Towards Orchestration in the Cloud-Fog Continuum

Xavier Jesus Merino Aguilera

Follow this and additional works at: https://repository.fit.edu/etd

Part of the Computer Engineering Commons
Towards Orchestration in the Cloud-Fog Continuum

by

Xavier Jesus Merino Aguilera

Master of Science
Computer Engineering
Florida Institute of Technology
2018

Bachelor of Science
Computer Engineering
Florida Institute of Technology
2016

A dissertation
submitted to the College of Engineering and Science
at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Engineering

Melbourne, Florida
December, 2022
© Copyright 2022 Xavier Jesus Merino Aguilera
All Rights Reserved

The author grants permission to make single copies.
We the undersigned committee hereby approve the attached dissertation

Towards Orchestration in the Cloud-Fog Continuum

by Xavier Jesus Merino Aguilera

Carlos E. Otero, Ph.D.
Professor
Computer Engineering and Sciences
Major Advisor

Munevver Subasi, Ph.D.
Associate Professor
Mathematical Sciences

Samuel P. Kozaitis, Ph.D.
Professor
Computer Engineering and Sciences

Josko Zec, Ph.D.
Associate Professor
Computer Engineering and Sciences

Philip J. Bernhard, Ph.D.
Associate Professor and Department Head
Computer Engineering and Sciences
Abstract

Title:
Towards Orchestration in the Cloud-Fog Continuum

Author:
Xavier Jesus Merino Aguilera

Major Advisor:
Carlos E. Otero, Ph.D.

The proliferation of the Internet-of-Things has raised demand for computing, storage, and network resources. The cloud model is ill-equipped to handle the volume and variety of data travelling to and from the cloud’s core as more data is generated and consumed at the network’s edge. Some applications necessitate low-latency connectivity and geographical awareness, highlighting the cloud’s centralization shortcomings. By localizing resources, minimizing bandwidth utilization, and lowering latency, the fog and edge layers are proposed to circumvent these limitations. At these layers, resource orchestration is crucial because poor resource management has an impact on service delivery. The aim of this study is to determine the viability of using cloud-native tools at the edge and fog layers by examining the overhead incurred by orchestration services in areas like networking, computing, time management, and software maintenance. Communication-brokering orchestration services impose prohibitive overheads, which may reduce the capacity of these layers to process loads.
# Table of Contents

Abstract ................................................................. iii

List of Figures ................................................................... viii

List of Tables .................................................................... ix

Acknowledgments ........................................................... x

Dedication ........................................................................ xi

1 Introduction ...................................................................... 1

  1.1 Motivation ................................................................... 2

  1.2 Research Scope and Organization ................................. 4

2 Background ...................................................................... 7

  2.1 From the Data Center to the Cloud ............................... 7

    2.1.1 Containers ......................................................... 10

    2.1.2 Container Runtimes and Engines ......................... 11

    2.1.3 Container Orchestrators ..................................... 13

  2.2 From the Cloud to the Fog and the Edge ..................... 16

  2.3 Cloud-Fog Orchestration ........................................... 18

    2.3.1 Orchestration in the Cloud Layer ......................... 18

    2.3.2 Orchestration in the Fog / Edge Layers ................. 20
2.3.3 Fog / Edge Orchestration Requirements .......................... 21
2.3.4 Evolution of Orchestration Services .............................. 22

3 Reference Architecture .................................................. 25
3.1 Design and Implementation ............................................ 25
  3.1.1 Network Fabric ................................................... 26
  3.1.2 Workload Orchestrator .......................................... 29
  3.1.3 Identity and Secret Manager ................................. 30
3.2 Software Bill of Materials ............................................ 31
  3.2.1 Components ...................................................... 31
    3.2.1.1 Infrastructure Components ............................ 31
    3.2.1.2 End User Components ................................. 32
  3.2.2 Services .......................................................... 33
    3.2.2.1 Infrastructure Services ............................ 33
    3.2.2.2 End User Services ................................. 33

4 Impact on Network Resources .......................................... 35
  4.1 Data Collection ................................................... 35
  4.2 Comparison Methodology .......................................... 37
  4.3 Results and Discussion .......................................... 38
  4.4 Related Works ..................................................... 43

5 Impact on Compute Resources .......................................... 46
  5.1 Data Collection ................................................... 47
  5.2 Comparison Methodology .......................................... 47
  5.3 Results and Discussion .......................................... 47
  5.4 Related Works ..................................................... 50
6 Impact on the Perception of Time ........................................ 52
  6.1 Additional Background ........................................... 55
    6.1.1 Hardware Clock Sources ................................... 55
    6.1.2 Linux Timekeeping ......................................... 55
    6.1.3 Virtualization and Time ................................... 56
  6.2 Reference Implementation ................................. 57
  6.3 Data Collection .................................................. 59
  6.4 Comparison Methodology ...................................... 62
  6.5 Results and Discussion ....................................... 63
    6.5.1 Sleep Inducing System Calls ......................... 63
    6.5.2 Time Reporting System Calls ......................... 67
    6.5.3 Timer System Calls ..................................... 70
  6.6 Related Works ................................................... 72

7 Impact on Software Maintenance ................................. 76
  7.1 Additional Background ....................................... 78
    7.1.1 Debugging Practice Advances .......................... 78
  7.2 Design and Implementation ............................... 81
    7.2.1 Reference Deployment .................................. 82
    7.2.2 Reference Workflow .................................... 83
    7.2.3 Reference Implementation ....................................
      7.2.4 Limitations ............................................. 88
  7.3 Data Collection .................................................. 90
  7.4 Prediction Modeling Methodology ....................... 91
  7.5 Results and Discussion ................................... 94
  7.6 Related Works .................................................... 97
List of Figures

3.1 Reference Architecture ........................................ 27

4.1 Network Impact from Orchestration Services .................. 39

5.1 CPU Utilization Across Service Levels .......................... 48

6.1 Workflow to Implement Time Namespace in Container .......... 59

6.2 Execution Time of System Calls .............................. 64

7.1 Reference Deployment ............................................ 83

7.2 Sequence Diagram for Checkpoint Process ...................... 86

7.3 Regression Plots ................................................. 94

7.4 Permutation Importance on Test Set .......................... 95

7.5 Checkpoint Time vs. Disk Speed ............................... 96
List of Tables

2.1 Fog/Edge Orchestrator Requirements ............................................ 22
3.1 Infrastructure Components ............................................................ 32
3.2 End User Components ................................................................. 33
3.3 Infrastructure Services ................................................................. 34
3.4 End User Services ....................................................................... 34
4.1 Network Metrics Collected ............................................................. 37
4.2 Network Metrics Comparison Results ............................................. 40
5.1 Compute Metrics Collected ............................................................. 47
5.2 Compute Metrics Comparison Results ............................................. 48
6.1 Available Clocks in Linux as of Kernel 5.13 ....................................... 56
6.2 Time Related System Calls ............................................................ 57
6.3 Time System Call Metrics Collected .............................................. 60
6.4 Time Metrics Comparison Results .................................................. 65
7.1 Amazon EC2 Instance Types Tested ................................................. 92
7.2 Software Checkpoint Metrics Collected .......................................... 93
Acknowledgements

I would like to express my deepest appreciation to Dr. Carlos Otero. My academic journey has been long, but through it all, Dr. Otero has always been there to provide timely advice and words of encouragement. I remember fondly the day that I stepped into his lab. I was fascinated and scared at the same time. I was excited about the research projects but scared that I would be unable to complete them. Dr. Otero made me feel welcomed, offered all the resources I needed for my growth, and genuinely cared about me on a personal level. I wouldn’t have gotten this far without his mentorship.

I’d also like to thank my family for their support over the years. I am grateful to my parents for their support and prayers since the day I left Ecuador. It’s been a long time, yet their confidence in me has never faltered. I am grateful to my brothers for providing emotional support at difficult times. I value the time I’ve spent with them, whether on the phone or in person. They never fail to make me laugh.

I am also grateful to Zachary Johnson. Thank you for diverting my attention away from my work when I needed it. For cracking Linux jokes, bringing me food, and staying up late with me. If it hadn’t been for your company, the Covid-19 pandemic would have driven me insane.

I’d like to thank God for all of the opportunities he’s given me, for putting all of these individuals in my life, for giving me the strength to keep going, and for his ongoing blessings.
Dedication

To my mentor, Dr. Carlos Otero, who always made time for me, even when life got in the way.

To my parents, who fueled my dreams and appetite for technology from an early age.
Chapter 1

Introduction

The growth of the Internet-of-Things has led to a rise in the need of computing power, storage, and network resources. As more data are being generated and consumed at the edge of the networks, the cloud model that enabled the affordable, on-demand, lease of these resources is ill-fitted to handle the volume and variety of data traveling to the cloud and back.

Cloud datacenters empower services with a variety of resources, however, the centralization of these resources results in data traveling further away from the user in order to be processed. Cloud-native applications may not be sensitive to the delays that stem from leveraging the cloud, but newer generation of applications (e.g., autonomous vehicles, remote monitoring, predictive maintenance, augmented reality, real-time gaming, telesurgery, etc.) showcase the limitations of the cloud due to their strict low-latency and location awareness requirements. In the absence of these quality-of-service (QoS) guarantees, services may be interrupted, degraded, or even malfunction.

To overcome these limitations, traditionally cloud-hosted resources are being moved closer to the end user, lowering latency and boosting location awareness. These expansions of the cloud, also known as fog and edge computing, aim to deliver resources
(e.g., compute, networking, and storage) to the network’s edge, enabling the building of scalable infrastructures near the end user.

1.1 Motivation

Adoption of the fog and edge layers is not without challenges. To properly allocate and manage resources in the cloud, orchestration technologies were necessary. The expansion to the edge and the fog broadens the scope of such technologies and presents new orchestration challenges.

An orchestrator, at its heart, is a centralized entity that oversees the lifecycle of an application that has been divided into microservices but is designed to run as a single logical entity, as well as the application’s interface with the underlying infrastructure. While orchestration services are widely used in industry because of their capacity to reduce the operational burden of clusters and services, considerable progress remains to be made to expand orchestrators’ capabilities to the fog and the edge. Furthermore, little is known about the performance impact of the features now employed in the cloud to conduct these jobs when they are carried over to the fog and edge layers.

Because orchestrators are involved in many parts of a deployment and rely on many virtualization strategies, they may accidentally introduce overhead in the environments in which they operate. The overhead may be prohibitively expensive at the fog and the edge, where resources are scarce. In comparison to the cloud, less research has focused on the impact on resource utilization necessary to orchestrate services in these contexts. This is not to argue that some components of cloud research are not applicable at the edge and fog layers, but rather that there is a gap in the literature that addresses issues that are critical in these mobile and dynamic environments. This chapter identifies various areas that require further investigation to evaluate whether cloud technology
contributions can be applied at the edge, and if so, with what overhead.

Traditional computing resources such as CPU, RAM, and networking capabilities have been investigated in cloud environments in an effort to improve resource use and reduce recurring infrastructure expenses. The key reason for being aware of these resources is cost, as the cloud’s pricing model takes into account the number of vCPUs, RAM, and network capacity assigned to a node. At the edge and fog layers, it is not a matter of pricing but rather reliability. Being in a limited environment means that resources must be used optimally to ensure service survivability. For example, CPU utilization is important because, while the environment is dynamic, elasticity is limited by locally available resources, unlike the cloud, which can use servers that are globally dispersed. Running out of allocated cloud resources necessitates upsizing infrastructure and incurring more expenses, whereas at the edge or fog, it may result in service interruptions or a reduction in projected service quality.

One of the most critical elements at the edge and fog layers is the perceived quality of service by the end user. It is insufficient to keep track of how computing resources are being used. Deployments in these layers necessitate operators guaranteeing the perceived bandwidth, latency, and security of a service. These factors, while significant in the cloud, take precedence when deploying at the edge and fog layers. In the cloud, latency and bandwidth are typically best estimated between two cloud endpoints; however, at the edge/fog, where communication links and devices may differ, there is a need to quantify network characteristics to adapt to diverse services and their demands.

Orchestrators are uniquely suited to respond to changing network environments by evicting services from their hosts and relocating them in others that can ensure their QoS. Unfortunately, this process, known as migration, results in services losing their perspective of time and refusing service. As such, there is a need for a workflow that
can guarantee a service’s perception of time across hosts, as well as a study of the overhead incurred by such an approach.

Furthermore, as services migrate among hosts in quest of greater QoS assurances, it becomes more difficult to localize faults when errors occur. As a result, there is a need for reproducing buggy services, their dependencies, and their environment to aid in the debugging process. Although these challenges have been noticed with orchestrated loads in the cloud, they are worsened by the fog and edge layers’ dynamic and diverse environment (i.e., various communication channels, architectures, and computation capabilities).

1.2 Research Scope and Organization

This section provides an outline for the remainder of the document and defines the areas on which each chapter focuses.

- Chapter 2 provides a brief history of the datacenter, how it developed into the cloud, and the ongoing shift to the fog and edge layers. It also explains the changes in software engineering that were required to fully leverage the cloud, as well as the resulting need for software orchestration. It describes the present state of the art in cloud orchestration, as well as the evolution of orchestration services needed to approach the challenges in the fog and edge layers.

- Chapter 3 identifies the requirements needed to orchestrate loads in the Cloud-Fog or Edge-to-Cloud continuum and proposes an architecture, built on available, open-source, components, that orchestrates loads with consideration to their geographical needs. In doing so, it provides several levels of commonly available cloud features (e.g., DNS-like service discovery, service mesh, health checks, encryption-as-a-service, among others) for use at the fog and the edge.
• Chapter 4 explores the overhead in the networking services when leveraging the features provided by orchestrators (mentioned above) to the deployed services. It focuses on service-to-service communication and the impact on quality-of-service, which are major concerns at the edge and fog layers at varying usage of infrastructure features. In other words, it seeks to evaluate whether there is any perceptible overhead in terms of perceived bandwidth or latency when managing networking services at the edge using cloud-based techniques, and if so, quantify that overhead to assess whether it could significantly affect deployed loads.

• Chapter 5 expands on Chapter 4 by attempting to quantify the overhead in processing power consumption. Given that elasticity is not an issue in the cloud, several technologies designed to manage networking services may consume excessive processing power in the constrained devices deployed at the edge and fog layers. As such, Chapter 5 seeks to determine if the various infrastructure services provided by cloud tools result in significant CPU overheads that may impede service delivery at the edge and in the fog, and if so, by how much.

• Chapter 6 and Chapter 7 depart from typical infrastructure resource monitoring to investigate the impact that orchestration at the edge and fog layers have on an application’s time perception and the challenges arising in software maintenance. In other words, these chapters delve into additional difficulties that were previously unnoticed in the cloud but have arisen as hot topics with the migration of services from the cloud to the edge.

• Chapter 6 investigates the influence of embedding services in virtual time on their ability to separate their sense of time (and its progression) from their hosts. As a result, unlike the existing state-of-the-art, it enables live migration of services without the risk of service interruption. This chapter describes the overhead
associated with utilizing virtual time and categorizes the impact based on the sort of behavior requested by the application and often supplied by the operating system via system calls.

- Chapter 7 focuses on the difficulties of software maintenance at the fog and edge levels. Interaction-based problems have emerged as the most common source of software bugs as a result of the adoption of these cloud extensions. Because of the concurrent behavior of several microservices, the variety of hardware, and the combinatorial expansion of existing standards and protocols, localizing the issue and thus debugging is a daunting undertaking for an operator. Furthermore, because microservices at the edge are dynamic, placing breakpoints on locally running copies of an application is not a viable method because the service state in production may differ or have been reached via a different execution path. While orchestration allows for automated deployment, the level of automation and decentralization makes it more difficult for operators to discover faults. This chapter describes a workflow for limiting the impact on the software maintenance process, as well as the overhead required to do so successfully with edge loads.

- Chapter 8 provides a brief review of the current orchestration challenges at the fog and edge layers as well as a summary of the prior chapters’ results and their implications for orchestrated environments.
Chapter 2

Background

2.1 From the Data Center to the Cloud

While the data center traces its roots back to the 1940s, with the need to calculate artillery fire during World War II and other war-related efforts [1], most would not characterize these primitive computation rooms as modern data centers. They were often complex, costly, prone to overheating, and suffered constant mechanical failures. With the advent of the transistor, computing technology was made more readily available to large enterprises in the commercial sector [2]. Without network connectivity, these islands of computing power would often receive data in batches and print out the results needed [3]. The need to constantly operate a service that was reliable, power efficient, and well maintained proved a challenge for many organizations. As such, many opted to outsource their computational needs rather than invest in their on-premises capabilities.

In the 1990s, the advent of communication technologies and the Internet resulted in a paradigm shift in computing [3]. The development of the client-server model slowly replaced mainframe rooms with microprocessor-based computers acting as servers,
gradually laying the foundation for modern data centers. Investment in data centers skyrocketed during the 90s and, while most of the capital invested was lost in the dot-com bubble [4], the infrastructure and software remained. As a result, the attention shifted to how to affordably expand access to the technology by optimizing hardware and resource usage (i.e., power, cooling, and cost). Virtualization was one such option for enhancing resource usage, and while the concept was not novel at the time, it was significant in cutting organizations’ IT capital and operational expenditures. When combined with additional developments in hardware, virtual desktop infrastructures, and application virtualization, data center sharing (i.e., colocation) became viable, and the concept of the cloud emerged [5].

Today, practically all corporations recognize the advantages of the cloud, and most firms, like in the days of the mainframe, prefer to avoid the cost and technical knowledge required to manage their own data centers. As a point of perspective, the average large-scale data center costs $215 million to construct [6]. While IT equipment accounts for the majority of the expense, finding a suitable location with the sufficient infrastructure required to supply energy, water, and Internet connectivity is frequently difficult. The acquisition of land and construction operations are costly and may take years to complete. Smaller scale data centers, which typically consist of less than 10 commercial racks and are usually collocated, have monthly expenditures of roughly $6000, without including maintenance or manpower [7].

Regardless of size, the cloud reduces capital expenditure on on-premises infrastructure as well as recurring expenditures for maintenance and updates. The increased availability of workloads, combined with the cloud’s scalability and flexibility, makes the use of Cloud Service Providers (CSPs) a sound strategy for a faster time-to-market while allowing customers to focus on their core-business rather than the IT operations required to support their digital infrastructure. In some cases, the security compliance
of some pre-built cloud infrastructures is an appealing proposition for firms without the IT competence to design comparable capabilities.

In general, cloud computing has evolved into a concept for providing on-demand, metered access to a pool of computer resources through the network [8]. These resources might be raw computer power, storage, bandwidth, networks, or even digital services. CSPs provide elastic access to these resources for a cost, enabling the leasing of infrastructure (IaaS), platforms (PaaS), and applications (SaaS). Cloud computing has grown significantly in recent years, and it is expected to be a $832 billion market by 2025 [9]. This widespread acceptance is powered by frictionless access to cost-effective resources, simplicity of provisioning, and rapid production deployments [10].

The introduction of cloud computing spurred yet another paradigm change in software engineering. Until this point, most applications had been monolithic in nature. A monolithic design requires that all modules in an application be tightly coupled in a single, self-contained entity. In practice, this corresponds to a single codebase that is deployed. Even in client-server applications, the monolithic architecture mandates that the server be one codebase and delivered as a whole unit. Because bottlenecks in the modules cannot be separated, scaling monolithic applications necessitates deploying more instances of the entire codebase [11], potentially demanding more resources than necessary. As a result, resources are overprovisioned, update cycles are lengthy, and flexibility is inadequate [12].

In contrast, cloud applications are divided into microservices. Traditional programs assumed that their needs could be predicted at deployment: the number of users, the type of devices, and the infrastructure used [13]. Cloud applications do not make such assumptions. Instead of relying on load forecasting, they can accommodate any load through scalability. This new generation of applications is based on microservices, which are compact, decoupled, and autonomous services that expose their endpoints.
via web services or APIs [14]. As a result, scaling operations necessitates an increase in the deployed instances of the limiting microservice rather than the overall architecture.

The dynamic nature of microservices results in ever-growing and shrinking deployments, which complements the cloud’s scalability and on-demand resource utilization approach. Furthermore, research demonstrates that microservices minimize deployment time and effort while allowing for continuous application integration with minimal downtime [15]. This is because each component in a microservice design has a small scope, allowing for rapid release cycles. Companies such as Google, Facebook, Netflix, and Mozilla have abandoned traditional protracted release cycles in order to remain competitive and provide end users with faster access to cutting-edge innovations [16].

