2020.

[135] Nathan Marz and James Warren. Big Data: Principles and Best Practices of Scalable Real-Time Data Systems. Manning Publications Co., 2015.

[136] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining TCP congestion control. In Proceedings of the 1996 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'96), pages 281–291, 1996.

[137] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. MiL testing of highly configurable continuous controllers: scalable search using surrogate models. In International Conference on Automated Software Engineering (ASE), pages 163–174. ACM/IEEE, 2014.

[138] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures. In IEEE Fog World Congress, FWC 2017, Santa Clara, CA, USA, October 30 - Nov. 1, 2017, pages 1–6. IEEE, 2017.

[139] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. Openflow: enabling innovation in campus networks. Comput. Commun. Rev., 38(2):69–74, 2008.

[140] Claudio Menghi, Shiva Nejati, Lionel C. Briand, and Yago Isasi Parache. Approximation-refinement testing of compute-intensive cyber-physical models: an approach based on system identification. In ICSE '20: 42nd International Conference on Software Engineering, pages 372–384. ACM, 2020.

[141] Mininet. Mininet. https://mininet.org, 2021.

[142] Hao Mo, Ligu Zhu, Lei Shi, Songfu Tan, and Suping Wang. Hetsev: Exploiting heterogeneity-aware autoscaling and resource-efficient scheduling for cost-effective machine-learning model serving. Electronics, 12(1), 2023.

[143] Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Beyond discrete modeling: A continuous and efficient model for IoT. In Proceedings of the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'15), pages 90–99, 2015.

[144] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, pages 1–12, 2015.

[145] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, pages 122–132. IEEE, 2009.

[146] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon S. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings, volume 5301, pages 782–796, 2008.

[147] Brice Morin, Nicolas Harrand, and Franck Fleurey. Model-based software engineering to tame the IoT jungle. IEEE Software, 34(1):30–36, 2017.

[148] Zeinab Movahedi, Mouna Ayari, Rami Langar, and Guy Pujolle. A survey of autonomic network architectures and evaluation criteria. IEEE Commun. Surv. Tutorials, 14(2):464–490, 2012.

[149] Shiva Nejati. Next-generation software verification: An AI perspective. IEEE Software, 38(3):126–130, 2021.

[150] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in Kubernetes for elastic container orchestration. Sensors, 20(16), 2020.

[151] Fotis Nikolaidis, Manolis Marazakis, and Angelos Bilas. IOTier: A virtual testbed to evaluate systems for IoT environments. In 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, Melbourne, Australia, May 10-13, 2021, pages 676–683, 2021.

[152] Mohammad Noormohammadpour and Cauligi S. Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. IEEE Commun. Surv. Tutorials, 20(2):1492–1525, 2018.

[153] NS-3. NS-3.35. https://www.nsnam.org, 2021.

[154] Joao Paulo Karol Santos Nunes, Thiago Bianchi, , Yoshiyuki Iwasaki, and Elisa Yumi Nakagawa. State of the art on microservices autoscaling: An overview. Anais do XLVIII Seminário Integrado de Software e Hardware (SEMISH 2021), 2021.

[155] João Paulo Karol Santos Nunes, Shiva Nejati, Mehrdad Sabetzadeh, and Elisa Yumi Nakagawa. Self-adaptive, requirements-driven autoscaling of microservices. In Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2024, Lisbon, Portugal, April 15-16, 2024, pages 168–174. ACM, 2024.

[156] OMNeT++. OMNeT++. https://omnetpp.org/intro/, 2022.

[157] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, D. Heimhigner, G. Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intell. Syst., 14(3):54–62, 1999.

[158] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with COOJA. In LCN 2006, The 31st Annual IEEE Conference on Local Computer Networks, Tampa, Florida, USA, 14-16 November 2006, pages 641–648, 2006.

[159] CO Oyeniran, Adebunmi Okechukwu Adewusi, Adams Gbolahan Adeleke, Lucy Anthony Akwawa, and Chidimma Francisca Azubuko. Microservices architecture in cloud-native applications: Design patterns and scalability. Computer Science & IT Research Journal, 5(9):2107–2124, 2024.

