Performance Analysis of Zero Trust in Cloud Native Systems

Simone Rodigari

Department of Computer Science, Munster Technological University, Cork, Ireland

Follow this and additional works at: https://sword.cit.ie/allthe

Part of the Computer Sciences Commons

Recommended Citation

This Master Thesis is brought to you for free and open access by the Dissertations and Theses at SWORD - South West Open Research Deposit. It has been accepted for inclusion in Theses by an authorized administrator of SWORD - South West Open Research Deposit. For more information, please contact sword@cit.ie.
Performance Analysis of Zero Trust in Cloud Native Systems

by

Simone Rodigari

This thesis has been submitted in partial fulfillment for the degree of Master of Science in Research (Cloud Computing)

in the

Faculty of Engineering and Science
Department of Computer Science

4th of January 2023
Declaration of Authorship

This report, Performance Analysis of Zero Trust in Cloud Native Systems, is submitted in partial fulfillment of the requirements of Master of Science in Research (Cloud Computing) at Munster Technological University Cork. I, Simone Rodigari, declare that this thesis titled, Performance Analysis of Zero Trust in Cloud Native Systems and the work represents substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged. I confirm that:

■ This work was done wholly or mainly while in candidature Master of Science in Research (Cloud Computing) at Munster Technological University Cork.

■ Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University Cork or any other institution, this has been clearly stated.

■ Where I have consulted the published work of others, this is always clearly attributed.

■ Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.

■ I have acknowledged all main sources of help.

■ Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 04/01/2023

04/01/2023
Abstract

Faculty of Engineering and Science
Department of Computer Science

Master of Science

by Simone Rodigari

Critical applications demand strong security implementations, low latency and high availability at constant rates, however, the performance of a software system is affected by the implementation of security. This research measures the performance overhead and possible mitigation in cloud native systems secured with a service mesh, which allows enabling security policies for the authentication, authorization and encryption of traffic within distributed systems. The side-car proxy is a core component of this architecture, acting as a policy enforcement point and intercepting networking communication from/to applications part of the mesh, consequently affecting the performance of applications hosted in the cloud. Physical resources are required to operate the control plane and data plane, while latency is affected by the enforcement of security policies and encryption. We configured a cloud environment consisting of a managed Kubernetes cluster, deployed a cloud native synthetic application, configured service mesh, tested the performance under load and analyzed results to establish the overhead in terms of latency, CPU and memory. The analysis is performed on both data and control plane. Additionally, a performance enhancement was explored with the use of extended Berkeley Packet Filter technology which operates at the Linux Kernel level. Results show a reduction in CPU consumption as well as latency.
Acknowledgements

I would like to thank Dr Sean McSweeney, Prof Donna O’Shea and Pat McCarthy for the guidance and assistance during the course of study. I have learned a lot from all of you, and far beyond the academic context. Postgraduate research is a ”character building experience”, as Sean would say, I could not agree more. I am a better researcher, engineer and person thanks to you all.

I would like to thank Donna for the help with the research scope definition and the paper publication. Because of you I have greater focus on security implementations. This pushed my career to the next level. It has been a wonderful experience working together. Pat, you have been a rock over the years, both during my undergraduate and postgraduate degree. Your resilience is exemplary, as well as your interpersonal skills, you have helped me overcome the hardest hurdles when I was on the verge to quit. Sean, I have always enjoyed our discussions and the lessons I have learned from you are numerous. An important one is the ”acceptance of abstraction layers” one fundamental pillar of computer science, which is a future-proof concept that enables modularized systems, collaboration and scalability.

I would also like to thank Martin McCarry. I would not be where I am without you. You introduced me to DevOps and exposed me to automation, virtualization, containerization, and RestAPI in an enterprise environment. With the substantial experience in the industry and the approach you have, you thought me how to tackle complex problems with a focus on value. I still advocate the standards you thought me for Git commits, and colleagues thank me for that. Thank you, Martin. Also, the coffee machine you gifted me kept me going over the past 3 years, that was also a major help.

Moreover, I would like to thank my brother Stefano, Jennifer, Max, Richard, Claudia, Ryan, Shay, Luca, Fabio, William, Giovanni, Davide, Paul, Micheal, Luca D.N., and Arjen. All of you stood by me in different times / circumstances and provided support / guidance when needed. I have learned life-skills from you and I will always be grateful for this.