2.1.1 Containers

As the cloud became more prevalent and in demand, it became important to break apart codebase monoliths in order to optimize the usage of leased resources. As a result, containers became the preferred technology for developing microservices based on isolated modules from an initial software monolith. While this was previously conceivable, prior virtualization attempts, notably virtual machines, had a high resource footprint, making it costly to adopt these techniques from both a monetary and performance standpoint. Containers enabled developers to package a microservice into a small portable image and provided an alternative to traditional, hypervisor-based virtualization by being a lightweight abstraction [17][18] with enforced isolation, increasing resource utilization, and allowing deployment in a variety of environments without modification. As a result, containers have emerged as the preferred deployment technique for deploying cloud-native applications.

Containers benefit from lightweight, scalable, and high-performance virtualization techniques at the operating system level, giving them a distinct edge over slow-to-boot
virtual machines [19]. Furthermore, in terms of image size, deployment time, and rolling updates, containers surpass virtual machines. This enables denser deployments with less overhead in CPU consumption [19][20]. Containers have gained in popularity as a result of these benefits, as well as the dependability, scalability, and flexibility required to operate on the cloud. The Cloud Native Computing Foundation (CNCF) has studied the cloud environment throughout time and observed a 300% adoption rate between 2016 and 2020, with 92% of respondents believing that containers are the new norm [21]. Containers make a lot of sense given what they provide because they are effective tools for packaging, distributing, and isolating software in a dependable and repeatable manner [22].

Linux containers are made up of orthogonal Linux primitives like namespaces and control groups (cgroups). A namespace is a Linux kernel feature that allows an application to have its own view of a resource that is distinct from the global resources [23]. A container can be limited to utilize a specified portion of its available resources (e.g., CPU, RAM, PIDs, RDMA, block I/O, etc.) by configuring control groups, which allow monitoring and limiting resource consumption [24]. This makes containers extremely versatile and sophisticated. An application, for example, can be deployed in a container that is separated from the rest of the system, while sharing only the network namespace with another container for network traffic inspection. Several container deployment and resource sharing strategies are evolving into their own deployment design patterns. In general, containers’ flexibility to mix and match features has made them adaptable to nearly any deployment.

2.1.2 Container Runtimes and Engines

Container runtimes automate the process of configuring the underlying Linux primitives, eliminating the need to manually spawn applications in namespaces, place them
in control groups, and enable security modules to restrict behavior. Because runtimes make this simple, by enabling repeatable and consistent deployments, and are lighter than virtual machines, their popularity grew swiftly.

In the early days, several projects aimed to create the tooling necessary to facilitate their adoption. Linux Containers (LXC) [25], Google’s Let Me Contain That For You (LMCTFY) [26], systemd-nspawn [27], Docker [28], and rkt [29] were among these efforts. The emphasis of these runtimes differed. LXC, for example, focuses on system containers, which are similar to virtual machines, whereas systemd-nspawn focuses on executing namespaced processes with cgroup management. Docker, which subsequently became the de-facto standard, initially used LXC with the goal of making it more accessible to developers and would later develop their own, libcontainer. LMCTFY was Google’s internal container runtime, however it was discontinued when libcontainer was open sourced and became part of the Open Container Initiative (OCI) [30]. When compared to other runtimes, CoreOS’s rkt offered distinguishing features such as rootless containers and tighter security settings.

Most of these container runtimes vanished over time, and some of their code bases were absorbed into the OCI standard. Prior to the OCI, most container runtimes were allowed to implement containerization however they saw fit for their use case; however, the OCI offered a set of defined rules, terminology, and lifecycle requirements. This resulted in runc [31], the first Go-based OCI runtime built from Docker’s work on libcontainer, railcar [32], an Oracle OCI implementation written in Rust, and crun [33], a Redhat C-based implementation. For a while, rkt remained the only non OCI implementation but it eventually was deprecated and some of its advancements were merged into the OCI standard. Only runc and crun are still in use as production low-level container runtimes. Other runtimes available today are intended for specialized environments requiring additional software isolation. By running on a unikernel
or a lightweight virtual machine, such runtimes try to further limit the container’s attack surface [34][35]. Examples of sandboxed runtimes include gVisor [36], nabla [37], kata [38], and AWS’ Firecracker [39].

While container runtimes are the foundation of container execution, other tools, such as those for creating and distributing container images, are required for widespread adoption. Docker calls its container toolkit an engine because it includes everything required to design, deploy, and distribute containers. Other initiatives emphasize responsibility separation and concentrate on various components of the container lifecycle.

As containers grew in popularity, orchestration became important, and integrating several runtimes became time-consuming. To solve this, the Container Runtime Interface (CRI) [40], an abstract runtime model, established itself as the industry standard in container orchestration. Any orchestrator that implements the CRI is capable of supporting numerous container runtimes. CRIs typically delegate container execution to an OCI runtime, though they enable for the orchestration engine to be decoupled from the container runtime. As an addition, the CRI supports the concept of a pod [41], which is a group of containers deployed as a functional unit on the same host.

2.1.3 Container Orchestrators

The migration to microservices highlighted the necessity for autonomous service management in complex applications. When the number of containers in a deployment grows, manually managing the services becomes impossible, especially when they are distributed across multiple hosts. Container orchestration solutions have attracted a lot of attention in recent years because of their ability to automate and schedule tasks like container deployment, administration, load balancing and scaling, among others. In fact, orchestration technologies have become the de facto norm for containerized
production load deployment.

In most cases, orchestration is accomplished through the use of a centralized tool that allows for the management of a container’s scheduling and lifespan. While orchestrators can handle the deployment of numerous services on a single local node, they commonly coordinate deployment in multi-node clusters based on host placement rules and policies [42]. Once a cluster is built, the orchestrator is in charge of scheduling containers across hosts while guaranteeing a certain number of resources (e.g., CPU usage, memory requirements, storage, network access) and overseeing the containers throughout their lifecycle [43].

Over the years, the role of an orchestrator has increased. They now manage multiple aspects of a deployment, such as container networking, scalability, security, availability, and rolling updates [44], all while exposing a distributed application as a single logical unit. Through self-monitoring and self-healing capabilities, as well as rolling deployments and automated rollbacks, operators can reduce the risk of an application’s downtime.

A more extensive explanation of an orchestrator’s services is provided below:

- **Scheduling**: The process of matching a job description or request with an appropriate node in order to fulfill the load’s request. This involves ensuring that there is enough processing power, memory, storage, etc. Scheduling also allows indicating the job’s preference (affinity) for a specific type of node or its aversion (anti-affinity) to running alongside other jobs.

- **Observability** and Disaster Response: Monitors a container’s performance metrics as specified by the operator over its lifetime. Common resource utilization measures include CPU utilization, memory availability, and storage usage. Furthermore, a deeper examination of a service may be performed by giving custom
tests that assess if a service is executing its tasks on a regular basis. This is known as health and availability monitoring, and it works in conjunction with scheduling to automatically reschedule failing or degraded containers as a failover strategy.

• **Scalability**: As a result of an orchestrator’s observability, it is possible to determine whether the number of deployed containers is sufficient to adequately service the demand. This judgment is facilitated by an operator-specified threshold, which, when combined with an orchestrator’s scheduling ability, allows a cluster to scale the number of containers up or down to fit the observed demand for a service.

• **Secret Management**: Orchestrators must interact with nodes in order to launch and configure services; this communication entails the sharing of sensitive configuration information such as usernames, passwords, tokens, etc. Orchestrators make it easier to securely distribute credentials by allowing users to choose encryption schemes and file permissions. When utilized in conjunction with dedicated third-party credential management, safe credential rotation and dissemination is made feasible.

• **Networking**: Because microservices are the result of the deconstruction of a software monolith, the network serves as a conduit for communication between these units. By placing containers in overlay networks, orchestrators ensure that containers that need to “speak” to each other can do so. Orchestrators can segment the network space and effectively arrange containers for communication and isolation by utilizing software-defined networking features. Because containers are ephemeral, the orchestrator may maintain a service discovery mechanism to help containers discover one other even after a host relocation or IP address
change.

- **Sunsetting, Upgrades, and Rollbacks**: Decommissioning and updating services needs careful planning to avoid application downtime. Orchestrators make it simple to gradually upgrade service containers and then decommission sunset containers. If an anomaly or error occurs during the process, orchestrators ensure that any modifications are rolled back to their previous stable state to ensure service continuity. Orchestrators use a similar idea for decommissioning nodes, ensuring that the services hosted in the sunset node are properly drained and that the application remains operational during the move.

Not all of the services listed above are available in all orchestrator offerings. However, the adoption of the Container Runtime Interface (CRI) is standardizing what loads can be orchestrated as well as what orchestration operations can be performed, allowing a common feature set among orchestrators. Common orchestration operations include the scheduling and deployment of containers and **pods**, network placement, and the scaling or draining of services. Other practices gaining traction involve the use of containers for online debugging, A/B testing, and live migration. Popular orchestrators include **Docker Swarm** [45], **Kubernetes** [46], **Apache Mesos** [47], **Marathon** [48], Netflix’s **Titus** [49], **Nomad** [50], among others.

### 2.2 From the Cloud to the Fog and the Edge

With increased cloud computing usage, the drawbacks of the cloud have become more evident. This centralized paradigm is inadequate for applications requiring real-time response, privacy, geographic awareness, and, on occasion, mobile, intermittent connectivity. The resource pools in the cloud are often centralized, highly localized, and physically distant from the end user. As a result, because applications rely on data
flow to and from the network’s core to make choices, they are susceptible to network hop delays [51]. Continuous data transmission to and from the cloud becomes impossible when considering the 79 ZBs of data [52] [53] that 75 billion IoT devices [54] are expected to generate yearly by 2025. At such data volumes, one can expect decreased and unacceptable service quality due to unavoidable increases in network congestion, resource contention, and processing lag [55]. Furthermore, under such cases, bandwidth in the network’s core would be spent inefficiently [56], limiting the cloud’s scalability.

To address these constraints, cloud model extensions such as edge and fog computing have been investigated in an attempt to decentralize the cloud by disaggregating network traffic and processing data close to the end user. In other words, more services and the resources on which they rely are being transported to the network’s rim, closer to the user. While both expansions reduce latency and reliance on the cloud’s core, the major distinction is where the processing takes place. Edge computing attempts to keep data processing on the device from whence it originated, thereby improving privacy [57], whereas fog computing attempts to offload data processing to a surrounding local infrastructure or an IoT gateway rather than the cloud [58]. The extensions are not mutually exclusive and can be used in conjunction based on the needs of the applications being served and the desired level of decentralization.

In contrast to the centralized cloud method, these cloud extensions are typically deployed close to the client device. Because the fog functions as an intermediary data center, it may handle data generated at the network’s edge when devices lack local processing capability or when more resources are necessary. For low latency responses, the interaction between the edge and fog layers is critical. Furthermore, because these extensions are decentralized, they provide greater service resilience even in the event of a cloud outage. Additionally, because of the diverse communication links employed at various layers, services can be successfully delivered even in the presence of a weak
network core or intermittent connections. While promising, it is worth noting that the fog may be resource constrained and may need to rely on the cloud for more powerful computational capabilities and long-term storage. This interplay between layers is known as the Edge-to-Cloud or Cloud-to-Fog continuum [59].

A typical large-scale microservice system may consist of hundreds of microservices [60]. Interactions between containers running on different hosts, which can be asynchronous at times, can become problematic. While cloud, fog, and edge deployments are similar in certain ways, the complexity at the fog and edge is exacerbated due to hardware constraints, heterogeneous ISAs, different peripherals, and many communication interfaces [61]. Resources are further constrained at the edge, and additional requirements such as temperature monitoring and energy conservation measures may result in throttled performance not seen in cloud deployments. Initiatives to deconstruct containers into application logic and environment containers are gaining traction in an effort to alleviate resource pressure in the limited settings of the fog and the edge with the goal of maximizing resource consumption [62][63]. These enhancements reduce container image bloating, make deployment easier in bandwidth-constrained environments, extend application reach across many classes of devices, and transfer some cloud benefits to the fog and the edge, such as rolling updates and quick rollback on failures, while retaining cloud native practices of continuous integration and deployment, albeit at the expense of operational complexity.

2.3 Cloud-Fog Orchestration

2.3.1 Orchestration in the Cloud Layer

In cloud systems, orchestration, or the management of interconnections and interactions between components, has received a lot of attention. While service orchestration
is at the heart of container orchestrators, resource management (e.g., servers, memory, bandwidth) is a major aspect in the cloud and is typically achieved through an orchestrators’ ability to interface with a cloud service provider’s API via vendor-supplied managed orchestration services (e.g., Amazon’s Elastic Kubernetes Service, Azure Kubernetes Service, etc.). As a result, orchestrators have been termed the “OS of the cloud.”

Orchestrators, thanks to their self-monitoring capabilities, can notice fluctuations in available resources and provide relief by requesting additional resources on-demand via their interaction with CSP APIs. Disaster response is another feature of cloud orchestration that limits the harm caused by cloud outages or failing network links to a service region. Orchestrators contribute to disaster recovery strategies implemented by operators by enabling automatic redundancy and selective microservice placement among cluster nodes. With multi-cloud deployments and services deployed to different cloud regions and availability zones nowadays, orchestrators make it simple to coordinate best tactics to reduce service outages, establish redundancy, and divert traffic to healthy and available areas of the cluster.

While orchestrators often manage the utilization of underlying infrastructure, they also cater to application requirements by monitoring deployments throughout their lifecycle. Cloud-native applications are more likely to be monitored and work well with an orchestrator’s capacity to horizontally scale microservices. When the load surpasses a specific threshold, resources such as computing power, memory, and storage can be automatically allocated to a new microservice replica. Similarly, scaling down deployments when resources are no longer required can help to reduce resource waste and cloud expenses. When combined with resource management, this results in a highly dynamic infrastructure that may grow or shrink as needed by the cluster’s applications, as well as a genuine pay-as-you-go cloud experience.
2.3.2 Orchestration in the Fog / Edge Layers

Overhead is associated with orchestrating and accessing resources via the Internet. Quality of service concerns become troublesome for some applications (e.g., real-time gaming, streaming, augmented reality; in general, applications sensitive to network latency) [64], while others have a deeper issue balancing security and usability [65]. As emerging applications grow more aware of their surroundings, mobile, and location-aware, current orchestrators can only meet a portion of their needs [55]. This new generation of applications (i.e., IoT-enabled services) includes, among others, health-monitoring [66], tele-surgery [67], industrial automation, smart cities, Vehicle-to-Vehicle (V2V) coordination, Machine-to-Machine (M2M) communication, among others [68].

The most significant disadvantage of cloud computing is the centralization of resources. Sending vast amounts of information generated by services to the center of the cloud and back under tight service quality requirements simply becomes untenable (i.e., unacceptable rise in network congestion, resource contention, and processing latency) [55]. More of these services (and the resources they rely on) are being brought to the network’s periphery to address these challenges, decentralize resources, and optimize bandwidth use. The use of fog and edge computing enables the orchestration of services based on their physical requirements, scheduling, quality of service requirements, and geographical distribution. This transition is not without difficulties. Because fog and edge computing push services and their data to the network’s edges, difficulties arise from reliability and integrity issues that are common in broadly decentralized networks [59].

While contemporary cloud platforms use orchestrating technologies to manage some aspects of the deployment, the cloud’s centralized approach remains a barrier for some applications. Unnecessary back-and-forth traffic from the network’s edge to its core degrades an application’s perceived quality of service and prevents it from achieving the
physical awareness that some applications require. As a result, while current orchestrators maintain network connectivity, they do not account for geographical awareness, latency, or network mobility. The utilization of resources by cloud orchestrators is another restriction. In contrast to the cloud, where resources are plentiful, the fog and the edge may be resource constrained, and typical orchestration technologies may not be appropriate for these environments due to their resource requirements.

2.3.3 Fog / Edge Orchestration Requirements

In practice, a fog and edge orchestrator must be prepared to meet the multiplicity of physical and virtualized devices that are being deployed, ensuring security and interoperability in an ever-changing environment. All of this must be accomplished while simultaneously being able to reliably scale an application, especially in the face of heavy device churn. It must also evaluate the quality of service required by each application, particularly those that are latency or time sensitive. Consideration of quality of service for each application necessitates two things: data disaggregation (to reduce traffic at the network’s core) [69] and a preference for processing data close to the user requesting it (to reduce processing latency). These two needs must be addressed while handling mobility changeover across many networks and communication links in a smooth manner. Furthermore, orchestration decisions are not static and may need to be modified over time when the environment or device changes. This necessitates the recording, monitoring, and optimization of scheduling techniques based on numerous monitored indicators. Additional criteria, such as energy usage, must be considered at the edge. Battery-powered devices may want to ensure proper application placement based on the energy consumption of the load, the device’s battery levels, or the host’s predicted service uptime. In some situations, orchestration in this domain may even necessitate the orderly draining/migration of a device’s task to prevent a mission from
being disrupted or to ensure service continuity even when devices run out of battery or when scheduled maintenance for a mobile device fleet is approaching. Table 2.1 presents a summary of orchestration requirements at the fog/edge layers.

Table 2.1: Fog/Edge Orchestrator Requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS</td>
<td>Guarantees quality-of-service for applications that require low latency or near real-time performance.</td>
<td>[64] [70]</td>
</tr>
<tr>
<td>Scalability</td>
<td>Allows services to scale up and down while considering device churn and hardware requirements.</td>
<td>[70] [71] [72]</td>
</tr>
<tr>
<td>Mobility</td>
<td>Manages a dynamic network of devices that may enter/leave the network or change location.</td>
<td>[73] [74] [75]</td>
</tr>
<tr>
<td>Security</td>
<td>Provides access control, machine-to-machine confidentiality, trust management, and manages privacy leaks.</td>
<td>[64] [76] [65]</td>
</tr>
<tr>
<td>Interoperability</td>
<td>Provides abstractions to services for executing across different providers and networks.</td>
<td>[64] [70] [77]</td>
</tr>
<tr>
<td>Reliability</td>
<td>Copes with the loss of connectivity brought about by mobility or device malfunction.</td>
<td>[74] [76] [77]</td>
</tr>
<tr>
<td>Telemetry</td>
<td>Monitors infrastructure and service metrics for up-to-date information and smart deployments.</td>
<td>[70] [74] [78] [79]</td>
</tr>
</tbody>
</table>

2.3.4 Evolution of Orchestration Services

Orchestration services have changed throughout time to meet changing needs. Vaquero et al. [74] classified orchestration service evolution into four waves.

The first orchestration wave was concerned with software placement and scheduling in dispersed cloud systems. The orchestrator serves as a resource broker, allocating hosts based on the requirements of the application. Today, orchestrators work on behalf of the human operator, seeking additional resources within the parameters set by the operator for each cloud provider. The orchestrator’s connection with the cloud provider’s APIs makes it simple to scale up and down the infrastructure and deliver the resources needed by the deployed applications.

The second wave extends the first wave by including scheduling for fog and edge devices. This wave considers virtual network functions (e.g., software routing, linking) and allows the orchestrator to use some aspects of software-defined networking (SDN) for the dynamic creation of virtual tenant networks (VTN) and the automatic deployment of applications in such networks due to the “softwareisation” of IT at the fog layer [59][80]. In other words, orchestrators, in conjunction with SDN technology, enable the separation of a network’s logical and physical planes while abstracting the
underlying network’s and its configuration’s complexity. As a result, orchestrators allow network segmentation, mediate service communication, and introduce secure tunneling solutions. Furthermore, the recent arrival of serverless technologies enhances the granularity of orchestration load ranging from virtual machines to microservices to serverless functions [74].

The third wave builds on the second by focusing on hardware programmability, data disaggregation, and data-flow management at the edge. This wave separates computer architecture, resulting in disaggregated computers with pools of remote, virtualized, CPU, RAM, storage, and even network functions. Service composition encompasses all of the processes required to create added-value services from these disaggregated components (e.g., CPU, RAM, storage, NFVs, services, etc.) and has proven difficult in decentralized environments with varying degrees of autonomy between layers, such as the fog/edge and cloud layers.

As a result, while orchestration may occur at each layer, the concept of choreography, or the contract between independent co-operating layers, becomes critical in order to ensure reliability [81] in the event of service failures at any layer of the Edge-to-Cloud continuum or the sudden loss of inter-layer connectivity. Given these challenges, data-flow management is critical at this wave because the main motivator behind architecture disaggregation is to optimize resources and response time for the demands of real-time and mission-critical applications by reducing traffic from the edge to the core of the network while maintaining reliability.

The fourth wave extends the third wave by addressing the complexity and dynamism of infrastructure in ever-changing network conditions. The fog and the edge are highly variable scenarios, in contrast to the cloud’s steady and redundant operational environment. Because of the greater possibility of failed devices and service outages at the fog/edge, resource orchestration may be deemed too computationally
intensive [44][82]. As a result, the orchestrator must use strategies for dynamic, at-runtime reconfiguration while still meeting service demands, as well as some choreography (decentralized collaboration) to incorporate autonomous protocols into each component for reliability [59].