[160] Oyekunle Claudius Oyeniran, Oluwole Temidayo Modupe, Aanuoluwapo Ayodeji Otitoola, Oluwatosin Oluwatimileyin Abiona, Adebunmi Okechukwu Adewusi, and Oluwatayo Jacob Oladapo. A comprehensive review of leveraging cloud-native technologies for scalability and resilience in software development. International Journal of Science and Research Archive, 2024.

[161] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. IEEE Trans. Cloud Comput., 7(3):677–692, 2019.

[162] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Architectural principles for cloud software. ACM Trans. Internet Techn., 18(2):17:1–17:23, 2018.

[163] Luis Hernán García Paucar and Nelly Bencomo. Knowledge base K models to support trade-offs for self-adaptation using Markov processes. In 13th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2019, Umea, Sweden, June 16-20, 2019, pages 11–16, 2019.

[164] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. J. Manag. Inf. Syst., 24(3):45–77, 2008.

[165] Tamas Pflanzner, Attila Kertész, Bart Spinnewyn, and Steven Latré. MobIoTSim: Towards a mobile IoT device simulator. In Muhammad Younas, Irfan Awan, and Joyce El Haddad, editors, 4th IEEE International Conference on Future Internet of Things and Cloud Workshops, FiCloud Workshops 2016, Vienna, Austria, August 22-24, 2016, pages 21–27, 2016.

[166] Andrey Pokhilko. UDPRequest. https://jmeter-plugins.org/wiki/UDPRequest/, 2023.

[167] Riccardo Poli and William B Langdon. Genetic programming with one-point crossover. In Soft Computing in Engineering Design and Manufacturing, pages 180–189. Springer, 1998.

[168] Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. In Evolutionary Computation, volume 6, pages 231–252. MIT Press, 1998.

[169] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. A field guide to genetic programming. Lulu. com, 2008.

[170] Amit M Potdar, Narayan D G, Shivaraj Kengond, and Mohammed Moin Mulla. Performance evaluation of docker container and virtual machine. Procedia Computer Science, 171:1419–1428, 2020.

[171] Konstantinos Poularakis, George Iosifidis, Georgios Smaragdakis, and Leandros Tassiulas. Optimizing gradual SDN upgrades in ISP networks. IEEE/ACM Transactions on Networking, 27(1):288–301, 2019.

[172] Olesia Pozdniakova, Dalius Mazeika, and Aurimas Cholomskis. Adaptive resource provisioning and auto-scaling for cloud native software. In Information and Software Technologies - 24th International Conference, ICIST 2018, Vilnius, Lithuania, October 4-6, 2018, Proceedings, volume 920 of Communications in Computer and Information Science, pages 113–129, 2018.

[173] Annisa Ayu Pramesti and Achmad Imam Kistijantoro. Autoscaling based on response time prediction for microservice application in Kubernetes. In 2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA), 2022.

[174] M. Priyadarsini, J. C. Mukherjee, P. Bera, S. Kumar, A. H. M. Jakaria, and M. Ashiqur Rahman. An adaptive load balancing scheme for software-defined network controllers. Comput. Networks, 164, 2019.

[175] Tariq Qayyum, Asad Malik, Muazzam Khattak, Osman Khalid, and Samee Khan. Fognetsim++: A toolkit for modeling and simulation of distributed fog environment. IEEE Access, PP:1–1, 10 2018.

[176] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: an intelligent fine-grained resource management framework for SLO-oriented microservices. In 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020, pages 805–825. USENIX Association, 2020.

[177] Federico Quin, Danny Weyns, Thomas Bamelis, Sarpreet Singh Buttar, and Sam Michiels. Efficient analysis of large adaptation spaces in self-adaptive systems using machine learning. In Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pages 1–12. ACM, 2019.

[178] Wajid Rafique, Lianyong Qi, Ibrar Yaqoob, Muhammad Imran, Raihan Ur Rasool, and Wanchun Dou. Complementing IoT services through software defined networking and edge computing: A comprehensive survey. IEEE Commun. Surv. Tutorials, 22(3):1761–1804, 2020.

[179] Andres J. Ramirez and Betty H. C. Cheng. Design patterns for developing dynamically adaptive systems. In Proceedings of the 2010 Workshop on Software Engineering for Adaptive and Self-Managing Systems SEAMS'10, pages 49–58, 2010.