Finally, I want to thank Pilz, Innowatts, and TrendMicro for the support and expertise provided since 2019. I am now working for IBM Cloud, and I would not be where I am without the experience and knowledge I had the opportunity to acquire over the past years of professional career.
## Contents

Declaration of Authorship .......................... i

Abstract ........................................... ii

Acknowledgements ................................... iii

List of Figures ..................................... vii

List of Tables ...................................... x

Abbreviations ....................................... xii

1 Introduction ....................................... 1
   1.1 Motivation ...................................... 5
   1.2 Contribution .................................... 9
   1.3 Structure of This Document ................... 9

2 Literature Review ................................ 10
   2.1 Cloud Native .................................... 11
   2.2 Virtualization and Containers ................ 11
      2.2.1 Hardware Virtualization .................. 12
      2.2.2 Operating System Virtualization ......... 14
      2.2.3 Container internals and Technologies .... 15
   2.3 Microservices ................................... 17
   2.4 Container Orchestration ....................... 19
      2.4.1 Kubernetes networking ..................... 21
   2.5 Security and Zero Trust ....................... 23
   2.6 Sidecar Proxy ................................... 25
      2.6.1 Service Mesh ................................ 26
      2.6.2 Istio Service mesh and Security implementation 28
   2.7 Extended Berkley Packet Filtering (eBPF) .... 29
      2.7.1 eBPF Vs iptables ........................... 30

3 Zero Trust Performance in Cloud Native Systems .... 33
   3.1 Project evolution ................................ 33
List of Figures

1.1 Flexera 2022: Type of Clouds used ............................................. 2
1.2 Architecture of the cloud scheme ................................................. 3
1.3 High level architecture of DMZ network ....................................... 6
1.4 Flexera 2022: Container Tools used ............................................. 7

2.1 Cloud native trailmap ................................................................. 12
2.2 Use of CPU for Apps in a VM ..................................................... 13
2.3 VMs VS Containers ................................................................. 14
2.4 Linux network namespaces ....................................................... 15
2.5 Docker architecture ................................................................. 17
2.6 Container cluster architectures .................................................. 20
2.7 Kubernetes components ........................................................... 20
2.8 Linux network namespaces and pods ......................................... 21
2.9 Kubernetes Services and labels .................................................. 22
2.10 Pod to pod networking ............................................................. 23
2.11 Device Agent/Gateway Model .................................................. 24
2.12 Sidecar and application containers .......................................... 26
2.13 Service Mesh Architecture ....................................................... 27
2.14 Istio Security Architecture ....................................................... 28
2.15 ebpf loader and verification architecture ................................... 30
List of Figures

2.16 Iptables chain processing and hook points ........................................... 31
2.17 Sidecar proxy networking in a Service Mesh ........................................... 31
2.18 Sidecar proxy networking in a Service Mesh with eBPF ............................ 32
2.19 Latency measurements ........................................................................... 32

3.1 Initial project architecture ....................................................................... 34
3.2 Architecture of Zero Trust multi-cloud ..................................................... 34
3.3 Google cloud architecture ..................................................................... 40
3.4 Pulumi architecture ............................................................................. 41
3.5 Pulumi and GCP interactions ................................................................. 42
3.6 Istio data and control plane .................................................................. 43
3.7 Grafana dashboard ............................................................................... 44
3.8 Load test sequence diagram .................................................................. 46

4.1 IaC flow ............................................................................................... 51
4.2 Test VM main functions .................................................................... 58
4.3 Test results directory structure ............................................................ 64