As shown in the literature, orchestration efforts are currently migrating from the second to the third wave, with some efforts focused on solving some fourth wave challenges independently. While technologies can be integrated in pairs, taking into account all of the orchestration requirements presented at the edge, fog, and cloud layers can be challenging, with an exponential increase in complexity due to the combinatorial expansion of load types, platform heterogeneity, environment dynamism, network links, QoS, security and privacy, energy consumption, etc. Furthermore, while these features are designed and created with the cloud in mind, some of the work may be repurposed at the fog/edge layers and even to increase intra-layer coordination. Recently, some work has been done in defining container interfaces such as the Container Network Interface (CNI) [83], the Container Storage Interface (CSI) [84], among others, with the goal of standardizing the mechanism for orchestrators to expose arbitrary storage and networks to their managed workloads and providing avenues for extending the capabilities of today’s orchestrators.
Chapter 3

Reference Architecture

This chapter demonstrates an architecture for orchestrating heterogeneous loads, even when geographically scattered, to test whether certain improvements in cloud technology could be relevant at the edge. Virtual machines (for legacy applications) and containerized applications are supported loads (for newer, cloud-native deployments). It is worth emphasizing that under this architecture, containerized programs are first-class loads, and the majority of the capabilities outlined in the next chapters are geared at microservices. Virtual machine support has been included in order to enable the integration and migration of future technologies to cloud-native standards. The architecture given below seeks to meet the requirements mentioned in Section 2.1.3 and is a reflection of the work done towards Vaquero’s [74] Third and Fourth Waves.

3.1 Design and Implementation

In this design, a datacenter is a collection of client and server nodes that enable the dispersed launch of applications throughout a small geographic area. A datacenter has a local perspective of its resources and provides low-latency services to its nearby
geographic area via cloud service providers’ availability zones. It enables for encrypted service-to-service communication, the storage of keys, secrets, and service configuration, and the ease of continuous integration and deployment. A region is made up of a collection of datacenters. Regions can be designed to be self-governing and to comply with data privacy legislation such as the California Consumer Privacy Act (CCPA) or the General Data Protection Regulation (GDPR) of the European Union. Multi-region federation, on the other hand, is possible for a single, logical picture of global infrastructure. Using this model, we can plan for high availability even in circumstances when a failure causes a datacenter outage. In the event of a region-level failure, federation ensures that loads are routed to neighboring regions that are available.

The design includes three fixed infrastructure components: Consul [85], a network fabric for connection across all clouds; Nomad, a heterogeneous-workload orchestrator capable of scheduling and delivering services at scale; and Vault [86], a secrets management engine. The orchestrator manages additional components that are deployed as infrastructure services. It is also feasible to replace Nomad with Kubernetes while sacrificing support for virtual machine orchestration. Figure 3.1 depicts a region’s reference architecture.

3.1.1 Network Fabric

Consul is in charge of maintaining the state of the datacenter. It answers to service requests, hosts a Key-Value (KV) store for runtime configuration updates, and acts as a secret storage backend for Vault. Consul is distributed as a single binary that can be used to configure a node (a compute instance) as either a server or a client. The RAFT [87] protocol is used by the server nodes to build a leader and maintain the service catalog, service queries, access-control lists, and the KV store in a consistent state. The process of establishing leadership has the inherent overhead of the unorganized com-
Figure 3.1: Reference Architecture
communication taking place between the nodes. Following the establishment of leadership, communication becomes more organized and structured, with the possibility of another leadership election if the node is absent or degraded. Other academics have pointed out the model’s scalability as a shortcoming of this leadership style, as it scales poorly as nodes multiply exponentially [44][82]. As a result, there is ongoing research on how to improve communication by moving away from $O(N^2)$ algorithms. Consul tries to keep this to a minimum by recommending a limited number of Consul servers that can be elected as leaders.

Consul also includes a gossip protocol based on the SWIM [88] protocol to help with cluster membership, failure detection, and event broadcasting. Every member of the cluster is aware of its own health (that is, each member assesses if it is working ideally, experiencing service deterioration, or experiencing CPU exhaustion), but it also listens to other members gossip about its health state (based on their interaction). To raise cluster suspicion, nodes “dogpile” on failing members. Neighboring nodes can either dispute (by admitting that the member is degrading but not failing) or validate the suspicion. Consul servers in separate datacenters can be linked together via WAN-level gossip. RAFT and SWIM work together to enable Consul to create a network mesh that can self-heal and redirect traffic depending on the telemetry received by Consul on each node.

Consul’s features can be used by applications in order to access a service catalog. A Consul agent installed on each node enables these discovery features. By registering services with a Consul agent, other services can find them as well as the healthy nodes where they are situated. Consul-registered services may choose to communicate with one another over a service mesh. The service mesh governs how services communicate with one another by enforcing access control regulations through intentions (i.e., service-to-service permissions), building trust across services, and providing
service-to-service encryption. The usage of a service mesh improves observability at the application layer (L7), allowing for service-level telemetry and a virtualized topology view. Consul accomplishes this through integration with Envoy [89] and the use of sidecar proxies to route service traffic. Envoy is a proxy service that allows traffic to be rerouted for various purposes. While traffic routing within a pod (a standardized abstraction unit comprising many services) is a typical use of Envoy, it may also be used to collect information on API performance, monitor service interactions, and even build security between services by brokering their communication. Consul agents are illustrated in magenta in Figure 3.1 with a label identifying their role.

3.1.2 Workload Orchestrator

Nomad is a versatile orchestrator that manages various types of workloads. Linux containers, native Windows applications, JVM applications, VMs, and other heterogeneous loads are examples. Nomad is distributed as a single binary that may be used to configure a node as either a server or a client. It scales deployment to bare metal and cloud environments. Nomad, like Consul, uses RAFT to create leadership and attain server consensus, and SWIM to manage client membership and node clustering. Nomad-deployed applications are scheduled to run on client nodes and automatically register with Consul’s service catalog, enabling for easy service discovery and health monitoring. Because of its extensible plugin structure, Nomad can accept many types of workloads, devices, and storage (through CSI-compliant plugins), all while delivering runtime metrics for enhanced visibility on the cluster. Nomad servers in separate datacenters connect via WAN-level gossip, and multi-region federation is possible via wider-level gossip.

Nomad evaluates its scheduling decision before deploying an application depending on the availability of resources in the client nodes. If it determines that the resources
at the client nodes have been depleted, *Nomad*’s horizontal cluster autoscaling ensures that an adequate number of client nodes (and resources) are available for deployment. Similarly, *Nomad* honors an application’s service-level agreement (SLA) by scaling up or down according on the throughput required by the application. Other features, such as running containers in a sandbox (*gVisor*, *Kata*, etc.), are supported by switching container engines. By default, *Nomad* makes use of the *Docker* engine but is open to any engine that is compatible under the Container Runtime Interface (CRI). *Nomad* nodes are displayed in green in Figure 3.1 with a label describing their role.

### 3.1.3 Identity and Secret Manager

*Vault* is an engine for managing secrets. A secret is any piece of information (username, password, API Tokens, certificates, etc.) that allows you to get access to a system. Because deployments may occur across clouds, safeguarding the infrastructure necessitates taking into account an ever-expanding (or contracting) network perimeter. As a result, secrets may end up in many, distinct locations. As secrets spread, there is no way to manage who has access to them, and thus no audit trail to determine what illicit actions have occurred. Changing credentials, or secret rotation, becomes difficult as deployments grow larger. *Vault* centralizes secrets at the datacenter level to prevent sprawl and encrypts data in transit and at rest. It grants services access to secrets via Access Control Lists (ACLs) and keeps an audit log. *Vault* provides services with ephemeral tokens to enforce secret rotation and ensures the uniqueness of keys across many instances of a service. This provides for greater traceability of secret leaks and simple revocation without the risk of a datacenter-level service interruption caused by rotating all secrets at once. *Vault* must be implemented in a cluster to ensure high availability. *Vault* delivers highly consistent storage by utilizing *Consul*’s KV store. Plugins can be used to modify the storage backend of *Vault*. *Vault* nodes
are represented in gray in Figure 3.1.

To boost redundancy, the regional reference design can be modified to incorporate more Consul, Vault, and Nomad servers as needed (i.e., $n - 2$ redundancy). Multiple regional architectures can be deployed and federated at scale throughout a geographic area. If global georedundancy is required for Vault, an extra tool, consul-replicate [90], must be used to ensure cross-region KV store replication. In terms of security, mTLS is enabled throughout the datacenter to ensure mutual authentication. Communication is secure when combined with ACLs, rotating credentials, appropriate Linux Security Modules (AppArmor [91], SELinux [92], and Seccomp [93]), and a service mesh. Additional components (such as Web Application Firewalls, load balancers, etc.) can be deployed as infrastructure services by Nomad.

3.2 Software Bill of Materials

This section intends to give a complete inventory of components used to implement the design, as well as a description of the infrastructure services and dependencies, to boost openness, promote research reproducibility, and enable processes such as risk assessment. Additional components used in following chapters are also listed here to offer a complete software bill of materials (SBOM) for all experiments conducted.

3.2.1 Components

3.2.1.1 Infrastructure Components

The infrastructure components are the pieces that form the basis of the architecture provided in this chapter and whose utilization is required for the experiments presented in later chapters. Each node in the reference cluster must install these components,
albeit with varying configurations. Table 3.1 lists the software components, their versions, purposes, and how they were obtained.

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
<th>Purpose</th>
<th>Distribution</th>
<th>Used In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consul</td>
<td>1.8.4</td>
<td>Network Fabric</td>
<td>Binary</td>
<td>All Experiments</td>
</tr>
<tr>
<td>Nomad</td>
<td>0.12.5</td>
<td>Workload Orchestrator</td>
<td>Binary</td>
<td>All Experiments</td>
</tr>
<tr>
<td>Vault</td>
<td>1.5.5</td>
<td>Secret and Identity Manager</td>
<td>Binary</td>
<td>All Experiments</td>
</tr>
<tr>
<td>Docker</td>
<td>20.10</td>
<td>Container Engine</td>
<td>Package Manager</td>
<td>All Experiments</td>
</tr>
<tr>
<td>Ubuntu Linux</td>
<td>20.04 LTS</td>
<td>Operating System</td>
<td>Image</td>
<td>All Experiments</td>
</tr>
<tr>
<td>QEMU</td>
<td>4.2</td>
<td>Virtual Machine Runtime</td>
<td>Package Manager</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>runC</td>
<td>1.0.1</td>
<td>OCI Container Runtime</td>
<td>Source / Binary</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>CRIU</td>
<td>3.15</td>
<td>Checkpoint/Restart Tool</td>
<td>Source / Binary</td>
<td>Chapter 7</td>
</tr>
</tbody>
</table>

Consul, Nomad, and Vault have been described in length in preceding sections. To coordinate heterogeneous jobs, the Nomad Task Driver requires additional components, such as Docker and QEMU [94]. Docker is the container creation and execution engine. Notably, podman [95] has replaced Docker as the engine in Nomad versions 0.12.9 and later. The QEMU task driver offers a generic virtual machine runner that can utilize the KVM [96] kernel module to execute hardware-accelerated virtualization. It mostly relies on qemu-system, which is part of the QEMU distribution. runC can be used to build and manage containers when greater flexibility in the container lifecycle is required, and when coupled with CRIU [97], it enables further orchestration operations (i.e., checkpoint, restore, migration). The aforementioned applications were all installed on Ubuntu 20.04 Long Term Support.

3.2.1.2 End User Components

End user components refers to the applications deployed on the infrastructure by the user. They may include on-demand infrastructure services like Envoy and other user services. Table 3.2 lists the end user components, their versions, purposes, and how
they were obtained.

Table 3.2: End User Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
<th>Purpose</th>
<th>Distribution</th>
<th>Used In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Envoy</td>
<td>1.14.4</td>
<td>Service Proxy</td>
<td>Container</td>
<td>Chapter 4, 5</td>
</tr>
<tr>
<td>qperf</td>
<td>0.4.11</td>
<td>Network Performance Monitoring</td>
<td>Container</td>
<td>Chapter 4, 5</td>
</tr>
<tr>
<td>Uftrace</td>
<td>0.9.4</td>
<td>Function Tracer</td>
<td>Source</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Gcc</td>
<td>11.2.0</td>
<td>Compiler Collection</td>
<td>Source / Binary</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Util-linux</td>
<td>2.36.1</td>
<td>Linux Utilities Collection</td>
<td>Package Manager</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Ubuntu Linux</td>
<td>21.10</td>
<td>Container Base Image</td>
<td>Container</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Gdb</td>
<td>9.2</td>
<td>Debugger</td>
<td>Package Manager</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Gdbserver</td>
<td>9.2</td>
<td>Remote Debugging Server</td>
<td>Container</td>
<td>Chapter 7</td>
</tr>
</tbody>
</table>

3.2.2 Services

3.2.2.1 Infrastructure Services

A port exposes the functionality of an infrastructure component as an infrastructure service. Typically, these services are expressed through REST APIs, gRPC calls, and other mechanisms. These services provide the cluster membership and leadership algorithms, as well as the network architecture that enables communication between services. Table 3.3 enumerates the components, the supplied services, and the port information.

3.2.2.2 End User Services

End user services refer to the exposed functionality of user deployed applications via a port. They may include infrastructure services on demand, such as Envoy sidecar proxy services, and other user-defined services. Nomad assigns ephemeral ports to user services. Table 3.4 enumerates the components, the supplied services, and the port information.
Table 3.3: Infrastructure Services

<table>
<thead>
<tr>
<th>Component</th>
<th>Service</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consul</td>
<td>DNS</td>
<td>8600</td>
</tr>
<tr>
<td></td>
<td>HTTP API</td>
<td>8500</td>
</tr>
<tr>
<td></td>
<td>HTTPS API</td>
<td>8501</td>
</tr>
<tr>
<td></td>
<td>gRPC API</td>
<td>8502</td>
</tr>
<tr>
<td></td>
<td>LAN Serf</td>
<td>8301</td>
</tr>
<tr>
<td></td>
<td>WAN Serf</td>
<td>8302</td>
</tr>
<tr>
<td></td>
<td>Server</td>
<td>8300</td>
</tr>
<tr>
<td>Nomad</td>
<td>HTTP API</td>
<td>4646</td>
</tr>
<tr>
<td></td>
<td>RPC</td>
<td>4647</td>
</tr>
<tr>
<td></td>
<td>WAN Serf</td>
<td>4648</td>
</tr>
<tr>
<td></td>
<td>Ephemeral Task Port Range</td>
<td>20000 – 32000</td>
</tr>
<tr>
<td>Vault</td>
<td>Server</td>
<td>8200</td>
</tr>
<tr>
<td></td>
<td>Server-To-Server Communication</td>
<td>8201</td>
</tr>
<tr>
<td>Docker</td>
<td>Daemon</td>
<td>2375</td>
</tr>
<tr>
<td></td>
<td>Secure Daemon</td>
<td>2376</td>
</tr>
</tbody>
</table>

Table 3.4: End User Services

<table>
<thead>
<tr>
<th>Component</th>
<th>Service</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Envoy</td>
<td>Consul’s Sidecar Proxy</td>
<td>21000 – 21555</td>
</tr>
<tr>
<td>qperf</td>
<td>Network Performance Monitoring</td>
<td>Ephemeral</td>
</tr>
<tr>
<td>Gdbserver</td>
<td>Remote Debugging Server</td>
<td>Ephemeral</td>
</tr>
</tbody>
</table>
Chapter 4

Impact on Network Resources

4.1 Data Collection

When deciding to deploy an application, operators may employ a variety of infrastructure service levels. This section discusses the data gathering efforts conducted in order to compare the performance impact of differing infrastructure service levels on the network resources of the target host. For the purpose of establishing a baseline, data was collected on traditional Docker deployments, which consist of using the Docker engine directly without any additional orchestration. Data was also collected in orchestrated deployments with varied degrees of infrastructure services. During multiple network tests, data on network performance and CPU utilization was obtained. qperf [98] was used to measure TCP latency and bandwidth between two applications deployed in containers. This requires a node operating in qperf server mode and another node operating in qperf client mode.

After deploying a server and client container on separate hosts inside the same network segment, three scenarios based on the features supplied to the operator were examined:
• **Docker (baseline):** The application is deployed utilizing the **Docker** engine. The operator gets faster startup times than virtual machines, and resource usage is enhanced. This scenario will be referred to as **docker** in the discussion.

• **Orchestrator:** The application is deployed using an orchestrator and registered in the service catalog. The registered service supplied by **qperf**’s server container is now available for discovery. It acquires a service address and is continuously monitored through a health and availability check. The client container uses service discovery to communicate with **qperf**’s server container. In this scenario, the operator enjoys simple management of the lifetime of applications across several clouds and at scale. This scenario applies to all Cloud-to-Edge layers, given that software-defined networks can be implemented at the edge and in the fog. These tests utilize **Nomad**, an open-source orchestrator. As such, this scenario is referred to as **nomad**.

• **Service Mesh:** An orchestrator is used to deploy the application and facilitates the connectivity between services. Consequently, the operator enjoys encryption on all inter-service connections, a service mesh arrangement, and Layer 7 observability. In the tests, **Consul** is used as the service mesh provider and its integration with **Envoy** is leveraged to broker and secure service-to-service communications through a sidecar proxy. This scenario is known as the **service mesh**.

The tests were carried out in an Amazon Web Services Virtual Private Cloud (VPC). Each instance ran **Ubuntu** Server 20.04 and the required software stack (**Consul** 1.8.4, **Nomad** 0.12.5, **Vault** 1.5.5) to implement the reference architecture outlined in Chapter 3. In each scenario, the network testing tool **qperf** 0.4.11 was distributed as a container.

Each obtained measurement is comprised of multiple of TCP latency and bandwidth
metrics. There were 120 measurements recorded for each scenario, for a total of 360 measurements. The metrics gathered are summarized in Table 4.1. Latency metrics are prefixed with `tcp_lat`, whereas bandwidth metrics are prefixed with `tcp_bw`.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_lat_latency</td>
<td>us</td>
<td>One-way TCP latency between two nodes.</td>
</tr>
<tr>
<td>tcp_lat_msg_rate</td>
<td>K/sec</td>
<td>Throughput in thousands of messages per second. Each message is by default 1 byte long.</td>
</tr>
<tr>
<td>tcp_lat_loc_cpus_used</td>
<td>%</td>
<td>CPU usage due to sending data.</td>
</tr>
<tr>
<td>tcp_lat_rem_cpus_used</td>
<td>%</td>
<td>CPU usage due to receiving data.</td>
</tr>
<tr>
<td>tcp_bw_bw</td>
<td>MB/sec</td>
<td>TCP streaming bandwidth</td>
</tr>
<tr>
<td>tcp_bw_msg_rate</td>
<td>K/sec</td>
<td>Throughput in thousands of messages per second. Each message is by default 64 KiB.</td>
</tr>
<tr>
<td>tcp_bw_send_cost</td>
<td>sec/GB</td>
<td>Time that it takes to send a GB.</td>
</tr>
<tr>
<td>tcp_bw_recv_cost</td>
<td>sec/GB</td>
<td>Time that it takes to receive a GB.</td>
</tr>
<tr>
<td>tcp_bw_recv_bytes</td>
<td>MB</td>
<td>Total MBs transferred.</td>
</tr>
<tr>
<td>tcp_bw_send_cpus_used</td>
<td>%</td>
<td>CPU usage due to sending data.</td>
</tr>
<tr>
<td>tcp_bw_recv_cpus_used</td>
<td>%</td>
<td>CPU usage due to receiving data.</td>
</tr>
</tbody>
</table>

### 4.2 Comparison Methodology

Hypothesis testing was employed to understand whether the use of varying degrees of orchestration services negatively affects each measured metric when compared to the baseline group. The services offered were classified in three groups: `docker (n = 120, baseline)`, `nomad (n = 120)`, and `service mesh (n = 120)`. The metrics showed a significant departure from normality ($p \leq 0.001$) as assessed by Shapiro-Wilk’s test. This was validated further by a visual examination of Q-Q plots. Although not ideal
from a statistical standpoint, the outliers were not removed as there was no reason to reject them or declare them invalid since they all fell within the possible range of values for each metric. Levene’s test revealed that homogeneity of variances was violated for all metrics \( p \leq 0.05 \) except tcp_bw_send_cost and tcp_bw_send_cpus_used.

A one-way Welch ANOVA was conducted to determine if each metric differed substantially between groups with differing degrees of orchestration services. The difference in group means was expressed through a null \( H_0 \) (all group population means are equal) and alternative \( H_A \) hypothesis (the means of the groups are not equal). This test was appropriate due to its robustness against deviation of normality, particularly when sample sizes are large and equal. For the cases where there is evidence that the null hypothesis can be rejected (i.e., there is a statistically significant difference between the means of the groups), a Games-Howell post-hoc test was undertaken to discover where the differences reside. Data is presented as mean ± standard deviation. In all cases, and for all tests, results are reported at the 95% confidence level.