[180] Brian Ramprasad, Marios Fokaefs, Joydeep Mukherjee, and Marin Litoiu. EMU-IoT - A virtual internet of things lab. In 2019 IEEE International Conference on Autonomic Computing, ICAC 2019, Umeå, Sweden, June 16-20, 2019, pages 73–83, 2019.

[181] Albert Rego, Sandra Sendra, José Miguel Jiménez, and Jaime Lloret. OSPF routing protocol performance in software defined networks. In Proceedings of the 2017 4th International Conference on Software Defined Systems (SDS'17), pages 131–136, 2017.

[182] G. Rétvári, F. Németh, R. Chaparadza, and R. Szabó. OSPF for implementing self-adaptive routing in autonomic networks: A case study. In Yacine Ghamri-Doudane, editor, Modelling Autonomic Communications Environments, Fourth IEEE International Workshop, MACE 2009, Venice, Italy, October 26-27, 2009. Proceedings, volume 5844, pages 72–85. Springer, 2009.

[183] Arthur Rodrigues, Ricardo Diniz Caldas, Genaína Nunes Rodrigues, Thomas Vogel, and Patrizio Pelliccione. A learning approach to enhance assurances for real-time self-adaptive systems. In Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2018, Gothenburg, Sweden, May 28-29, 2018, pages 206–216, 2018.

[184] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst., 4(2):14:1–14:42, 2009.

[185] Theresia Ratih Dewi Saputri and Seok-Won Lee. The application of machine learning in self-adaptive systems: A systematic literature review. IEEE Access, 8:205948–205967, 2020.

[186] M. Scheffer, J.M. Baveco, D.L. DeAngelis, K.A. Rose, and E.H. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. Ecological Modelling, 80(2):161–170, 1995.

[187] Eric O. Scott and Sean Luke. ECJ at 20: toward a general metaheuristics toolkit. In Proceedings of the Genetic and Evolutionary Computation Conference Companion,

GECCO 2019, Prague, Czech Republic, July 13-17, 2019, pages 1391–1398. ACM, 2019.

[188] SednaLab. IoTECS. https://github.com/sednalab/IoTECS/tree/main, 2023.

[189] Joanna Sendorek, Tomasz Szydlo, and Robert Brzoza-Woch. Software-defined virtual testbed for IoT systems. Wireless Communications and Mobile Computing, 2018:1068261:1–1068261:11, 2018.

[190] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, Chetan Arora, and Frank Zimmer. Dynamic adaptation of software-defined networks for IoT systems: a search-based approach. In SEAMS '20: IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Seoul, Republic of Korea, 29 June - 3 July, 2020, pages 137–148, 2020.

[191] Parminder Singh, Pooja Gupta, Kiran Jyoti, and Anand Nayyar. Research on auto-scaling of web applications in cloud: Survey, trends and future directions. Scalable Comput. Pract. Exp., 20:399–432, 2019.

[192] Manfred Sneps-Sneppe and Dmitry Namiot. On web-based domain-specific language for internet of things. CoRR, abs/1505.06713, 2015.

[193] Hui Song, Rustem Dautov, Nicolas Ferry, Arnor Solberg, and Franck Fleurey. Model-based fleet deployment of edge computing applications. In MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, pages 132–142, 2020.

[194] Dipa Soni and Ashwin Makwana. A survey on MQTT: a protocol of internet of things (IoT). In International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017), volume 20, pages 173–177, 2017.

[195] Kachane Sonklin, Charles Wang, Dhammika Jayalath, and Yanming Feng. Connected-vehicle data exchanges and positioning computing based on the publish-subscribe paradigm. Journal of Computer and Communications, 07:82–93, 01 2019.

[196] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. In Second International Conference on Fog and Mobile Edge Computing, FMEC 2017, Valencia, Spain, May 8-11, 2017, pages 39–44, 2017.

[197] Thierry Sotiropoulos, Hélène Waeselynck, Jérémie Guiochet, and Félix Ingrand. Can robot navigation bugs be found in simulation? An exploratory study. In Proc. of the 2017 IEEE International Conference on Software Quality, Reliability and Security, pages 150–159, 2017.

[198] Michael Stein, Alexander Frömmgen, Roland Kluge, Frank Löffler, Andy Schürr, Alejandro Buchmann, and Max Mühlhäuser. TARL: Modeling topology adaptations for networking applications. In Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems SEAMS'16, pages 57–63, 2016.