5.1 Latency analysis by percentiles ............................................................. 74
5.2 Latency results per test run .................................................................. 75
5.3 CPU test run plot ............................................................................. 76
5.4 Memory test run plot (kube-proxy) ........................................................ 77
5.5 Memory test run plot (eBPF) ................................................................. 78
5.6 CPU test run plot - after cleaning ........................................................ 79
5.7 Overall latency overhead for kube-proxy networking ............................ 88
5.8 Overall latency overhead for eBPF networking ........................................ 89
5.9 Comparison overall Latency overhead for eBPF networking .................... 90
5.10 Comparison overall Latency overhead for kube-proxy networking ............ 90
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.11</td>
<td>Overall CPU overhead for kube-proxy networking</td>
<td>91</td>
</tr>
<tr>
<td>5.12</td>
<td>Overall CPU overhead for eBPF networking</td>
<td>92</td>
</tr>
<tr>
<td>5.13</td>
<td>Comparison overall CPU overhead for eBPF networking</td>
<td>92</td>
</tr>
<tr>
<td>5.14</td>
<td>Comparison overall CPU overhead for kube-proxy networking</td>
<td>93</td>
</tr>
<tr>
<td>5.15</td>
<td>Overall Memory overhead for kube-proxy networking</td>
<td>94</td>
</tr>
<tr>
<td>5.16</td>
<td>Overall Memory overhead for eBPF networking</td>
<td>94</td>
</tr>
<tr>
<td>5.17</td>
<td>Comparison overall Memory overhead for eBPF networking</td>
<td>95</td>
</tr>
<tr>
<td>5.18</td>
<td>Comparison overall Memory overhead for kube-proxy networking</td>
<td>95</td>
</tr>
<tr>
<td>5.19</td>
<td>Overall Latency overhead</td>
<td>98</td>
</tr>
<tr>
<td>5.20</td>
<td>Comparison overall Latency overhead</td>
<td>99</td>
</tr>
<tr>
<td>5.21</td>
<td>Overall CPU overhead</td>
<td>103</td>
</tr>
<tr>
<td>5.22</td>
<td>Comparison overall CPU overhead</td>
<td>104</td>
</tr>
<tr>
<td>5.23</td>
<td>Overall Memory overhead</td>
<td>107</td>
</tr>
<tr>
<td>5.24</td>
<td>Comparison overall Memory overhead</td>
<td>108</td>
</tr>
<tr>
<td>A.1</td>
<td>CPU plots by system configuration at 1400 qps</td>
<td>123</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Container Solutions comparison ........................................... 17
2.2 Microservices advantages .................................................... 18
3.1 Cluster Specification .......................................................... 42
3.2 Cluster Specification .......................................................... 43
3.3 Load generator tools comparison ............................................ 45
4.1 Technology stack ............................................................... 49
5.1 Primary cluster configuration ............................................... 72
5.2 Test configuration parameters .............................................. 73
5.3 Latency mean and standard deviation for Kube-Proxy clusters ....... 80
5.4 Latency mean and standard deviation for eBPF clusters ............... 81
5.5 CPU results - kube-proxy networking Service Mesh no Policies .... 82
5.6 CPU results - eBPF networking Service Mesh no Policies .......... 82
5.7 CPU results - kube-proxy networking Service Mesh Zero Trust .... 83
5.8 CPU results - eBPF networking Service Mesh Zero Trust .......... 83
5.9 CPU results - kube-proxy networking (no service mesh) .......... 83
5.10 CPU results - eBPF networking (no service mesh) .................. 84
5.11 Memory results - kube-proxy networking Service Mesh no Policies 84
5.12 Memory results - eBPF networking Service Mesh no Policies ...... 85
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.13</td>
<td>Memory results - kube-proxy networking Service Mesh Zero Trust</td>
<td>85</td>
</tr>
<tr>
<td>5.14</td>
<td>Memory test results - eBPF networking Service Mesh Zero Trust</td>
<td>86</td>
</tr>
<tr>
<td>5.15</td>
<td>Memory test results - kube-proxy networking (no service mesh)</td>
<td>86</td>
</tr>
<tr>
<td>5.16</td>
<td>Memory results - eBPF networking (no service mesh)</td>
<td>86</td>
</tr>
<tr>
<td>5.17</td>
<td>Latency mean and standard deviation for each test run on Kube-Proxy clusters</td>
<td>97</td>
</tr>
<tr>
<td>5.18</td>
<td>Latency mean and standard deviation for each test run on eBPF clusters</td>
<td>97</td>
</tr>
<tr>
<td>5.19</td>
<td>CPU results - kube-proxy networking Service Mesh no Policies</td>
<td>100</td>
</tr>
<tr>
<td>5.20</td>