### 4.3 Results and Discussion

This section presents the analysis and results obtained from the collected data. Refer to Table 6.4 for additional information on the statistical significance of the tests, the difference between the means of the orchestration service levels, and their confidence intervals. Figure 4.1 shows the overhead across different orchestration service levels.

Latency and throughput, as measured by qperf’s TCP latency test, were statistically significantly different between the groups. The docker group (79.19us±2.09us) had the lowest latency, followed by the nomad group (97.57us±3.74us) and the service mesh group (179.38us±3.99us). The increase in latency from the docker to the nomad group (18.38us, 95% CI (8.26us to 28.50us)) was statistically significant \( p < 0.001 \), as
Figure 4.1: Network Impact from Orchestration Services
Table 4.2: Network Metrics Comparison Results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Descriptives</th>
<th>ANOVA</th>
<th>Games-Howell Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Level Mean Std. Dev</td>
<td>Stat.* df1 df2 Sig.</td>
</tr>
<tr>
<td>tcp_lat_latency</td>
<td>docker</td>
<td>79.19 22.92</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>97.57 40.96</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>179.38 43.79</td>
<td>nomad</td>
</tr>
<tr>
<td>tcp_lat_msg_rate</td>
<td>docker</td>
<td>13.30 2.63</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>11.57 3.36</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>5.79 0.89</td>
<td>nomad</td>
</tr>
<tr>
<td>tcp_bw_bw</td>
<td>docker</td>
<td>472.89 144.75</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>446.23 148.71</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>278.04 113.02</td>
<td>nomad</td>
</tr>
<tr>
<td>tcp_bw_msg_rate</td>
<td>docker</td>
<td>7.15 2.20</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>6.76 2.26</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>4.33 1.73</td>
<td>nomad</td>
</tr>
<tr>
<td>tcp_bw_send_cost</td>
<td>docker</td>
<td>1.48 1.42</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>1.73 1.59</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>3.16 1.62</td>
<td>nomad</td>
</tr>
<tr>
<td>tcp_bw_recv_cost</td>
<td>docker</td>
<td>1.80 0.19</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>1.86 0.37</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>4.00 0.76</td>
<td>nomad</td>
</tr>
<tr>
<td>tcp_bw_recv_bytes</td>
<td>docker</td>
<td>928.15 307.02</td>
<td>nomad docker</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>878.14 307.35</td>
<td>service mesh docker</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>554.57 227.35</td>
<td>nomad</td>
</tr>
</tbody>
</table>

* Asymptotically F distributed.
* The mean difference is significant at the 0.05 level.

was the rise from the nomad to the service mesh group (81.82us, 95% CI (68.91us to 94.73us), p < 0.001). As more orchestration services were employed, throughput, measured in thousands of 1-byte messages per second, declined. Values decreased from the docker group (13.30 K/sec±0.24K/sec) to the nomad (11.57 K/sec±0.31 K/sec), and service mesh group (5.79 K/sec±0.08 K/sec). The decrease from the docker group to the nomad group (-1.73 K/sec, 95% CI (-2.64 K/sec to -0.81 K/sec)) was statistically significant (p < 0.001), as was the decrease from the nomad group to the service mesh group (-5.78 K/sec, 95% CI (-6.53 K/sec to -5.03 K/sec), p < 0.001).

Bandwidth and throughput, as measured by qperf’s TCP bandwidth test, were statistically significantly different between the groups. Bandwidth decreased from the docker group (472.89 MB/sec ± 13.21 MB/sec) to the nomad (446.23 MB/sec ±13.57
MB/sec), and service mesh group 278.04 MB/sec ±10.32 MB/sec), in that order. Only the drop from the nomad to service mesh group (−168.19 MB/sec, 95% CI (−208.43 MB/sec to −127.96 MB/sec)) was statistically significant (p < 0.001). As additional orchestration services were employed, throughput, measured in thousands of 64 KiB messages per second, declined. Values decreased from the docker group 7.15 K/sec±0.20 K/sec) to the nomad (6.75 K/sec±0.21 K/sec), and service mesh (4.33 K/sec±0.16 K/sec) group, in that order. Only the drop from the nomad to service mesh group (-2.43 K/sec, 95% CI (-3.04 K/sec to -1.81 K/sec)) was statistically significant (p < 0.001).

qperf also reports other derived metrics, such as the time required to send and receive a gigabyte of data and the total number of bytes delivered and received per session. This study does not include the cost of receiving a gigabyte or the total number of bytes sent. The time to send a gigabyte increased from the docker (1.48s±0.13s) to the nomad (1.73s±0.15s), and service mesh (3.16s±0.15s) group, in that order. Only the rise from the nomad to the service mesh group (1.43s, 95% CI (0.94s to 1.92s)) was statistically significant (p < 0.001).

The increase in latency was anticipated in both the nomad and service mesh situations. This is due to the fact that the operation of the orchestrator raises network utilization throughout the entire datacenter in order to obtain consensus, retain leadership, and report health checks. Even under ideal network conditions, employing the orchestrator results in latency increases of approximately 23% relative to the docker baseline. Compared to the baseline, the service mesh scenario results in an increase of 226%. Consul provides the capacity to deploy a service mesh via sidecar proxies or via direct API integration. In the service mesh scenario, the sidecar proxy approach was evaluated because no modifications to qperf’s source code were necessary. The native integration is restricted to Go applications via the officially supported API, and the original source code must be modified to exploit the existing integration between
Consul and Envoy. Given that Go is a popular language for web services, adoption may be simple for contemporary services, but adoption for legacy services may require significant application changes, such as a rewrite or wrapper driver. Alternative solutions, such as those given by the Consul Community, may be accessible and may enable the use of Envoy proxies with third-party languages. Given the Consul API and its REST interface, it is possible for anyone to build a module or library that interfaces with it to configure the connections, albeit at the cost of ongoing software maintenance with each Consul release.

Consul asserts that the use of their service mesh via a sidecar proxy adds microseconds of latency and suggests using native integration for latency-sensitive services. This study demonstrates that, under optimal conditions, the increase is measured in microseconds. Note that the presented data refers to service-to-service latency within the datacenter. As a reference, consider that, as of 2020, the roundtrip time for a packet traveling from California to the Netherlands is around 150ms; a packet traveling from Orlando, FL to Miami, FL experiences a roundtrip time of 10ms; and a packet traveling within a datacenter should experience a roundtrip time of roughly 500us [99]. In the service mesh scenario, the one-way latency within the datacenter was measured to be 179.38us±3.99us within the boundaries of the Virtual Private Cloud.

In terms of bandwidth, reported in MB/s, there was a decrease as more orchestration services were employed. This was anticipated due to the constant gossip in the network. Using Nomad decreased the available bandwidth by nearly 6% compared to the baseline, while the service mesh scenario decreased it by 70%. Similarly, the throughput, which was reported in thousands of messages per second, decreased as the number of orchestration services increased. For the latency and bandwidth tests, throughput was evaluated differently. The message size for the latency test was 1 byte. On the other hand, the message size was set to 64 KiB for the bandwidth test. Focusing
on the 64 KiB message size test, a reduction of over 6% may be noticed for the nomad scenario compared to the baseline. In the service mesh scenario, performance degrades by around 61%. These measures are essential for businesses like Netflix, whose operational performance depends on their capacity to maximize their data transmission rate. As a result of the degraded bandwidth encountered in the service mesh scenario, the expenses to maintain the bandwidth’s baseline level would inevitably increase as more compute power is required. As a result of the network overhead, it may not be able to service as many end customers as desired.

In comparison to the docker baseline, latency tends to grow when more orchestration features are employed. Even though the increase in latency is statistically significant, it is still within acceptable limits as it remains in the microsecond range. The use of a mutual TLS-encrypted service mesh results in a large reduction in bandwidth. This degradation may necessitate additional compute power consumption to achieve baseline values. In turn, this could complicate data aggregation and processing at the edge.

### 4.4 Related Works

Peng et al. [77] discussed network dimensioning, security, and scheduling based on the capabilities and reliability of the nodes comprising their Cloud-Fog models. They considered heterogeneous network links, multiple network interfaces, and varied configurations. In their implementation, they studied a Fog Network Size – Latency tradeoff that allowed them to optimize the network dimensions based on node proximity and resources provided. Other research has focused on the ability to monitor the quality of a connection to implement flexible routing strategies, increase visibility, and guarantee QoS. While the focus on infrastructure telemetry (remote monitoring) can be applied
to several orchestration areas, it has been recently studied to focus on the synchronization of cloud environments and mesh networks deployment on top of popular container orchestrators.

Focusing on orchestrators, Stamper et. al [100] examined the relationship between multiple service deployments, and the scheduling latency required to guarantee a placement that was optimal in terms of quality of service and mission survivability at the edge. In their experiments, they made use of a network-aware scheduler for the deployment of a mesh network on top of Kubernetes that could be used to broker service connections. The result was that the deployment and scheduling aspect of the task was substantially slower than the default Kubernetes scheduler, sometimes taking in excess of 300 seconds for a single task placement. However, this overhead came with significant improvements in quality of service and service survivability as it was able to account for fluctuations in bandwidth and latency and place loads on the proper hosts based on the minimum required guarantees.

The preceding sections demonstrated that a service mesh can be applicable at the edge with minimal overhead. It seems, however, that other researchers have found overhead stemming from the service placement decision as orchestrators are currently unaware of their network environments, as well as the expensive $O(N^2)$ computations required to calculate placement decisions.

Pusztai et al. [101] made similar observations by noting that orchestrators presume that microservices are interconnected via speedy communication links. While this is true in the cloud, it is not guaranteed at the fog and edge layers, where device heterogeneity and network delays are widespread. While they did not focus on mesh networks, they developed a scheduling plugin to optimize placement for asynchronous microservices.

Marchese and Tomarchio [102] concurred on the orchestrators’ disregard for Fog-
Edge deployments. They investigated the applicability of the Kubernetes’ default scheduler at the fog-edge layers while taking the network conditions into account. Because Kubernetes does not prioritize network conditions, they propose a network-aware scheduler plugin that optimizes scheduling decisions based on the state of a network and an application’s perceived QoS. Their plugin is one of the first to allow pods to be evicted from nodes if a more suitable node can be found. The cost of eviction and re-placement is not discussed although they do note the need to research descheduling techniques further.

While consideration for an environment’s network performance is important, most cloud clusters are designed to be distributed across locations. Most orchestrators offer node placement techniques by adding taints and tolerations but do not offer any meaningful geographical awareness to their applications. As a result, Rossi et al. [103] focused on the geo-distributed aspect of the fog and edge layers and proposed some extensions to Kubernetes in order to decentralize its operation and improve its self-adapting capabilities via monitoring the environment and network conditions.

Similarly, to consider the geographically distributed aspect of today’s clusters, Toka [104] proposed a topology clustering mechanism to consider an application’s latency requirements when performing scheduling on geographically distributed systems.
Chapter 5

Impact on Compute Resources

As part of the tests conducted in Chapter 4, qperf reported on the CPU consumption during data transfer. Overhead in computing resource utilization is shown by CPU usage over the baseline. This chapter quantifies such overhead while using various degrees of infrastructure services.

The qperf findings were considered because the network is the conduit through which containers expose their services and connect with one another. As a result, because services are spawned and destroyed based on thresholds, often CPU utilization, a network transfer test is a more relevant test for measuring CPU consumption than a completely idle scenario. In a dynamic environment, when services are idle, the orchestrator has the option of decommissioning a container, task, or pod to make place for others or just to reclaim resources. As a result of cost-cutting efforts, idle services are becoming increasingly rare. The tests discussed in this chapter concentrate on CPU utilization when executing a full bandwidth test, specifically on the sender’s side.
5.1 Data Collection

For information on the experiment setup and the various tiers of infrastructure features investigated, see Section 4.1. For convenience, the metrics related to CPU utilization are shown on Table 5.1.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_lat_loc_cpus_used</td>
<td>%</td>
<td>CPU usage due to sending data on the latency test</td>
</tr>
<tr>
<td>tcp_lat_rem_cpus_used</td>
<td>%</td>
<td>CPU usage due to receiving data on the latency test</td>
</tr>
<tr>
<td>tcp_bw_send_cpus_used</td>
<td>%</td>
<td>CPU usage due to sending data on the bandwidth test</td>
</tr>
<tr>
<td>tcp_bw_recv_cpus_used</td>
<td>%</td>
<td>CPU usage due to receiving data on the bandwidth test</td>
</tr>
</tbody>
</table>

5.2 Comparison Methodology

Details on the comparative process used to assess the metrics gathered can be found in Section 4.2.

5.3 Results and Discussion

This study disregards the CPU usage on the receiving end. As such, some metrics such as tcp_lat_rem_cpus_used and tcp_bw_recv_cpus_used are ignored in this discussion. It instead focuses on CPU utilization at the sender’s end. However, all the metrics and the comparisons performed can be seen in Table 5.2. Figure 5.1 shows the CPU utilization across service levels.

The sender’s CPU consumption increased from the docker (52.64% ± 2.30%) group to the nomad (57.75% ± 2.17%), and then to the service mesh (80.98% ± 2.33%) group,
## Table 5.2: Compute Metrics Comparison Results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Descriptives ANOVA</th>
<th>Games-Howell Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Level</td>
<td>Mean</td>
</tr>
<tr>
<td>tcp_lat_loc_cpus_used (%)</td>
<td>docker</td>
<td>28.06</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>29.07</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>35.95</td>
</tr>
<tr>
<td>tcp_lat_rem_cpus_used (%)</td>
<td>docker</td>
<td>28.18</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>30.72</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>37.31</td>
</tr>
<tr>
<td>tcp_bw_send_cpus_used (%)</td>
<td>docker</td>
<td>52.64</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>57.75</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>80.98</td>
</tr>
<tr>
<td>tcp_bw_recv_cpus_used (%)</td>
<td>docker</td>
<td>83.50</td>
</tr>
<tr>
<td></td>
<td>nomad</td>
<td>81.34</td>
</tr>
<tr>
<td></td>
<td>service mesh</td>
<td>104.63</td>
</tr>
</tbody>
</table>

* Asymptotically F distributed.
* The mean difference is significant at the 0.05 level.

In that order. Only the increase in CPU consumption from the nomad to the service mesh group was statistically significant ($p < 0.001$). Similar results can be seen in the latency-derived metrics.

When compared to the baseline, CPU use increased by 10% in the Nomad scenario. When employing the service mesh, the overhead climbed to 54%. We suggest treating this information with caution. Non-idle time is recorded as CPU utilization. This still includes the time spent waiting on the memory subsystem. In other words, the stated

![CPU Utilization](image)

**Figure 5.1: CPU Utilization Across Service Levels**
CPU use includes the time the system spends waiting on memory results and is thus inflated. Instructions per cycle (IPC) would be a more precise metric. If IPC < 1.0, the system is most likely stalled due to memory access rather than CPU constrained [105]. Tuning based on Performance Monitoring Counters (PMCs) could result in increased throughput while requiring less computational power. Such strategies have the potential to mitigate the degradation encountered by the service mesh scenario. In reality, certain systems that consider this have improved resource use. Netflix, for example, examined these optimizations for their streaming service and reported a 13% reduction in the number of containers required to provide peak traffic while meeting latency requirements [106].

The overhead caused by orchestration features can be large, which is important to consider because the processing power used by infrastructure services has a detrimental impact on services by consuming resources that could otherwise be utilized to support requests. As a result, the degradation of other resources, such as network performance, discussed in Chapter 4, could be mitigated by increasing the infrastructure’s available processing power through the addition of nodes. Some optimization solutions have been investigated to alleviate the additional compute power requirement by utilizing the orchestrator’s scheduler to make load allocations based on memory subsystems. If necessary, the tested orchestrator’s extensibility enables for the implementation of these optimizations via plugins.

Memory use is another component that is frequently studied with processing power consumption. This study did not include memory utilization reports because, while it varies with application size, it is also regulated at deployment by setting soft and hard limits on resource consumption. It is worth noting that the container is not the only source of memory consumption; rather, the container engine selected might influence the node’s overall memory consumption. runC, the current standard engine,
is reported to utilize about 15 MB of RAM while running. Other solutions, such as crun, consume roughly 3.7 MB and are being investigated for edge deployments [107]. The container engine in this experiment was Docker with a runC backend and is worth noting that Docker includes other components that also impact memory utilization. Overall, containerized processes are estimated to have a 2-4% overhead on CPU usage [108][109][110].

5.4 Related Works

Avino et al. [111] sought to characterize the CPU overhead in mobile edge deployments. It was found that Docker is a lightweight solution that incurs in variable overhead depending on the number of containers running in the background and foreground, network management, and exposed ports. From a pure computational perspective, for services that do not make use of network functions, the overhead is negligible and constant. While CPU utilization overhead of a container drawing solely CPU resources is considered to be negligible, it is widely accepted to hover between 2-4% [108][109][110].

Those findings were ratified by Preeth et al. [112] when evaluating resource availability to unconstrained containers. Containers had full access to the host resources when control groups are not in place, hence allowing them to consume as much CPU as needed. They also noted that container’s weak spot was network utilization, with significantly degraded performance over a baremetal approach.

This supports the approach taken in this chapter where the CPU overhead was measured when networking features were used. The network typically serves as the conduit for service to service communication and is paramount when implementing a microservice-based environment. It is worth noting that the aforementioned researchers
made observations based only on the behavior of the container runtime/engine. This chapter presented several levels of orchestration features, including the implementation of a service mesh. A service mesh requires a sidecar container to be launched within the network namespace of the original container and implicates data transfer between the two containers as an intermediate step before communication is established between distinct services. In other words, the service mesh container acts as a proxy between the two services, hence lowering the bandwidth and worsening the latency perceived by the application.

Ganguli et al. [113] attempted to measure the impact of a service mesh in an orchestrated environment. They noted throughput degradation of around 50–75% when using Istio and Envoy and investigate to find that the root cause is the impact on CPU utilization. A higher CPU utilization lowers the cluster’s ability to respond to requests by consuming CPU cycles that could otherwise be used to serve requests. This is due to the forwarding mechanisms being CPU intensive. The use of `iptables` in the CNI plugins accounts for half of the CPU overhead with the remaining stemming from the sidecar proxies and communication encryption. Their results are in agreement with the results showcased in this chapter and highlight the importance of optimizing rule matching for dynamic systems at the fog and edge layers.
Chapter 6

Impact on the Perception of Time

With the introduction of containerization, applications are subjected to a dynamic operating model that demands them to be restarted, updated, migrated, and even rolled back to previously working versions on a regular basis. Because there are no hard assurances on host placement or scheduling, an application may be relaunched on a different host or at a later time, causing it to lose its sense of time and refuse service due to incongruent states. In fog and edge environments, where deployments are dynamic and may need to be rescheduled regularly, orchestrating a service’s time perception is critical to ensuring service survival. Unfortunately, until now, process time has been linked to a server. It is now possible to associate time with a service thanks to the recent release of the Linux time namespace. Processes with a time namespace can obtain their own timeline, regardless of the host. The most common container engines presently lack support for the time namespace.

Virtual time is defined as a technique for structuring distributed systems to use a temporal coordinate system that is more computationally meaningful than real time since it supports synchronization and concurrency control procedures [114]. Virtual time does not have to be linked to real time; in fact, it is conceivable to have multiple
local virtual clocks that are only weakly synchronized yet are all progressing toward higher virtual times. They may occasionally jump backwards, but the overall trend is for them to be monotonic. Multiprocessing systems and distributed systems have certain commonalities in terms of virtual time [115], and each process and action may be characterized by the temporal coordinates \((x, t)\), where \(x\) is the process and \(t\) is the instant in its virtual time. All processes share the host’s chronology in the absence of virtual time.

To some extent, all Linux processes are already virtualized. After all, for scheduling purposes [116], the Linux Completely Fair Scheduler (CFS) provides the concept of virtual runtime [117] (i.e., a measure of a thread’s runtime). CFS’ virtual time is not meant to expand beyond the limits of a host because its goal is to increase the performance of interactive processes in desktop systems [118]. Microservices, on the other hand, are now meant to be loosely connected and scalable, with the ability to utilize disposable resources in dynamically provided settings. As a result, applications are often restarted, upgraded [119], migrated [120], and even rolled back to previously known versions [121]. Processes must be dynamically scheduled in this modern operating model, and there are no guarantees that a process, once paused, will resume in the same host or within a defined time range. When migrating, a new process is spawned on the receiving host, and CFS’s virtual time concept is insufficient to prevent the process from feeling it has advanced or regressed in time. Due to an incongruent view of system time, this may result in service refusal [122]. This is due to the fact that time is associated with a server rather than a service.