[199] Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. Fogify: A fog computing emulation framework. In 5th IEEE/ACM

Symposium on Edge Computing, SEC 2020, San Jose, CA, USA, November 12-14, 2020, pages 42–54, 2020.

[200] Matthias Tichy, Jakob Pietron, David Mödinger, Katharina Juhnke, and Franz J. Hauck. Experiences with an internal DSL in the IoT domain. In STAF 2020 Workshop Proceedings: 4th Workshop on Model-Driven Engineering for the Internet-of-Things, 1st International Workshop on Modeling Smart Cities, and 5th International Workshop on Open and Original Problems in Software Language Engineering co-located with Software Technologies: Applications and Foundations federation of conferences (STAF 2020), Bergen, Norway, June 22-26, 2020, pages 22–34, 2020.

[201] Hong Linh Truong. Using IoTCloudSamples as a software framework for simulations of edge computing scenarios. Internet Things, 14:100383, 2021.

[202] András Vargha and Harold D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, 25(2):101–132, 2000.

[203] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. Challenges and opportunities in edge computing. In 2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, New York, NY, USA, November 18-20, 2016, pages 20–26. IEEE Computer Society, 2016.

[204] Rafael Vaño, Ignacio Lacalle, Piotr Sowiński, Raúl S-Julián, and Carlos E. Palau. Cloud-native workload orchestration at the edge: A deployment review and future directions. Sensors, 23(4), 2023.

[205] Christian Veenhuis. Structure-based constants in genetic programming. In Luís Correia, Luís Paulo Reis, and José Cascalho, editors, Progress in Artificial Intelligence - 16th Portuguese Conference on Artificial Intelligence, EPIA 2013, Angra do Heroísmo, Azores, Portugal, September 9-12, 2013. Proceedings, volume 8154 of Lecture Notes in Computer Science, pages 126–137. Springer, 2013.

[206] Tao Wang, Fangming Liu, and Hong Xu. An efficient online algorithm for dynamic SDN controller assignment in data center networks. IEEE/ACM Trans. Netw., 25(5):2788–2801, 2017.

[207] Philip Wette, Martin Dräxler, and Arne Schwabe. MaxiNet: Distributed emulation of software-defined networks. In 2014 IFIP Networking Conference, Trondheim, Norway, June 2-4, 2014, pages 1–9. IEEE Computer Society, 2014.

[208] Danny Weyns. An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. Wiley, 2021.

[209] Danny Weyns, Omid Gheibi, Federico Quin, and M. Jeroen Van Der Donckt. Deep learning for effective and efficient reduction of large adaptation spaces in self-adaptive systems. CoRR, abs/2204.06254, 2022.

[210] Wireshark Foundation. Wireshark v3.6.0. https://www.wireshark.org/, 2021.

[211] Junbi Xiao, Song Chen, and Mengmeng Sui. The strategy of path determination and traffic scheduling in private campus networks based on SDN. Peer-to-Peer Netw. Appl., 12(2):430–439, 2019.

[212] Shuaiyu Xie, Jian Wang, Bing Li, Zekun Zhang, Duantengchuan Li, and Patrick C. K. Hung. PBScaler: A bottleneck-aware autoscaling framework for microservice-based applications. IEEE Trans. Serv. Comput., 17(2):604–616, 2024.

[213] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, volume 37 of JMLR Workshop and Conference Proceedings, pages 2048–2057, 2015.

[214] Xuezhi Zeng, Saurabh Kumar Garg, Peter Strazdins, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, and Rajiv Ranjan. IOTSim: A simulator for analysing IoT applications. Journal of Systems Architecture: Embedded Software Design, 72:93–107, 2017.

[215] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2025. https://arxiv.org/abs/2303.18223.

[216] Liehuang Zhu, Md. Monjurul Karim, Kashif Sharif, Chang Xu, Fan Li, Xiaojiang Du, and Mohsen Guizani. SDN controllers: A comprehensive analysis and performance evaluation study. ACM Comput. Surv., 53(6):133:1–133:40, 2021.

[217] Yuan Zuo, Yulei Wu, Geyong Min, and Laizhong Cui. Learning-based network path planning for traffic engineering. Future Gener. Comput. Syst., 92:59–67, 2019.