Virtualizing time for each process is not a novel concept. One of the first attempts to include some code into the Linux kernel allowed a process to maintain its own view of time by storing offsets added to the current system time [123]. The only use case at the time was to speed up the \texttt{gettimeofday()} system call by storing a copy of time
in userspace. However, the vDSO [124], a kernel read-only shared library mapped into all userspace applications, became the norm for faster system calls, and this attempt at virtual time was not implemented in the kernel.

With an emphasis on OS-level virtualization techniques that had grown popular as a result of containers, Linux maintainers developed support for a time virtualization approach, the time namespace [125]. This allowed an application to keep its own time by leveraging the monotonic and boot time clocks. Unfortunately, using this namespace in microservices and outside of research environments is not feasible because it is not yet included in the Open Container Initiative (OCI) specification [126] and so is not supported by mainstream container engines [127] [128] and orchestrators. Furthermore, due to implementation details, this namespace can only virtualize time reporting but not the rate of time passage [129], restricting virtual time’s full potential.

Because of the recent introduction of time namespaces, container engine support is non-existent, and little is known about the impact of virtual time on application performance. This document describes a workflow for manually generating containers that take advantage of this new capability, as well as an evaluation of the overhead of time virtualization in Linux containers. The performance impact research described in this chapter takes into account a wide range of time-related system calls and their vDSO counterparts. The study focuses on the overhead caused by namespace usage. System calls relating to sleep, time reporting, and process timers are among those included. In particular, the overhead associated with time virtualization is evaluated in two scenarios: (1) when it is utilized independently of other container-enabling primitives, and (2) when all of the normal characteristics of a production-grade container are applied. These findings are contrasted to a no-time virtualization baseline in which the host supplies unaltered time information.
6.1 Additional Background

6.1.1 Hardware Clock Sources

In current hardware, a variety of timekeeping devices (e.g., TSC, PIT, RTC, ACPI, HPET, and LAPIC) are available to act as clock sources for a system [130]. A clock source’s goal is to provide a chronology for the system, indicating where you are in time. In general, a system requires a reliable clock source that never goes backward, never stops ticking, avoids abrupt time jumps, has a decent resolution or frequency, and is easily available to userspace code. In reality, this means that, independent of the underlying hardware design, a clock source must be monotonic (i.e., continually rising), have high precision and a constant frequency, not move suddenly in time, and provide atomic access. Because the remaining timers do not fit the criterion or are not always accessible [131], the TSC (Time Stamp Counter), an auto-incremented CPU register, is the recommended clock source on modern x86 hardware. This is not to imply that TSC is without problems; rather, as compared to the alternatives, the weaknesses highlighted by TSC are an acceptable compromise.

6.1.2 Linux Timekeeping

In Linux, time is kept using the idea of clocks, which are abstractions on the previously stated hardware counters. Applications can interface with Linux clocks for timekeeping rather than using clock sources directly (e.g., via the rdtsc command for the TSC) [132]. This enables for faster access than a clock source might provide. For example, the Completely Fair Scheduler (CFS) frequently requires timing information, and while the clock sources may be quite accurate, access to them is not fast enough. In such circumstances, trading accuracy for speed is essential, which the CFS achieves using the sched_clock() kernel function [133]. Other userspace cases, such as high-
performance databases and financial applications, may experience excessive delay while accessing the clock source, forcing the compromise of using Linux clocks. Linux has various clock variants and has added several alternatives over time to meet the needs of programs and speed up information retrieval [134]. Table 6.1 displays a list of Linux clocks and their features.

Table 6.1: Available Clocks in Linux as of Kernel 5.13

<table>
<thead>
<tr>
<th>Clock Function</th>
<th>Function</th>
<th>Introduced</th>
<th>Adjustments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK_REALTIME</td>
<td>Reports the number of seconds since the Epoch.</td>
<td>Linux 2.6</td>
<td>Yes</td>
</tr>
<tr>
<td>CLOCK_MONOTONIC</td>
<td>Reports number of seconds since kernel booted.</td>
<td>Linux 2.6</td>
<td>Partially³</td>
</tr>
<tr>
<td>CLOCK_PROCESS_CPUTIME_ID</td>
<td>Measures CPU time consumed by all threads of a process.</td>
<td>Linux 2.6.12</td>
<td>No</td>
</tr>
<tr>
<td>CLOCK_THREAD_CPUTIME_ID</td>
<td>Measures CPU time consumed by a thread.</td>
<td>Linux 2.6.12</td>
<td>No</td>
</tr>
<tr>
<td>CLOCK_MONOTONIC_RAW</td>
<td>Like CLOCK_MONOTONIC but without adjustments.</td>
<td>Linux 2.6.28</td>
<td>No</td>
</tr>
<tr>
<td>CLOCK_REALTIME_COARSE</td>
<td>Faster but less precise than CLOCK_REALTIME.</td>
<td>Linux 2.6.32</td>
<td>Yes</td>
</tr>
<tr>
<td>CLOCK_MONOTONIC_COARSE</td>
<td>Like CLOCK_MONOTONIC but less precise.</td>
<td>Linux 2.6.32</td>
<td>Partially³</td>
</tr>
<tr>
<td>CLOCK_BOOTTIME</td>
<td>Like CLOCK_MONOTONIC but counts suspended time.</td>
<td>Linux 2.6.39</td>
<td>Partially³</td>
</tr>
<tr>
<td>CLOCK_REALTIME_ALARM</td>
<td>Interval timer on CLOCK_REALTIME with system waking.</td>
<td>Linux 3.0</td>
<td>Yes</td>
</tr>
<tr>
<td>CLOCK_BOOTTIME_ALARM</td>
<td>Interval timer on CLOCK_BOOTTIME with system waking.</td>
<td>Linux 3.0</td>
<td>Partially³</td>
</tr>
<tr>
<td>CLOCK_TAI</td>
<td>Reports International Atomic Time ignoring leap seconds.</td>
<td>Linux 3.10</td>
<td>No</td>
</tr>
</tbody>
</table>

³ Whether the clock is affected by adjustments.
⁴ Only affected by adptime and NTP adjustments.

System calls allow applications to access clock-related functionality. This could entail getting the current date and time, the resolution of a clock, and applying interval timers to existing clocks. Because system calls are expensive due to the mode switch from user mode to kernel mode, numerous improvements have been developed, such as the usage of the vDSO (virtual dynamic shared object), particularly for commonly used system calls relating to time. The vDSO is a small, shared library that is mapped into the address space of all userspace applications to provide kernel information quickly and reduce call overhead. See Table 6.2 for a list of time-related system calls that are explored in this work.

6.1.3 Virtualization and Time

Nowadays, virtualization is widely used. Hypervisor-based virtualization adds challenges to timekeeping. For example, during virtual machine migration, the TSC value
may change, and the frequency may differ between hosts. As a result, several initiatives to improve the TSC, such as paravirtualized clocks (e.g., pvclock, kvmclock, TSC page) [135] and hardware modifications such as TSC scaling, have been documented. The host OS serves as the time reference for all containerized loads in OS-level virtualization. Container migration is similarly prone to having discontinuous temporal jumps [122]. While the issue has been addressed for virtual machines, it remains a challenge with container engines that are commonly used. The time namespace is an attempt to give each container its own view of time, allowing it to retain its time perception even after being relocated.

### 6.2 Reference Implementation

This section demonstrates how containers can utilize the time namespace. Typically, a container engine (such as Docker or podman) is used to deploy a container with the default settings. The container engine would be responsible for applying the correct namespaces and control groups to the application. Popular container engines do not currently support the newly suggested time namespace, necessitating manual application deployment. This section explains how to construct a standard container from
scratch using the `unshare` command from the `util-linux` package [136]. Alternately, one can accomplish the same result by calling `unshare()` [137] with the necessary flags (e.g., `CLONE_NEWNET`, `CLONE_NEWNS`, etc.).

Figure 6.1 displays an activity diagram demonstrating the steps necessary to operate an application under a time namespace as responsibility is transferred from the host to the newly created container. While this diagram only depicts the configuration required for the time namespace, it is assumed that the other options for the `unshare` command have been configured correctly for the other namespaces. The `/proc` pseudo-filesystem is presumed to be mounted in the container. The host begins the procedure by downloading a base image from an image repository (#1). This is similar to how Docker uses base images with the `FROM` command in a Dockerfile. The root filesystem of the image is then extracted (#2). This can be accomplished via `docker export` or `podman`. If the root filesystem does not have all of the essential files to deploy an application, you should alter it to include those files (#3). Notably, unlike Docker, this method does not use a layered filesystem (e.g., `AUFS`, `OverlayFS`) [28], therefore any changes to the root filesystem are persistent. When the root filesystem is complete, execute `unshare` with the appropriate flags to use the specified namespaces (#4). In addition, you must pick a change of root to the root filesystem directory, similar to how `chroot` [138] operates. The `unshare` command creates the namespaces (and hence applies the concept of a container). If the time offsets for the `CLOCK_BOOTIME` and `CLOCK_MONOTONIC` clocks were not specified when the `unshare` command was executed, you must add the necessary offsets to the container’s `/proc/self/timens` offsets file (#5). This must be done prior to launching any application. After establishing the offsets, you must fork and execute your application using `exec` (#6). This can be avoided by including the fork flag in the `unshare` command. The program will then execute within the given namespaces with its own perception of time. This implementation
applies no resource limits, but the mount, UTS, user, IPC, network, PID, cgroup, and

Figure 6.1: Workflow to Implement Time Namespace in Container

To show the capabilities of the time namespace and the lifecycle of programs that

run in containers with virtual time enabled, this chapter uses live migration tools to
demonstrate that a container can maintain its own sense of virtual time regardless of
the host. Specifically, CRIU was used to generate a checkpoint on the first host, migrate
the checkpoint to the second host, and restart the container. While the time of the
second host has been altered on purpose, the container retains its own sense of virtual
time.

6.3 Data Collection

This chapter evaluates the performance impact of time virtualization on applications
relative to a non-virtualized baseline. Information regarding the timing of the execu-
tion of 11 time-related system calls and, when available, the \texttt{vDSO}-accelerated equivalents of these calls is gathered. In this study, system call data is referred to as metrics. The metric name is taken from the system call name, and when the clock or access mode is supplied, it is appropriately labeled (e.g., \texttt{vdso\_gettimeofday}, \texttt{clock\_gettime\_boottime}). Table 6.2 gives a listing of the studied system calls, their intended purpose, and whether they have a \texttt{vDSO} equivalent. Table 6.3 outlines the collected metrics.

Table 6.3: Time System Call Metrics Collected

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Reports</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{clock_getres}</td>
<td>Finds resolution of \texttt{CLOCK_MONOTONIC}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{clock_gettime_monotonic}</td>
<td>Retrieves the time of \texttt{CLOCK_MONOTONIC}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{vdso_clock_gettime_monotonic}</td>
<td>Retrieves the time of \texttt{CLOCK_MONOTONIC} using the \texttt{vDSO}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{clock_gettime_boottime}</td>
<td>Retrieves the time of \texttt{CLOCK_BOOTTIME}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{vdso_clock_gettime_boottime}</td>
<td>Retrieves the time of \texttt{CLOCK_BOOTTIME} using the \texttt{vDSO}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{gettimeofday}</td>
<td>Retrieves the time of \texttt{CLOCK_REALTIME}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{vdso_gettimeofday}</td>
<td>Retrieves the time of \texttt{CLOCK_REALTIME} using the \texttt{vDSO}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{nanosleep}</td>
<td>Suspends execution until time has elapsed in \texttt{CLOCK_REALTIME}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{clock_nanosleep}</td>
<td>Suspends execution until time has elapsed in \texttt{CLOCK_MONOTONIC}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{timer_create}</td>
<td>Creates a per-process interval timer on \texttt{CLOCK_MONOTONIC}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{timer_settime}</td>
<td>Starts or stops a timer on \texttt{CLOCK_MONOTONIC}.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{timer_getoverrun}</td>
<td>Gets timer’s overrun count (on \texttt{CLOCK_MONOTONIC}).</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{timer_delete}</td>
<td>Stops and delete a specified timer (on \texttt{CLOCK_MONOTONIC}).</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{timerfd_create}</td>
<td>Creates a timer on \texttt{CLOCK_MONOTONIC} and returns a file descriptor.</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
<tr>
<td>\texttt{timerfd_settime}</td>
<td>Starts or stops a file descriptor-based timer (on \texttt{CLOCK_MONOTONIC}).</td>
<td>Execution Time</td>
<td>ns</td>
</tr>
</tbody>
</table>

\textbf{Uftrace} \cite{Uftrace} was used to obtain system call timing information. \textbf{Uftrace} is a tool inspired by \textbf{ftrace} that enables the tracing of compiler-instrumented C/C++ applications. \textbf{Uftrace} is an effective instrument for tracing user space functions, library functions, Linux kernel operations, and system events. It was chosen above alternatives such as \textbf{strace} and \textbf{ltrace} because it incorporates the benefits of both while having a smaller overhead for tracing.

Depending on the level of virtualization employed, three levels were investigated:

1. \textbf{Non-virtualized}: The application is placed directly on the host and shares all globally available resources. There are no resource limitations, and the application executes as the root user. This level is known as the \textit{baseline}. 60
2. **Time Namespace Only:** The application is deployed directly on the host and shares all global resources except the system’s time perception. This is achieved through the time namespace by configuring offsets for the `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` clocks. There are no resource limitations, and the application executes as the root user. This level is known as the *namespace* level.

3. **Container:** The application is deployed in a custom-made container and is segregated by utilizing the eight available namespaces (i.e., cgroup, IPC, network, mount, PID, time, user, UTS). Namespaces provide the container with an isolated instance of the global resource, hence decreasing the amount of shared information. The container’s base image is Ubuntu 21.10. Similar to the last namespace scenario, the time namespace was created by specifying offsets for the `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` clocks. There are no resource limitations, and the application executes as the root user. This level is known as the *container* level.

The tests were conducted on a virtual machine instance (4 vCPUs, 8 GB RAM) that was hosted on a 12-core AMD Ryzen 9 3900X. Oracle’s VirtualBox was the hypervisor used in the test. Ubuntu 21.10 with kernel version 5.13.0-22-generic (x86_64) was installed in the virtual machine. Testing locally was favored over testing on AWS instances because, despite the fact that the most recent Amazon Linux AMI contains kernel version 5.10 (which supports time namespaces), the `util-linux` packages are outdated and the `unshare` command does not recognize the time namespace as a valid option. The vCPU and RAM combination was chosen to mirror the resource allocation provided by AWS for a compute-optimized xlarge instance. The virtual machine utilized Ubuntu’s glibc 2.34-0ubuntu3, gcc 11.2.0, and `uftrace` v0.9.4. Oracle’s VirtualBox 6.1.26r145957 was installed as the hypervisor for the guest operating
VirtualBox keeps all guest-visible time sources synchronized with the mono-
tonic host time [140]. The virtual machines were later ported to a QEMU runtime to be managed by Nomad.

Each system call resulted in the collection of 200 records, each of which represented the average of one million system call executions. This was performed to generate a more representative sample and to defray the expense of the initial vDSO call, which typically results in a page fault [141].

6.4 Comparison Methodology

Using hypothesis testing, it was determined if time virtualization (with varied degrees of OS-level virtualization) decreases system performance by increasing the execution time of time-related system calls relative to the control group. Based on the level of OS-level virtualization employed, the tests were divided into three groups: non-virtualized \((n = 200, \text{ baseline})\), namespace \((n = 200)\), and container \((n = 200)\). The Shapiro-Wilk’s test was utilized to assess normality, and Q-Q plots were visually examined to validate any concerns of deviation from normality. Although statistically undesirable, the outliers were retained because there was no reason to reject or invalidate them. Additionally, Levene’s test was utilized to assess the homogeneity of variances.

For each metric with heteroscedasticity, a one-way Welch ANOVA was conducted to assess if the execution time differences across groups with varying amounts of OS virtualization were statistically significant. For those measurements that satisfied the premise of variance homogeneity, a one-way ANOVA was done. The difference between group means was represented in terms of the null hypothesis \(H_0\) (all group population means are equal) and the alternative hypothesis \(H_A\) (the means of the groups are not equal). These tests are suitable due to their resistance to deviations from normality,
especially when sample sizes are large and equal. In cases where the null hypothesis may be rejected (i.e., there is a statistically significant difference between the means of the groups), a post-hoc test (i.e., Tukey HSD for ANOVA, Games-Howell for Welch’s ANOVA) was performed to discover where the differences lay. The data is expressed as the mean ± standard deviation. In all instances and for all tests, results are presented at the 95% confidence level.

6.5 Results and Discussion

This section presents the analysis and results obtained from the collected data. To facilitate the discussion of our findings, the system calls are categorized into three groups based on their functionality: sleep (which includes system calls that induce delays), time (which includes system calls that report time), and timers (concerned with system calls that arm and disarm per-process timers). Refer to Table 6.4 for additional information on the statistical significance of the tests, the difference between the means of the virtualization levels, and their confidence intervals. Figure 6.2 shows the execution time of the system calls.

6.5.1 Sleep Inducing System Calls

clock_nanosleep() and nanosleep() are among the system calls that create delays. Both system calls support high-resolution sleep, which suspends the execution of the calling thread until the set time expires, a signal triggers a handler in the calling thread, or the process terminates [142] [142]. The interval will be rounded up to the next multiple if the requested time is not an exact multiple of the clock’s granularity. The experiment aimed for a delay of 100 milliseconds per system call.
Figure 6.2: Execution Time of System Calls
<table>
<thead>
<tr>
<th>Metric</th>
<th>Descriptives</th>
<th>ANOVA</th>
<th>Post Hoc Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Level</td>
<td>Mean</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>clock, nanoseep</td>
<td>baseline</td>
<td>1.02 × 10^8</td>
<td>5.61 × 10^6</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>1.02 × 10^8</td>
<td>6.26 × 10^6</td>
</tr>
<tr>
<td>clock, nanoseep*</td>
<td>baseline</td>
<td>1.01 × 10^8</td>
<td>2.36 × 10^6</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>1.02 × 10^8</td>
<td>3.07 × 10^6</td>
</tr>
<tr>
<td>clock, getres</td>
<td>baseline</td>
<td>129.92</td>
<td>2.594</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>132.96</td>
<td>1.934</td>
</tr>
<tr>
<td>clock, gettime, boottime</td>
<td>baseline</td>
<td>146.85</td>
<td>1.913</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>147.99</td>
<td>1.751</td>
</tr>
<tr>
<td>clock, gettime, monotonic*</td>
<td>baseline</td>
<td>146.99</td>
<td>1.363</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>146.91</td>
<td>1.598</td>
</tr>
<tr>
<td>gettimeOfDay*</td>
<td>baseline</td>
<td>121.37</td>
<td>1.296</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>123.98</td>
<td>2.833</td>
</tr>
<tr>
<td>timer, create</td>
<td>baseline</td>
<td>282.56</td>
<td>5.219</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>287.11</td>
<td>6.997</td>
</tr>
<tr>
<td>timer, settime</td>
<td>baseline</td>
<td>192.23</td>
<td>4.223</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>194.81</td>
<td>6.666</td>
</tr>
<tr>
<td>timer, delete</td>
<td>baseline</td>
<td>227.05</td>
<td>3.822</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>230.59</td>
<td>4.223</td>
</tr>
<tr>
<td>timer, gettimeofday</td>
<td>baseline</td>
<td>172.18</td>
<td>7.403</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>174.97</td>
<td>5.636</td>
</tr>
<tr>
<td>timerfd, create</td>
<td>baseline</td>
<td>576.83</td>
<td>8.572</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>589.47</td>
<td>10.17</td>
</tr>
<tr>
<td>timerfd, settime</td>
<td>baseline</td>
<td>241.17</td>
<td>3.456</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>249.45</td>
<td>5.407</td>
</tr>
<tr>
<td>vdsoclock, gettime,</td>
<td>baseline</td>
<td>56.81</td>
<td>1.091</td>
</tr>
<tr>
<td>boottime</td>
<td>container</td>
<td>60.34</td>
<td>1.058</td>
</tr>
<tr>
<td>vdsoclock, gettime,</td>
<td>baseline</td>
<td>56.39</td>
<td>0.769</td>
</tr>
<tr>
<td>monotonic*</td>
<td>container</td>
<td>60.36</td>
<td>1.212</td>
</tr>
<tr>
<td>vdsoclock, gettimeOfDay</td>
<td>baseline</td>
<td>55.36</td>
<td>0.886</td>
</tr>
<tr>
<td></td>
<td>container</td>
<td>58.14</td>
<td>1.267</td>
</tr>
</tbody>
</table>

*a* Welch's ANOVA used because metric exhibited heteroscedasticity.

*b* Asymptotically F distributed.

* The mean difference is significant at the 0.05 level.
The metrics associated with this set of system calls share the same name as the system calls they are derived from:

- `clock_nanosleep`
- `nanosleep`

Contrary to `clock_nanosleep()`, which counts time against a user-specified clock, POSIX.1-compliant versions of `nanosleep()` use the `CLOCK_REALTIME` clock. In Linux, `nanosleep()` uses `CLOCK_MONOTONIC` because it is unaffected by discontinuous jumps in the system time, a requirement for `nanosleep()` when using `CLOCK_REALTIME`. Consequently, Linux's implementation is functionally equivalent to the POSIX-compliant version. In contrast to `clock_sleep()`, `clock_nanosleep()` can sleep until an absolute time is reached, rather than relying solely on sleep intervals. This gives the user more control and prevents a process from sleeping for an extended period if the time has been modified by an administrator or updated via NTP.

The execution time of `clock_nanosleep()` increased from the baseline (101.57ms ± 5.61ms) to the namespace (102.19ms ± 6.26ms) and to the container (102.29ms ± 6ms). Similarly, `nanosleep()` execution time climbed from the baseline (101.12ms ± 2.36ms) to the namespace (101.34ms ± 2.38ms) and then to the container level (101.59ms ± 3.07ms). Although the kernel source code indicates clock offsets being considered for `clock_nanosleep()`, similar to how `adjtime` and NTP adjustments are taken into account, the differences in virtualization levels in both system calls were not statistically significant.

Because the sleep mechanism is mostly dependent on the scheduler, it was anticipated that sleeping calls across virtualization levels would differ little. A sleeping task surrenders the processor to the scheduler, exposing it to additional scheduling latency, which may prevent it from running immediately after the sleeping phase finishes. In
addition, other sources of latency may be created due to the fact that the kernel may execute other sleeping tasks while the process sleeps. As other processes execute, the tasks may also compete for the instruction cache, which may result in stalls when the process resumes owing to cache misses. This is not quantified in this study, but it is possible to do so by employing performance counters. This effect was minimized to the greatest extent possible by conducting the experiments exclusively.

Linux is not a real-time operating system, hence precise sleep intervals cannot be guaranteed. Due to the fact that all processes, including containerized ones, are scheduled by the same kernel, you may count on sleeping for a duration close to the one given [143]. Therefore, Madden [144] suggested that when processor monopolization is not a concern, the use of nanosleep() should be discouraged in favor of busy wait. Notably, the granularity of sleeping system calls has improved since the Completely Fair Scheduler (CFS) became the default scheduler [145].

### 6.5.2 Time Reporting System Calls

In the domain of time reporting system calls, clock_getres(), clock_gettime(), and gettimeofday() as well as their vDSO equivalents were evaluated. This is due to the fact that these system calls report time or its precision. clock_gettime() and gettimeofday() are used by the system to retrieve time information, whereas clock_getres() determines a clock’s resolution. gettimeofday() exclusively reports CLOCK_REALTIME, as opposed to clock_gettime() which allows you to select the clock whose time you wish to report. The following metrics are related to this group:

- clock_getres
- clock_gettime_monotonic
- clock_gettime_boottime
- gettimeofday
- vdso_clock_gettime_monotonic
- vdso_clock_gettime_boottime
- vdso_clock_gettimeofday

All metrics in this group showed statistically significant differences in execution time across virtualization levels.

The `clock_getres()` execution time increased from the baseline (129.92ns ± 2.594ns) to the namespace level (133.36ns ± 2.394ns). From the namespace level to the container level, there was a slight drop (132.96ns ± 1.934ns). The `gettimeofday()` execution time increased from the baseline (121.37ns ± 1.296ns) to the namespace level (123.98ns ± 2.833ns). From the namespace to the container level, there was a slight decrease (123.44ns ± 2.221ns). The execution time of `clock_gettime()` increased from the baseline (146.85ns ± 1.913ns) to the namespace (148.06ns ± 2.004ns) level when CLOCK_BOOTTIME was given. There was a minor decrease in performance from the namespace to the container level (147.99ns ± 1.751ns). The execution time increased from the baseline (146.09ns ± 1.363ns) to the namespace level (147.29ns ± 1.999ns) when CLOCK_MONOTONIC was set. From the namespace to the container level, there was a small reduction (146.91ns ± 1.598ns). The rise from the baseline to the namespace level and the increase from the baseline to the container level were statistically significant for the aforementioned system calls, however the difference between the namespace and container levels was not.

When analyzing the vDSO counterparts, the execution time of the accelerated `clock_gettime()` increased in the following order: baseline (56.81ns ± 1.091ns), namespace (59.89ns ± 0.947ns), and container level (60.34ns ± 1.058ns). The time increased
as follows when \texttt{CLOCK_MONOTONIC} was used: \textit{baseline} (56.39ns ± 0.769ns), \textit{namespace} (59.96ns ± 0.920ns), and \textit{container} level (60.36ns ± 1.212ns). The accelerated \texttt{gettimeofday()} execution’s time grew incrementally from the \textit{baseline} (55.36ns ± 0.886ns) to the \textit{namespace} (57.58ns ± 1.114ns), and the \textit{container} level (58.14ns ± 1.267us). All \texttt{vDSO} calls demonstrated statistically significant increases from the \textit{baseline} to the \textit{namespace} and \textit{container} levels.

The examined system calls utilized \texttt{CLOCK_REALTIME}, \texttt{CLOCK_MONOTONIC}, and \texttt{CLOCK_BOOTTIME}. Only \texttt{CLOCK_MONOTONIC} and \texttt{CLOCK_BOOTTIME} are virtualized via the time namespace. Every non-\texttt{vDSO} call exhibits a statistically significant difference between the \textit{baseline} and all OS-virtualization levels, but not between execution in a time namespace and a fully namespaced container. Because \texttt{CLOCK_REALTIME} is shared with the host OS, it was not anticipated that \texttt{gettimeofday} would result in a significant difference between the \textit{baseline} and other virtualization levels. However, the increase was not too costly, with our worst-case scenario adding 2.15\% in overhead, in line with the 3\%-5\% overhead that is associated with the use of containers [108] [109] [110]. Surprisingly, the addition of other virtualization features (i.e., more namespaces) on top of the time namespace did not result in a significant change in the execution time of system calls. This indicates that the time namespace may be added to existing container standards as its overhead margins are acceptable.

In terms of \texttt{vDSO} accelerated calls, the change from the \textit{baseline} is significant at all OS-virtualization levels (from a time \textit{namespace} to a full \textit{container}) and across virtualization levels. The mean difference is greater when accessing the \texttt{CLOCK_MONOTONIC} and \texttt{CLOCK_BOOTTIME} clocks, with ranges of 2.83 – 3.78 and 3.37 – 4.21ns, respectively, than when accessing the \texttt{CLOCK_REALTIME} clock, with a range of 1.98 – 3.03ns. This could be due to the additional calculations required by the kernel to maintain the time offsets for virtualized clocks. No additional experiments were undertaken to confirm
that vDSO calls are faster than system calls since (1) it was outside the scope of this study and (2) it is commonly acknowledged that system calls have a greater access latency than vDSO calls. Due to access latency, these slight discrepancies observed with the vDSO calls may have been amortized in the system call results, resulting in variances between the namespace and container levels that are not statistically significant.

6.5.3 Timer System Calls

This study analyzes the following per-process time system calls: `timer_create()`, `timer_settime()`, `timer_gettime()`, `timer_delete()`, `timerfd_create()`, and `timerfd_settime()`. These system calls create, arm, disarm, delete, and report the overrun count of a timer. The “fd” versions allows setting up timers using file descriptors, which is a more event-loop friendly method than ordinary timers (i.e., you can utilize `select()`, `poll()`, and the `epoll` API). The metrics associated with this system call group have the same name as the system calls from which they were produced:

- `timer_create`
- `timer_settime`
- `timer_gettime`
- `timer_delete`
- `timerfd_create`
- `timerfd_settime`

There was a statistically significant difference in execution time between virtualization levels for all system calls in this group.
timer_create()’s execution time climbed from the baseline (282.56ns ± 5.219ns) to the namespace (285.83ns ± 5.980ns), and then to the container level (287.11ns ± 6.097ns), in that order. The execution time of timer_settime() increased from the baseline (192.23ns ± 4.223ns) to the namespace (194.03ns ± 3.666ns), and then to the container level (194.81ns ± 4.223ns), in that order. The execution time for timer_delete() climbed from the baseline (227.05ns ± 3.822ns) to the namespace (229.67ns ± 4.226ns), and then to the container level (230.59ns ± 4.711ns), in that order. The “fd” variations followed the same pattern. The timerfd_create() execution time climbed from the baseline (576.83ns ± 8.572ns) to the namespace (589.08ns ± 10.017ns), and then to the container level (589.47ns ± 11.378ns), in that order. The execution time of timerfd_settime() increased from the baseline (241.17ns ± 3.456ns) to the namespace (249.45ns ± 5.407ns), and finally to the container level (249.79us ± 5.378ns). In the aforementioned system calls, the increase from the baseline to the namespace level, as well as the increase from the baseline to the container level, were statistically significant, while the difference between the namespace and container levels was not.

The timer_getoverrun() system call execution time increased from the baseline (172.18ns ± 7.403ns) to the namespace (174.97ns ± 5.636ns), and finally to the container level (178.16us ± 7.200ns). The increase from the baseline to the namespace level was statistically significant, as was the increase from the namespace to the container level.

The results show that there is no discernible difference in execution times while creating, arming, disarming, and removing timers on CLOCK_MONOTONIC, in both traditional and file descriptor system calls variations. The file descriptor versions, on the other hand, have longer mean execution times, which is most likely due to the underlying open files interacting with the timer. timer_getoverrun() was the lone
exception, with statistically significant changes in execution times across all virtualization levels. There is no overrun system call provided for file descriptor variants, as this information is collected via the **read()** system call. Under our worst-case scenario, the time to create a timer increased by 2.19%, the time to arm/disarm a timer increased by 3.57%, and the time to delete a timer increased by 1.56%. In all cases, the overhead is less than 4%, which is similar to the overhead associated with containerized loads [108][109][110].

### 6.6 Related Works

There have been other experiments towards virtual time recorded in the literature. This section excludes works that employ virtual machines exclusively and favors those that use some type of OS-level virtualization (e.g., namespaces, containers, jails, zones). This study focuses on their use of virtual time and any discussion of performance overhead.

Zheng and Nicol [108] created a virtual time system that used OpenVZ’s Virtual Environments to replicate the functional and temporal behavior of network communication by trapping the execution of system calls and returning an illusion of virtual time as required by the simulation. This is one of the early attempts to use OS-level virtualization to decouple the virtualization of execution and time in order to prevent the execution from reflecting the host’s job serialization. Their technique necessitates changes to the OpenVZ kernel in order for tasks to be properly scheduled, as well as changes to system calls such as **gettimeofday()**. In their trials, they discovered a 4.9% overhead over the non-virtualized baseline. Jin et al. [146] extended the work by using this custom kernel to support **S3F**, a parallel network simulator. They introduced the ability to advance in virtual time only when there is activity in an application or
the network. Although not explicitly stated, it is assumed that they inherit the same overhead as Zheng and Nicol’s [108] work due to their use of the same custom kernel.

Lamps et al. [147] introduced a set of Linux kernel modifications in TimeKeeper to incorporate Linux Containers (LXC) into virtual time for network simulation. The main concept is to give each container a dilated view of time to make it seem as if time advances more slowly than real time to make network resources appear faster. To accomplish this, they modified the Linux task_struct to include the dilation factor and other timekeeping variables, exposed an API to control time operations (e.g., dilation, freeze/unfreeze, time leaping, etc.), modified system calls (e.g., gettimeofday(), sleep(), poll()), and used hrtimers to schedule container execution. The choice of hrtimers had an effect on virtual time accuracy, however in 90% of situations they were able to keep virtual time within 4us of the intended virtual time. This is due to their approach’s lack of scheduling flexibility, which involves submitting a process to a fixed execution time slice. They did not examine each modification for overhead when evaluating their technique for overhead, but rather how many containers they could deploy while keeping the validity of virtual time. This was shown to be connected to the time dilation factor utilized for the containers: $6/(TDF + 1)$.

TimeKeeper has been the inspiration for the saga of work presented by Yan and Jin. They expanded the task_struct fields and added a pair of system calls to un-share time and set the dilation factor [148]. They combined these developments with the work of Handigol et al. [149] in order to endow the Mininet-Hifi emulator with virtual time capabilities. They improved it by incorporating a freezing system capable of pausing and restarting a container’s virtual time. This was a source of trouble for Mininet-Hifi since containers use the same system clock of the physical machine and this leads to a wrong perception of time because a container’s clock keeps advancing even if it is not running. Yan and Jin molded their work into a Linux namespace,
the clock namespace \cite{150,151}. The final implementation, VT-Mininet, resembles modern day practices because there is use of control groups (via Mininet-Hifi), container primitives, and traffic shaping. The average overhead for a \texttt{gettimeofday()} system call in their experiments was 13ns. They claim to have identified a 0.03% overhead after 1,397,829 calls to \texttt{gettimeofday()}. Because some time-related system calls are expedited using vDSO, the containers can avoid the features implemented by Yan and Jin. They disabled the use of vDSO for specific system calls as a workaround. 

\textbf{Containernet} \cite{152} is another Mininet fork in which Docker containers are utilized instead of the manual combination of container-building primitives.

Navarro et al. \cite{153} used a counter of time-related system calls invoked by a process to implement virtual time. This guarantees a monotonic logical time, which can be utilized to ensure reproducibility of execution phases in containerized processes. They intercept vDSO calls and replace them with system calls in their implementation. Their work does not investigate the overhead of their virtual time solution. The greater access latency of system calls when compared to vDSO calls, on the other hand, is well documented.

The use of Jails in BSD-based systems is another application of OS-level virtualization. Grau et al. \cite{154} implemented time virtualization (based on time dilation) for virtual nodes using a hybrid technique that used VMs, Jails, and BSD’s Virtual Routing (i.e., network processes). Their system evaluation did not include runtime overhead tests, instead focusing on the memory consumption of each virtual node and discovering that their solution was lighter by consuming less memory than those implemented under Xen for Linux-based systems. Unfortunately, because no runtime overhead was provided, their solution cannot be compared to others.

\textbf{Emulab} was adapted by Hibler et al. \cite{155} to use FreeBSD Jails. Emulab makes use of the concept of virtual time to ensure the order of occurrences in an experiment \cite{156}.
The modifications were designed to allow for the emulation of systems larger than the underlying testbed via light virtualization. *Emulab* can now use *Docker* containers as well as virtual machines. [157] There was no mention of virtualization overhead because the authors believed that fidelity was not as critical for small scale simulations. Starting with Solaris 11, virtual time has been introduced in Solaris Zones, allowing each non-global zone to set its own time via the `clock_settime()` system call. [158]
Chapter 7

Impact on Software Maintenance

Approximately 40% of a developer’s time is spent fixing bugs [159]. This is due to the fact that debugging is “largely a matter of trial and error.” [160] In other words, debugging is analogous to a scientific, iterative procedure that begins with the selection of a debugging technique, followed by the gathering of evidence and formulation of a hypothesis [161]. The fact that the primary source of issues has varied over time is not helpful. Prior to the 2000s, memory-related (e.g., overwriting reserved memory, index out of bounds, etc.) and vendor-related problems (e.g., buggy compilers, faulty logic boards, etc.) were the most prevalent [162]. Early in the 21st century, parallel and concurrent behavior in software was the key culprit [159]. With the widespread adoption of cloud-native software and microservices, as well as their expansion to the edge, interaction-based problems (e.g., lack of sequencing control in replies to asynchronous invocations of other microservices, connection timeout settings) have become the most prevalent cause of errors [163].

In addition to the concurrent behavior of several microservices, hardware heterogeneity and the combinatorial expansion of supporting various standards, protocols, and integration middleware also contribute to interaction-based issues [164]. In these
conditions, it is more difficult to detect miscommunication, cascading failures, data integrity issues, service discovery, and authentication failures, which hinders fault localization and debugging [163]. This is troublesome at the edge due to the difficulty of recreating malfunctioning environments [164]. In addition, because microservices are dynamic, the conventional method of placing breakpoints on locally running services may not result in successful fault localization because the local service state may differ from the production service state. In addition, a breakpoint may be reached via many execution paths, which complicates attempts at fault localization. Furthermore, because deployments may be conducted on a range of hardware, certain failed execution paths may go unreported.

Despite the proliferation of debugging tools in the 2000s, developers have maintained their debugging methods, relying primarily on symbolic debuggers. Even when testing solutions for microservices at the edge are available, only 9% of developers utilize them due to poor support for the multitude of edge devices [164]. In addition, while node-level logging is useful for identifying problems, “there is no environment that tracks everything.” [164] Even with good logging and tracing, it can be difficult to locate a bug in a “haystack” of logs and traces. In other instances, the transitory nature of microservices, the difficulties caused by connectivity and mobility constraints [74], and the persistent threat of device churn at the edge [70] further complicate the task.

This chapter examines contemporary techniques to microservice debugging and explains how their applicability to fog/edge contexts is constrained by hardware limitations, heterogeneous architectures, excessive runtime overhead, and complex deployments. It also presents a solution that aims to reduce the time required to debug microservices and their interactions by employing checkpoint/restart (C/R) techniques to reproduce malfunctioning environments across a variety of hardware configurations and without requiring code instrumentation or specialized kernels. This solution is
adaptable and future-proof, with the ability to leverage the newly introduced time namespace to enable actual checkpoint-enabled time-travel debugging for containers by giving the recreated environment its own perspective of system time that may differ from the host. To estimate the overhead of this technique in an orchestrated environment, two models are presented: one that emphasizes explainability at the expense of accuracy for the most extreme loads seen at the edge, and the other that concentrates only on accuracy.

7.1 Additional Background

Debugging software is a time-consuming yet necessary activity. It has been dubbed the “dirty little secret of computer science,” [160] and it is believed to be “twice as hard as writing the program in the first place.” [165] Throughout this process, developers make an effort to link an observed failure to a defect, or bug. According to Gould and Drongowski (1974), debugging is an iterative process that includes choosing a debugging technique, uncovering clues, and forming a hypothesis based on the results [161]. This process of thinking backwards, from failure to an underlying cause, is known as scientific debugging.

7.1.1 Debugging Practice Advances

It has been found that as developers acquire expertise, their techniques become more flexible, resulting in more accurate mental representations of the program’s goals. When tasked with discovering flaws in unfamiliar code, Katz and Anderson (1987) found that engineers try to focus on gathering data to understand how the program operates [166]. A couple of years later (1989), Vessey emphasized that the mental model, rather than the way the software is structured, is the most important factor in
deciding how difficult it is to locate a flaw [167][168]. Eisenstadt discovered in 1997 that most developers have trouble finding a bug when there is a “spatial chasm” between the underlying cause and the symptom [169]. In other words, the symptoms manifested themselves in a location far from the origin of the problem. He discovered that most developers preferred print statements and code inspection over tools when they added overhead or resulted in “heisenbugs”, bugs whose behavior is altered during debugging. The major causes of bugs back then were memory related (e.g., overwriting reserved parts of memory, index out of bounds, etc.) and defective vendor issues (e.g., buggy compilers, faulty logic boards, etc.).

In the first decade of the 2000s, research was mostly focused on identifying specific bug categories, developing tools to automate bug detection, and studying how students are taught debugging and formal testing methods. Despite the availability of various debugging tools, Perscheid et al. [159] discovered in 2017 that most engineers relied on symbolic debuggers and print statements. He observed that engineers spend 20–40% of their time debugging code, and that most issues took more than a week to resolve. Most of the time, problems were caused by faulty design and parallel behavior. Perscheid et al. confirmed Eisenstadt’s results by discovering that difficult-to-debug problems are mainly caused by bugs that manifest far from their origin and, in certain cases, by the difficulty to recreate the behavior in a controlled environment. In other situations, the use of debugging tools impeded problem detection since the overhead made debugging difficult.

Despite having access to more modern tools, Beller et al. (2018) discovered that developers see print statement debugging as the universal debugging approach [170]. Most developers, according to their research, do not use their IDE’s capabilities, and when they do, they merely use basic line breakpoints to halt the program’s execution and walk through the code line by line. He noticed that most developers get frustrated
during the first few minutes of a debugging session since they had to restart the ses-
sion after stepping past the region of interest. Beller et al. revealed that component
interactions are implicated in the most difficult issues to solve. They are a frequent
cause of integration issues and debugging efforts.

That same year, Zhou et al. [163] investigated debugging microservices and dis-
covered that logging is at the heart of the effort, followed by the use of current fault
localization techniques. In most situations, it takes developers many days to iden-
tify the underlying reasons. Zhou et al. categorized the primary reasons as internal,
interaction-based, or environmental. Internal causes lie in the implementation of each
individual component. Interaction-based problems emerge when several microservices
fail to interact correctly, such as a lack of sequencing control in responses to asyn-
chronous invocations of other microservices. Environmental faults arise in the setup
of the runtime infrastructure; these are generally associated with the container engine,
the orchestration platform, and the resources available for a service. To determine
what is wrong, the developers look through the logs and try to replicate the runtime
environment. This is often repeated until the failure, its extent, and, finally, the un-
derlying causes and location are discovered. The average time to fix a problem grew
as the number of microservices increased. A single microservice takes ten hours, two
microservices take twenty hours, three microservices take forty hours, and more than
three microservices take more than 48 hours [163].

In practice, a large-scale microservice-based system does not consist of a few services
but rather hundreds of microservices that are deployed in containers [171]. Interactions
between containers running on several hosts, which can at times be asynchronous, can
be challenging to trace and comprehend. While cloud and fog/edge deployments are
similar in certain ways, fog/edge deployments are more complex due to hardware con-
straints, heterogeneous ISAs, diverse peripherals, multiple communication interfaces,
and performance limitations due to tight temperature monitoring or energy-saving measures. In addition, with current initiatives to decompose containers (into application logic and environment containers) [62][63] to reduce resource burden at the edge, maintaining and debugging microservices has proven challenging, with fault localization times growing exponentially as the number of microservices involved increases.

Understanding the interactions between components and the topology of the system is an effective method for debugging distributed systems. In fault localization efforts, manual testing, tracing, logging, and visualization are prioritized. Microservices, however, are more challenging than typical distributed systems due to their ephemeral nature, particularly at the edge, where nodes can frequently enter and exit service regions due to mobility, connectivity challenges, and power constraints. Standard tracing, logging, and visualization software cannot assume that a host is solely responsible for a service and should instead support multiple logical views of the same system: an aggregated service-level view, a shared view for services that share the same state, and a container-level view for each service. Moreover, representations that rely exclusively on logs may not always uncover all the information necessary to establish a “happens-before” relationship between events [172][115]. In a similar manner, time-drift between nodes may lead to improper ordering [173].

### 7.2 Design and Implementation

This section introduces a checkpoint/restart enabled technique to enhance the developer’s debugging practices. By recommending a method for duplicating the state of microservices at the edge, the cost of constantly reinitializing environments until a bug is discovered is decreased. This method can be utilized regardless of heterogeneous hardware to aid developers in their debugging practices and, if desired, encourage co-
operation. The objective is to achieve this without imposing a new tool outside of the
developer’s comfort zone and avoiding unnecessary overhead that would render tools
useless to developers by modifying the behavior of a problem (i.e., “heisenbugs” and
execution timing sensitive bugs).

7.2.1 Reference Deployment

In a typical microservice deployment, an orchestrator (such as Kubernetes, Nomad,
or Mesos) coordinates the distribution of service loads across several hosts in accor-
dance with placement criteria. These criteria permit the orchestrator to assign the
optimal host for a service depending on its specifications. A deployment typically in-
volves distributing containers across various hosts and customizing their interaction.
Figure 7.1 displays a simplified picture of a system, omitting the orchestrator and
other container hosts while focusing on a single container and its connection to the
remote host from which the user begins debugging. The container consists of a base
image, the developer’s application, and a debugging server. Stateful and legacy ap-
lications may demand persistent storage space on the host for their operations, as
indicated by the volumes attached to the container in the reference deployment. Addi-
tionally, some applications may require a container init system such as tini [174] or
dumb-init [175] for signal handling. Because each container requiring debugging must
include a debugging server, this method already resembles how most developers debug
remote applications and aims to minimize introducing additional debugging overhead.
Therefore, rather than adding the burden of learning yet another debugging tool, the
proposed technique acts as a complement to the tools with which the developer is
already accustomed. Notably, although the reference diagram does not account for
orchestrators, it is compatible with loads deployed by orchestrators.
7.2.2 Reference Workflow

As reconstructing deployment environments is a significant source of debugging challenges in edge deployments, the solution suggested in this chapter aims to record the whole state of a container to disk. To do this, CRIU [97], an application checkpoint and restore tool, is utilized to capture and save the state of the container and its environment. While CRIU can dump the state of an application, it cannot do so for processes with attached debuggers due to limitations inherited from the use of the ptrace API, which does not permit multiple tracers to inspect a task.

This limitation is circumvented by utilizing a container’s use of cgroup subsystems, namely the freezer subsystem. Once this limitation is removed, CRIU will be able to checkpoint processes at precise line numbers or with granularity comparable to breakpoints. This allows the capture of a program and its environment at any point in time. This container checkpoint/restore technique is based on the author’s earlier work describing the live container migration technique [121]. This strategy, along with others outlined in [122], takes into account the networking components of an application and the parties with whom it communicates in order to simplify the replication of
microservice-based applications across many networks.

With the recent inclusion of the time namespace to the Linux kernel, the time drift between checkpoint and restore (which was a potential cause of TCP disconnections at restore time) can also be taken into account and avoided.

Consequently, not only the application but also the interaction of its components may be captured, and the reference deployment can be modified to accommodate other services that interact with the deployed application. The simultaneous freezing of all other containers would enable the capture and debugging of the full networked environment. In addition, because this capture recreates the full environment, other tools such as distributed tracing systems (e.g., Zipkin [176], Jaeger [177]) can be utilized to augment log analysis efforts in fault localization.

The Linux control group mechanism is important to the workaround for the constraint described above. A control group permits the splitting of a set of jobs into groups based on a set of parameters that impose usage limitations on the resources. The cgroup freezer permits the job management system to start and stop tasks. This is particularly important since sequences of SIGSTOP and SIGCONT signals are not always sufficient to suspend and resume userspace operations because the signals are observable and can be captured by processes. Instead, the freezer cgroup utilizes kernel freezer code that is transparent to tasks, giving it complete control, even over applications such as gdb.

Figure 7.2 depicts a breakpoint-like granularity checkpoint, wherein an application and its environment are checkpointed at a debugger-inserted breakpoint. The user begins by requesting the launch of a container (#1). The host executes the container (#2), and the deployed application is launched with a specified PID (#3). The user then launches gdbserver (#4) within the deployed container using the PID (#5) of the containerized application as the target. Execution is therefore suspended (#6)
until the user creates a remote connection (#7) and transmits instructions (#8). The user may then place breakpoints (#9) and resume application execution (#10, #11) thereafter. Once the breakpoint (#12) is reached, execution is again halted (#13, #14) and the user can then proceed to freeze the application. This entails generating a leaf (or subdirectory) within the container’s freezer subsystem (#16), relocating the application’s process tree to the newly generated leaf (#17), and then freezing the leaf (#8). The application is therefore suspended (#19). The user then detaches the debugger (#20). This program cannot move since it has been halted at the breakpoint and frozen. Normal operation would continue for non-frozen (or thawed) programs once the debugger is removed. This results in the termination of the gdbserver process within the container. The user is now able to freeze the entire container (#21), necessitating the freezing of the global freezer subsystem (#22). The user can then thaw the previously built freezer subsystem leaf (#23, #24), resulting in an application that is thawed and ready to execute once the global freeze thaws. Since the container is still frozen, CRIU can be used to checkpoint (#25) the container and its environment. By default, the container is destroyed after a checkpoint (#26), but this behavior can be altered to keep it operating if required. Once completed (#27), the user will have access to the container’s root filesystem, its state, and all environment components for subsequent recreation, analysis, and debugging.

7.2.3 Reference Implementation

The reference deployment utilizes runC 1.0.1 as the container host engine and CRIU 3.15 as the checkpoint/restore tool. Docker 20.10.7 is used to generate and export container images (or Dockerfiles) as an OCI (Open Container Initiative) bundle that runC may utilize. Other OCI bundle generators, such as podman, may be substituted for it. This technique uses runC directly, rather than the Docker daemon or the direct
Figure 7.2: Sequence Diagram for Checkpoint Process
connection between podman and runC. This is because it provides greater flexibility during deployment, checkpointing, and environment recreation. An OCI bundle requires a custom-built spec file. This file indicates which program is to be executed within the root filesystem of the container, as well as extra mount locations, hooks, namespaces, capabilities, and cgroups. Each OCI bundle comes with a default spec file that must be modified to accommodate the application. In the proposed implementation, the capability \texttt{CAP\_SYS\_PTRACE} is added to the spec file for debugged containers. Additionally, the user is permitted to select volumes as mount points accessible to the container. Figure 7.1 displays the container template in general. This container contains the application being debugged, gdbserver, and a container init system. This template enables the user to debug gdb-compatible programming languages, such as Ada, Assembly, C, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal, and Rust.

The template can be modified to target other languages. When debugging a pre-built image, a debug variant can be created by using the original image as the basis image for the debug version and adding the components of the template. In addition, the user can deliver updated artifacts via the volumes in the spec file for more flexibility. This eliminates the need to rebuild an image after every debugging-related modification.

Sometimes it is required to debug containers that have already been deployed. If \texttt{CAP\_SYS\_PTRACE} was not allocated to the container when it was constructed, more steps are required. Because it is impossible to modify the capabilities of a running container, it must be recreated with the required capabilities. This additional step would need the consecutive use of CRIU to checkpoint/restore the container on the same host, as well as the addition of the \texttt{CAP\_SYS\_PTRACE} capability. Since this is almost instantaneous (for the majority of microservices), the user will experience minimal
disruption. Using this workaround, debugging is possible for any container that has been deployed. It should be noted that, if granted the `CAP_SET_PCAP` capability, some applications may be able to grant themselves (or their children) additional permissions, but this is not recommended. Notably, the same method may be used to adjust the resource restrictions of the recreated environment, enabling the developer to evaluate a task under different conditions.

The proposed design does not mandate a particular orchestration engine; rather, the usage of orchestration services improves debugging by removing constraints from the reference deployment depicted in Figure 7.1. By launching the server as a sidecar service with a shared process namespace, the necessity for a debugging server in the application’s image (and, by extension, the need to produce debug versions of the images) can be eliminated. This is a typical pattern used to co-locate services within a pod in Kubernetes environments [178]. In addition, the suggested method is probe-friendly; after restoration, a process will continue to be subject to periodic readiness and liveness tests, as the developer intended.

### 7.2.4 Limitations

This solution has restrictions due to the checkpoint/restore process. CRIU may refuse to dump an application if it utilizes a direct connection to hardware in such a way that it may have loaded state into the hardware and there is no generic mechanism to retrieve the information for a proper dump and subsequent restore. There are exceptions for devices such as the null, zero, and other virtual network devices (such as those utilized by OpenVPN). In some instances, this constraint can be circumvented by adding a layer of virtualization to the device or exposing it as a service via an HTTP endpoint.

In the case of GPUs, for example, Eiling et al. [179] devised a workaround for Nvidia GPUs by proposing a virtualization layer that can be deployed via dynamic
linking, thereby eliminating the need to instrument the original code and permitting GPU access to be detached from the application. This exposes CUDA-as-a-service and enables CRIU to checkpoint the program using Cricket, their suggested software, while retaining the GPU state. AMD has taken comparable measures to guarantee that its ROCm (Radeon Open Compute Module) workloads can be transparently checkpointed and restored by altering kernel drivers and producing a plugin to extend CRIU's capabilities [180]. The result of their efforts was the first C/R of TensorFlow workloads.

Hou et al. [?] employ a split process technique to isolate application code from library code in C/R OpenGL programs using CRIU.

Other incompatible circumstances for C/R utilizing CRIU include the usage of specialized interprocess communication techniques. Incompatible scenarios include sockets that are not TCP, UDP, UNIX, packet, or netlink. Some inter-process communication facility settings, including corked UDP datagrams (data gathered into a single datagram and sent when the UDP_CORK option is disabled) and packetized pipes (pipes constructed with O_DIRECT that conduct I/O in packet mode), are also incompatible.

While CRIU is compatible with numerous architectures (such as x86-64, ARM, AArch64, PPC64, and s390), restoring requires that both the host and the target have the same instruction set architecture (ISA). This is because the checkpoint maintains execution environment and application state information. If the application utilizes ISA extensions such as Intel’s AVX or AMD’s 3DNOW, it expects a suitable CPU to be restored. This does not imply that the process must be restored on the same processor, but rather that the host must support the same subset of ISA extensions.

When utilizing this technique for debugging at the edge, it is optimal to attempt a restoration on equivalent hardware. This limitation can be overcome by virtualizing container execution with QEMU (for applications that can benefit from newer extensions but do not require them, resulting in a smaller set of required ISA extensions) or by

89
implementing Barbalace et al’s proposed changes for edge environments, which enable ISA heterogeneity through the use of intermediate representation (IR) binaries [181].

7.3 Data Collection

This chapter proposes a solution to reduce the software maintenance efforts undertaken at the fog and edge layers. The application is momentarily frozen while an application checkpoint is being prepared, and any interruption must be small for services not to be irrevocably impacted. As a result, it is important to estimate the downtime associated with checkpointing the application for debugging purposes. When the checkpoint operation is finished, the application resumes execution where it left off. While a complete set of services can be checkpointed at the same time to ensure that a service and its dependencies are frozen and not affected, knowing how long a checkpoint will take might be useful when considering planned service maintenance and live debugging/profiling sessions.

With this in mind, the checkpointing process was timed on numerous hosts with varying degrees of an application’s memory consumption. Each host was profiled by obtaining information such as the number of CPU cores, their architecture, and the total amount of RAM in the system. Other areas such as the performance of the disk attached to the system were also profiled since the disks house the application dump. To do so, two commonly used disk benchmarking tools were employed, dd [182] and fio [183]. The throughput was tested with dd with 4KiB and 128KiB sized block loads. Another throughput test was conducted with fio, with 128KiB sized blocks and another with 4KiB sized blocks to achieve the maximum number of IOPS. Following the profiling tests, checkpointing tests were performed with the test program allocated 64, 128, 256, 512, 1024, and 2048 MiB of RAM. Thirty measurements were obtained.
for each memory level.

These tests were carried out in an AWS Virtual Private Cloud (VPC). Ubuntu 21.04, runC 1.0.1, and CRIU 3.15 were installed on each instance. These tests covered 27 instance types. We examined every ephemeral storage disk size available to an instance family, including general, compute, memory, and storage optimized instances. In addition, EBS-only instances were also included and their provisioned throughput and IOPS varied. For these instances, gp3 (general purpose SSDs) disks and provisioned different IOPS (3000, 6250, 9500, 12750, and 16000) and several bandwidth levels (150, 250, 500, 750, and 1000 MiB/s) were tested. For each IOPS level, all the listed bandwidth levels were explored. The instance supplied with 3000 IOPS was the sole exception, as AWS requires a minimum of IOPS to obtain the maximum provisioned bandwidth of 1000 MiB/s. Burstable instances were excluded in the experiment to eliminate the effects of accumulated CPU and EBS credit bursts. Overall, data was collected on 13140 observations. Table 7.1 has a list of the instance types that were tested, and Table 7.2 contains a list of the metrics that were collected.

7.4 Prediction Modeling Methodology

To estimate the checkpointing time that would be incurred with our proposed method, an initial effort at linear multiple regression was made. This preliminary model took into account the host’s features (architecture, number of CPU cores, instance memory, storage class, disk bandwidth, and disk IOPS) as well as the checkpointed process memory consumption.

The dependent variable (i.e., elapsed time) was not normally distributed, was highly skewed, and too peaked, as measured by the skewness and kurtosis values. As a workaround, and to improve predictions, we used an inverse transformation on the
Table 7.1: Amazon EC2 Instance Types Tested

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Architecture</th>
<th>vCPUs</th>
<th>RAM (GiB)</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>c5ad</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>4</td>
<td>75 GiB NVMe SSD</td>
</tr>
<tr>
<td>c5d</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>4</td>
<td>50 GiB NVMe SSD</td>
</tr>
<tr>
<td>c6gd</td>
<td>large</td>
<td>aarch64</td>
<td>2</td>
<td>4</td>
<td>118 GiB NVMe SSD</td>
</tr>
<tr>
<td>d2</td>
<td>xlarge</td>
<td>x86_64</td>
<td>4</td>
<td>30.5</td>
<td>3 x 2000 GB HDD</td>
</tr>
<tr>
<td>d3</td>
<td>xlarge</td>
<td>x86_64</td>
<td>4</td>
<td>32</td>
<td>3 x 2000 GB HDD</td>
</tr>
<tr>
<td>d3en</td>
<td>xlarge</td>
<td>x86_64</td>
<td>4</td>
<td>16</td>
<td>3 x 14 TB HDD</td>
</tr>
<tr>
<td>h1</td>
<td>2xlarge</td>
<td>x86_64</td>
<td>8</td>
<td>32</td>
<td>2000 GB HDD</td>
</tr>
<tr>
<td>i3</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>15.25</td>
<td>475 GB NVMe SSD</td>
</tr>
<tr>
<td>i3en</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>16</td>
<td>1250 GB NVMe SSD</td>
</tr>
<tr>
<td>m5d</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>8</td>
<td>75 GiB NVMe SSD</td>
</tr>
<tr>
<td>m5ad</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>8</td>
<td>75 GiB NVMe SSD</td>
</tr>
<tr>
<td>m5ad</td>
<td>xlarge</td>
<td>x86_64</td>
<td>4</td>
<td>16</td>
<td>150 GiB NVMe SSD</td>
</tr>
<tr>
<td>m5ad</td>
<td>2xlarge</td>
<td>x86_64</td>
<td>8</td>
<td>32</td>
<td>300 GiB NVMe SSD</td>
</tr>
<tr>
<td>m5ad</td>
<td>8xlarge</td>
<td>x86_64</td>
<td>32</td>
<td>128</td>
<td>2 x 600 GiB NVMe SSD</td>
</tr>
<tr>
<td>m5ad</td>
<td>12xlarge</td>
<td>x86_64</td>
<td>48</td>
<td>192</td>
<td>2 x 900 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>medium</td>
<td>aarch64</td>
<td>1</td>
<td>4</td>
<td>EBS-Only (gp3 SSD)</td>
</tr>
<tr>
<td>m6gd</td>
<td>medium</td>
<td>aarch64</td>
<td>1</td>
<td>4</td>
<td>59 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>large</td>
<td>aarch64</td>
<td>2</td>
<td>8</td>
<td>118 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>xlarge</td>
<td>aarch64</td>
<td>4</td>
<td>16</td>
<td>237 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>2xlarge</td>
<td>aarch64</td>
<td>8</td>
<td>32</td>
<td>474 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>4xlarge</td>
<td>aarch64</td>
<td>16</td>
<td>64</td>
<td>950 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>8xlarge</td>
<td>aarch64</td>
<td>32</td>
<td>128</td>
<td>1900 GiB NVMe SSD</td>
</tr>
<tr>
<td>m6gd</td>
<td>12xlarge</td>
<td>aarch64</td>
<td>48</td>
<td>192</td>
<td>2 x 1425 GiB NVMe SSD</td>
</tr>
<tr>
<td>r5d</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>16</td>
<td>150 GiB NVMe SSD</td>
</tr>
<tr>
<td>r5ad</td>
<td>large</td>
<td>x86_64</td>
<td>2</td>
<td>16</td>
<td>75 GiB NVMe SSD</td>
</tr>
<tr>
<td>r6g</td>
<td>medium</td>
<td>aarch64</td>
<td>1</td>
<td>8</td>
<td>EBS-Only (gp3 SSD)</td>
</tr>
<tr>
<td>r6gd</td>
<td>medium</td>
<td>aarch64</td>
<td>1</td>
<td>8</td>
<td>59 GiB NVMe SSD</td>
</tr>
</tbody>
</table>

dependent variable, predicting the checkpointing speed instead of the elapsed time. Additionally, the application’s memory measurement was transformed to powers of two. In addition to those changes, the analysis limited the independent variables to disk speed and application memory. The number of CPU cores was omitted because it was highly correlated with the amount of memory in an instance and provided no significant prediction power, this is expected as AWS allocates an amount of memory
Table 7.2: Software Checkpoint Metrics Collected

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>EC2 instance type and size</td>
</tr>
<tr>
<td>architecture</td>
<td>Instance’s CPU architecture</td>
</tr>
<tr>
<td>vcpus</td>
<td>Number of CPU cores/threads</td>
</tr>
<tr>
<td>instance_memory</td>
<td>Amount of RAM (GiB) available to the instance.</td>
</tr>
<tr>
<td>storage_class</td>
<td>Determines if ephemeral or EBS storage was used.</td>
</tr>
<tr>
<td>fio_iops</td>
<td>Number of IOPS as measured by fio</td>
</tr>
<tr>
<td>fio_bw</td>
<td>Disk throughput (MiB/s) as measured by fio</td>
</tr>
<tr>
<td>dd_4k</td>
<td>Throughput (MiB/s) as measured by dd when the block size is 4KiB.</td>
</tr>
<tr>
<td>dd_128k</td>
<td>Throughput (MiB/s) as measured by dd when the block size is 128KiB.</td>
</tr>
<tr>
<td>app_memory</td>
<td>Amount of RAM (MiB) that the process consumes.</td>
</tr>
<tr>
<td>elapsed_time</td>
<td>Time it took to checkpoint the container.</td>
</tr>
</tbody>
</table>

per CPU core. Observations whose standardized residual was greater than three standard deviations were filtered out. It is worth mentioning that some conditions for a multiple linear regression model, such as residual homoscedasticity (as determined by the studentized residual by unstandardized predicted value scatterplot in Figure 7.3) and residual normality (as determined by a Normal Q-Q plot), were violated.

Because the multiple linear regression model violates some assumptions, a second model was prepared. This second model is not focused on the explicability of the model, but rather its predictive potential. As such, a random forest regressor was employed. The data was split into a training set and testing set. The training set was composed of 75% of the available data and the remaining 25% was used for testing. In this model, the mean squared error was used as the criterion for determining the quality of a split and allowed all nodes to extend to a maximum of ten levels. This ensemble model considers 100 trees in the forest. The mean squared error, explained variance, and $R^2$ scores were used to assess the regression’s performance. To address the constraints of impurity-based (i.e., Gini importance) feature importance for regression models on continuous and high cardinality data, the model’s features, and their importance (based on permutations on the whole model), were examined. See Figure 7.4.
7.5 Results and Discussion

The linear multiple regression model statistically significantly predicted the checkpoint speed, $F(2, 12731) = 100106.579$, $p < 0.0005$, adj $R^2 = 0.94$. The two variables considered in the model contributed statistically significantly to the prediction, $p < 0.0005$. The fitted regression model was:

$$elapsed\_time\_inv = 9.291 + (0.004 \times dd\_128k) - (0.857 \times app\_memory)$$
The second model, focused on predictive performance rather than explainability, is based on a random forest regressor. With an $R^2$ score of 0.99, the model explained 99% of the variance in the data. The feature importance graph shows that the application’s memory footprint is the most important predictor of checkpoint time (and thus checkpointing speed), followed by disk speed as assessed by `dd` using 128k-sized blocks. Other features, such as the number of CPU cores or the number of IOPS a disk can accomplish, are less important. This classifies the checkpointing process’s workload as one that is more concerned with disk bandwidth than with the disk’s ability to swiftly meet I/O requests. While other parameters, such as `fio_bw` and `fio_iops` could have been used in the model, the prediction performed well without them, and because `fio` is not a standard tool, the usage of `dd`, a typically pre-installed tool, was preferred.

The mean checkpoint duration for bins of 20 MiB/s disk speed was plotted to visualize the trend in the data dependent on disk speed and the size of an application’s RAM footprint. See Figure 7.5.

Both models agreed that the application’s memory footprint and disk speed are the most relevant predictors of checkpointing time. In other words, this C/R approach is dependent mostly on a process’ size in memory and the host’s disk throughput, with greater throughput resulting in faster checkpointing speeds. The overall trend indicates
Figure 7.5: Checkpoint Time vs. Disk Speed

that checkpoint times decrease as throughput increases and that rising application sizes result in longer checkpointing times at any fixed level of disk throughput.

This is significant because it enables resource planning and capacity analysis for edge clusters based on the loads that developers want to run and debug, while ensuring little (or negligible) downtime when using the approach presented in this chapter. When employing orchestrators to deploy resources, as is common nowadays, developers can alter deployments based on node affinity attributes to enhance debugging by leveraging the C/R techniques demonstrated in this work. Even when deployments cannot be changed, this information is useful for planned maintenance and even for scheduled game days (i.e., simulations of failures or events to test system reactions), when tests are performed in a safe environment that closely mirrors the production environment. Overall, the use of this strategy impacts the cost and time of development by reducing the window of iterative debugging and the time spent on efforts to reproduce the bug in development clusters, a major source of debugging difficulties at the edge, thereby accelerating bug discovery and root-cause analysis efforts without imposing any new debugging tool or practice.
7.6 Related Works

While the literature on debugging techniques is rich, the applicability of these tech-
niques to edge applications and service-oriented architectures (SOAs) has received com-
paratively less attention. Due to the similarity between microservices and conventional
distributed systems, this section draws from the greater body of literature to study de-
bugging strategies applicable to edge deployments.

Abrahamson et al. [184] presented a method for enhancing log data with a dis-
tributed timestamp to retain the relative sequence of events in Java programs. Coupled
with ShiViz, their implementation, ShiVector, provides a graphical representation of
the system’s behavior. Beschastnikh et al. [185] utilized ShiVector and ShiViz to en-
hance logs at the nodes of distributed systems and repurposed the tool for distributed
data-store systems. Zhou et al. [163] expanded upon ShiViz to visualize microservices
more efficiently by focusing on a service-level analysis and made the case for instru-
menting the relationship between microservices via routinely deployed service meshes
(e.g. Istio [186], Consul) to enable distributed tracing. Distributed tracing is not
always appropriate, as they noted.

Other applications will require instrumenting the source code (binary or bytecode)
to collect and record data about their functionality. While applications that primar-
ily rely on REST APIs for communication can be easily traced through the requests
issued and consumed by the system, and thus tracked via a service mesh or other
L7 implementations, other applications will require instrumenting the source code (bi-
nary or bytecode). This is the case with industrial microservices, which are frequently
quite heterogeneous and may employ REST APIs, message queues, RPCs, and socket
connectivity, among others.

In certain instances, tracing can be detrimental due to metric overload, runtime
or throughput overhead, and the storage requirements for the traces, which are often maintained locally on the node. These expenses may place a strain on the resources at the edge. Aumayr et al. [187] saw a 10% decrease in runtime performance and a 1% increase in latency for traced web services. Similarly, Zhu et al. [188] reported a 14% performance detriment with CAOPLE-implemented microservices. Tanno and Iwasaki [189] attempted a similar strategy with a runtime overhead of up to 37 times the baseline. Perhaps even more significant is the memory consumption cost that certain optimizations, such as those recommended by Schulz and Bockisch [190], introduce when executing applications via BITE. While BITE’s approach of instrumenting an application with traces that allow a re-run is suitable for hosts with no fixed constraints, containers are typically scheduled based on their resource requirements; and in edge environments, an application may not be able to tolerate a 50x increase in their baseline memory consumption. Other instrumentation-based techniques, such as those investigated by Guo et al. [191] and Buhse et al. [192], are implemented at runtime by substituting dynamically linked libraries. Complex methods utilize hardware performance counters and are architecture-dependent (e.g., Intel’s Processor Trace, ARM’s Embedded Trace Macrocell, etc.) [193][194]. Such approaches are important for debugging microservice-based systems that interface with monolith (and hybrid) applications, but are outside the focus of this chapter. Other well-known open-source tracing programs include Zipkin, Jaeger, Magpie [195], Pinpoint [196], SkyWalking [197], and Haystack [198]. Notably, the integration of Jaeger, Istio, and Kubernetes has become the industry standard for microservices tracing and fault localization [199].

Some of the aforementioned developments in tracing serve as the basis for more advanced approaches such as record-replay (RR). Record-replay permits repeatable executions of an application by capturing the execution trace for subsequent replay. Debugging is perhaps the most important application of record-replay since it mini-
mizes the time required to locate errors. Traditional iterative or cyclical debugging, on the other hand, is time-consuming and does not ensure bug reproduction. As a result, record-replay techniques have been implemented in other fields, including hardware verification [200], OS kernel debugging [201], and software testing. The implementations of record-replay are frequently hampered by large log sizes (to record all execution activity), high runtime overhead (due to recording), replay slowdown, complex implementations that make a generic solution impractical, and intrusive mechanisms that can lead to code instrumentation and the accidental introduction of “heisenbugs”.

Typically, record-replay is achieved using system call interception, binary (or bytecode) analysis and instrumentation [191][192][202], the usage of custom kernel modules [203], and logging. The environment’s resource constraints make recording prohibitively expensive and nearly impossible at the edge with solutions that require specific hardware support due to the heterogeneity of deployable nodes.

A typical optimization is the use of checkpoints, or periodic snapshots of an application’s state, to reduce some of the space overhead by removing logs older than the most recent checkpoint if they are no longer required. Checkpoints serve multiple purposes, including resuming execution at separate places in an application’s chronology without having to re-execute the preceding events. Vilk et al. [204] utilized checkpoints in web applications to navigate to a specified point in the logic and visual state of the application. Through shareable checkpoints, Thompson et al. [205] enabled collaborative audio remixing in web audio applications. This C/R-assisted approach to software debugging is commonly referred to as Time-Travel Debugging (TTD). C/R has also been utilized to facilitate software testing by facilitating fuzzing [201] and automating unit test extraction [206]. Marra et al. [207] utilized C/R approaches to reduce fault localization times, perform live code modifications, and minimize re-deployments for Spark [208] applications in distributed contexts. During unit testing, Sudsee and
Kaewkasi [209] employed C/R to minimize recalculations in Spark jobs.

The choice of C/R engine influences the scope of the target domain and the functionality offered. Grossman and Sarkar [210] created CHIME to provide a fundamental framework for debugging and profiling multithreaded applications. Hursey et al. [211] examined the usage of C/R methods and MPI implementation enhancements for debugging multithreaded MPI applications in HPC environments. They utilized the Berkeley Lab CheckpointRestart (BLCR) service to record the current state of an application. Similar to the approach presented in this chapter, debugging sessions are restarted at known intermediate phases in their work, hence lowering the overall debugging time. They chose BLCR for their C/R engine over DMTCP due to the faster checkpointing speed. CHIME uses a compiler-based method to include checkpoint directives into the application, making it more portable and independent of platform. The work presented in this chapter features the C/R engine CRIU, which is the most used engine for containerized processes. Instead of preloading particular libraries such as BLCR and DMTCP, it employs any standard Linux kernel after version 3.11 [212]. Checkpointing with CRIU is transparent to current programs since, unlike CHIME, it does not require recompilation. In contrast to DMTCP and BLCR, it does not need to notify processes or register custom signal handlers when a checkpoint is imminent, hence keeping the behavior of the C/R-ed program. Unlike DMTCP, CHIME, and BLCR, CRIU is compatible with container technology since it supports namespaces. DMTCP’s only benefit over CRIU and BLCR is that it supports debugging C/R applications. Due to the use of the ptrace API by CRIU, debuggers (and by extension, applications under the control of a debugger) cannot be checkpointed, necessitating the approach presented in this study.

C/R is not considered a “true” record-replay solution like Mozilla’s rr [213] because it is used to narrow the debugging session as opposed to capturing the complete state leading up to the checkpoint [212]. The implementation described in this chapter is not
intended to replace rr. Even when the recording is configured to capture fluctuations in “dirty” memory regions, record-replay and reverse debugging may result in significant overheads or hefty checkpoints. In Intel microarchitectures introduced after Nehalem (2010), system call filtering and hardware performance counters enable rr’s approach. Due to the design’s simplicity (i.e., avoidance of complex dynamic binary instrumentation), rr can add up to 20% overhead to debugged workloads. Since rr emulates a single-core machine, multithreaded applications incur additional overhead. The use of hardware performance counters in rr hinders its adoption at the edge, where microarchitectures vary. Even in the cloud, rr is inadequate due to its incomplete support for alternative x86-64 microarchitectures such as AMD’s Zen (and its revisions) [214]. In multi-tenancy cloud platforms, the lack of virtualized hardware performance counters inhibits rr’s adoption. Moreover, minor alterations to microarchitectures have been known to produce errors and impede rr’s capacity to replay [215]. Mozilla’s rr is compatible with gdb and the languages supported by gdb, but other record-replay systems are language-specific [190][192], restricting their applicability in a microservice-based solution where services may be developed in a variety of languages.

Non-recording options, such as Parikshan [216] and DeLorean [203], are more comparable to this chapter’s method. DeLorean is a time-traveling solution that performs periodic checks of an application’s important events. DeLorean does not require a custom kernel, but it utilizes a custom kernel module, hardware checkpoints, and the LD_PRELOAD interface to preload a library. In contrast to the technique presented in this chapter, DeLorean does not give online debugging capabilities (i.e., the ability to enter and leave debugging sessions as needed), but instead requires executing the program inside the framework from the beginning, and only through gdb. Due to its exclusive usage of in-memory checkpointing, their method is not ideal for debugging already-deployed programs or applications with a big memory footprint. This can also
cause issues at the edge, where memory is frequently constrained. Parikshan creates replica debug containers using live cloning and anticipates that problems may be precisely recreated by delivering identical production network traffic to the debug containers. As with the chapter’s proposed solution, they allow the developer to use their preferred debugging tools to identify and resolve the issue’s source. Parikshan, unlike the suggested method, uses Virtuozzo’s OpenVZ kernel to support ptrace API-based checkpointing programs. OpenVZ has never received considerable Linux community support, and only a small fraction of their work has been incorporated into the mainline Linux kernel. This is problematic for developers and system administrators who rely on specific Linux kernel functionalities that OpenVZ has not adopted. Unlike OpenVZ, the Linux kernel allows unprivileged users to use containers (e.g., rootless containers) and enforces additional isolation via SELinux.

Using delta debugging techniques to find buggy scenarios by repeatedly executing microservices under varying conditions is another unorthodox approach to microservice debugging. These approaches resemble the iterative nature of debugging, and while they can aid in identifying the root cause by automating the process, they incur the expense of constantly starting a process and advancing towards the desired state. Although modifications have been suggested, such as the use of parallel executions to address all possible combinations of outcomes simultaneously, these are still time-consuming and disregard the restricted resources at the edge, especially the energy impact on battery-powered devices. The problem is compounded when examining interactions across microservices, as this necessitates the usage of the delta debugging technique for each microservice involved in the interaction. Zhou et al. [217] acknowledged this constraint by focusing on what they refer to as the “simplest setting” in microservice environments: one microservice node and one microservice instance. Their proposed method [163] incorporates an interaction between a microservice or-
chestrator (i.e., Kubernetes) and its service mesh (i.e., Istio). Running parallel copies of the target microservice, at times more than 60 replicas, enabled them to identify a failing scenario in $18 - 46$ minutes on average [217]. This may be acceptable in cloud environments, but at the edge, a 60x overhead on the application’s resource use is unsustainable. Although not addressed in their study, even a methodical replica rollout strategy, such as the one characteristic of Kubernetes’ Deployment objects, would be inadmissible due to the significant debugging time overhead. In addition, unlike cloud environments, the automatic provisioning of nodes for replica scheduling, upon which their optimization depends, may not be available at the edge.

As noted previously, standard strategies for debugging microservices-based applications include distributed tracing, log analysis, and visualization. These tactics, along with other unconventional approaches, complement the work presented in this chapter because they may be implemented in the recreated environment.
Chapter 8

Implications and Conclusion

The introduction of cloud computing enabled the leasing of elastic network resources. With this new computing model, software design migrated away from conventional monolithic systems. Newer designs advocated service-oriented architectures and emphasized microservices as scalable software building components. In contrast to software monoliths, scaling microservice-based applications requires the deployment of replicas of the bottlenecked module as opposed to a whole copy of the system. As a result, microservices complemented the scalability of the cloud, made more efficient use of leased resources, and became the standard for deployment.

As applications evolve and QoS requirements become more rigorous, the cloud has proven inadequate. Under the cloud model, data is transferred to and from the network core and processed in centralized datacenters. The new generation of applications requires near-real time responsiveness and, at times, geographic awareness. The cloud is not favorable to these types of loads. To circumvent these constraints, the fog and the edge were conceived as extensions to the cloud. Simply said, by relocating the application’s required resources closer to the end user, these extensions can ensure low latency and geographic awareness. Real-time systems, the Industrial Internet of
Things, and even online gaming applications can benefit from these cloud extensions. Among the more ambitious applications include telesurgery and augmented reality.

However, the expansion to the edge and the fog is not without challenges. Just as resources must be managed in the cloud, orchestration is likewise required in the fog and edge layers. At these layers, resource management is not a question of cost reduction, but rather of service survivability. Inappropriate resource management degrades QoS guarantees and impacts service delivery. Since the fog and edge layers are resource constrained, service degradation cannot be fully countered with the additional leasing of resources. Consequently, tactics that optimize the utilization of resources in these layers are the subject of extensive research.

As part of the ongoing effort to manage resources at the fog and the edge, cloud-native solutions are being evaluated for their suitability in these contexts. This document examined the overhead incurred by adopting different levels of orchestration services that are popular in the cloud, at the fog and edge layers. More specifically, this document explored the overhead in the areas of networking, compute, the management of time and its perception, as well as the impact on software maintenance.

Regarding network overhead, Chapter 4 concentrated on the impact on throughput and latency across multiple orchestration service levels. Due to the ongoing chatter required to coordinate the nodes in a cluster, the usage of an orchestrator had some foreseeable impact; nevertheless, for the majority of analyzed metrics, this overhead was not statistically significant. However, after the orchestrator mediated communication over a service mesh, the application’s observed throughput and latency decreased substantially.

Along with network metrics, CPU utilization metrics were also collected and further analyzed in Chapter 5. Using a service mesh configuration resulted in significant compute overhead. Half of the observed overhead can be attributed to the Linux net-
working internals that enable data forwarding, while the remainder can be credited to the sidecar proxy and network mesh communication encryption. Significant overhead is a danger to service delivery since it consumes resources that could be used to fulfill service requests. Options that account for this overhead during the placement phase of the scheduler have been found to reduce the overhead and maintain service level without requiring more hardware resources. Current research focuses on the development of scheduler plugins that consider network awareness and memory subsystems to make the best placement determination for latency and throughput sensitive loads, as well as de-scheduling rules that can evict loads from a node and place them in another if network conditions deteriorate or if a better QoS can be experienced in another host.

Given that microservices are ephemeral and can be evicted and restarted on another host, migration has been studied widely. Existing hurdles to this process include the loss of time perception that occurs during relocation. When this occurs, services are unable to continue serving requests, and mission delivery is negatively impacted. Through the newly introduced time namespace, Chapter 6 presented a procedure to preserve a service’s time perception even after migration to a new host. Due to the novelty of the time namespace, there is a paucity of research on its added overhead. Chapter 6 aims to fill the void by testing different time-related system calls under multiple degrees of virtualization, including the ones used by orchestrators. In keeping with the overhead of a fully-namespaced container, time virtualization introduces a 2–4% overhead relative to a no-virtualization baseline. Therefore, orchestrators are well-positioned to incorporate time management into their responsibilities.

Device heterogeneity is an additional source of problems. In contrast to the cloud, whose nodes are predominantly homogeneous, the fog and the edge host devices with varying ISAs, network interfaces, processing power, degree of mobility, etc. As a result, the administration of these devices necessitates automated orchestration to accomplish
load placement based on the capacity of the host. Unfortunately, the diversity makes fault localization and debugging more difficult due to the inherent challenge of duplicating and locating problematic configurations. In addition, as the number of microservices and their interactions grow, it becomes exponentially more difficult to identify errors and deliver patches. Chapter 7 presents a technique to assist the reconstruction of buggy environments by recording the state of a whole application and its dependent services in order to recreate precise copies that may be utilized as a debugging aid. However, because this procedure needs a temporary interruption of service, it may have a negative influence on service delivery. As a result, Chapter 7 quantified the service outage duration caused by this technique when applied to services of varying resource usage. It was observed that the procedure relies strongly on the storage performance of the host and the memory footprint of the program being snapshotted. However, for the majority of services, this is a quick procedure that causes minimal service interruption. Several optimizations have been investigated to shorten the blackout period, with some focusing on incremental checkpointing and others on the usage of various storage subsystems to improve the dumping speed of a service’s state, and therefore lower the blackout period.

In conclusion, the fog and the edge, conceived as extensions of the cloud, provide cloud native orchestration technologies with a new set of issues. In addition to the fact that these new layers require coordinating more aspects than those already existent in the cloud, the orchestration itself adds resource overhead to an already resource-constrained environment. This document aimed to estimate the overhead introduced by numerous orchestration services and workflows that are frequently implemented in the cloud and to classify the impact in terms of network, compute, time and its perception, as well as software maintenance. Brokering communications through a service mesh had the greatest influence on the throughput experienced by an application and increased
CPU usage, hence decreasing the overall compute capability of a node. Due in part to the strategies given in this document for time management and software maintenance, it was possible to enable additional orchestration use cases while imposing minor overhead in the majority of circumstances.
Bibliography


119


[145] Red Hat, Inc. usleep(3) and nanosleep(2) have better granularity with CFS scheduler in RHEL6. https://access.redhat.com/solutions/186663, 2015.