<td>CPU results - eBPF networking Service Mesh no Policies</td>
<td>100</td>
</tr>
<tr>
<td>5.21</td>
<td>CPU results - kube-proxy networking Service Mesh Zero Trust</td>
<td>101</td>
</tr>
<tr>
<td>5.22</td>
<td>CPU results - eBPF networking Service Mesh Zero Trust</td>
<td>101</td>
</tr>
<tr>
<td>5.23</td>
<td>CPU results - kube-proxy networking (no service mesh)</td>
<td>102</td>
</tr>
<tr>
<td>5.24</td>
<td>CPU results - eBPF networking (no service mesh)</td>
<td>102</td>
</tr>
<tr>
<td>5.25</td>
<td>Memory results - kube-proxy networking Service Mesh no Policies</td>
<td>105</td>
</tr>
<tr>
<td>5.26</td>
<td>Memory results - eBPF networking Service Mesh no Policies</td>
<td>105</td>
</tr>
<tr>
<td>5.27</td>
<td>Memory results - kube-proxy networking Service Mesh Zero Trust</td>
<td>105</td>
</tr>
<tr>
<td>5.28</td>
<td>Memory test results - eBPF networking Service Mesh Zero Trust</td>
<td>106</td>
</tr>
<tr>
<td>5.29</td>
<td>Memory test results - kube-proxy networking (no service mesh)</td>
<td>106</td>
</tr>
<tr>
<td>5.30</td>
<td>Memory results - eBPF networking (no service mesh)</td>
<td>107</td>
</tr>
</tbody>
</table>
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>PCP</td>
<td>Public Cloud Provider</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>DMZ</td>
<td>De-Militarized Zone</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>CNCF</td>
<td>Cloud Native Computing Foundation</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Manager</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick EMUlator</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>LXC</td>
<td>LinuX Container</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol address</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>VPC</td>
<td>Virtual Private Cloud</td>
</tr>
<tr>
<td>CIDR</td>
<td>Classless Inter-Domain Routing</td>
</tr>
<tr>
<td>CNI</td>
<td>Container Network Interface</td>
</tr>
<tr>
<td>ZTA</td>
<td>Zero Trust Architecture</td>
</tr>
<tr>
<td>SDP</td>
<td>Software Defined Perimeter</td>
</tr>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
</tr>
<tr>
<td>PA</td>
<td>Policy Administrator</td>
</tr>
<tr>
<td>PE</td>
<td>Policy Engine</td>
</tr>
<tr>
<td>PEP</td>
<td>Policy Enforcement Point</td>
</tr>
<tr>
<td>CSA</td>
<td>Cloud Security Alliance</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JWT</td>
<td>JSON Web Token</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>eBPF</td>
<td>extended Berkeley Packet Filtering</td>
</tr>
<tr>
<td>XDP</td>
<td>eXpress Data Path</td>
</tr>
<tr>
<td>TC</td>
<td>Traffic Control</td>
</tr>
<tr>
<td>IaC</td>
<td>Infrastructure as Code</td>
</tr>
<tr>
<td>GitOps</td>
<td>Git Operations</td>
</tr>
<tr>
<td>DevOps</td>
<td>Development Operations</td>
</tr>
<tr>
<td>mTLS</td>
<td>mutual Transport Layer Security</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>GCP</td>
<td>Google Cloud Platform</td>
</tr>
<tr>
<td>EKS</td>
<td>Elastic Kubernetes Service</td>
</tr>
<tr>
<td>GKE</td>
<td>Google Kubernetes Engine</td>
</tr>
<tr>
<td>KPIs</td>
<td>Key Performance Indicators</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role-Based Access Control</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma-Separated Values</td>
</tr>
<tr>
<td>QPS</td>
<td>Query Per Second</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CI/CD</td>
<td>Continuous Integration / Continuous Delivery (or Deployment)</td>
</tr>
<tr>
<td>gRPC</td>
<td>google Remote Procedure Call</td>
</tr>
<tr>
<td>RSS</td>
<td>Resident Set Size</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>HPO</td>
<td>Horizontal-Pod Autoscaler</td>
</tr>
</tbody>
</table>
To my son Julian, you are my strength. Never stop being curious, humble, and a learner. Keep challenging yourself and try to make the world a bit better. I love you.
Chapter 1

Introduction

The National Institute of Standards and Technology (NIST) defines Cloud computing [1] as a ".. model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Modern internet applications are made accessible via standard browsers to be consumed by end users directly on any devices, including mobile appliances, tablets, laptops or desktop stations. The general availability of the internet and the increased accessibility and affordability of internet devices allows large pools of users to access and consume internet applications simultaneously. The traditional data-center approach for hosting servers and network infrastructures is no longer the de-facto standard due to the increased demand for the ability to scale, reduced response time, enhanced security and faster delivery of new features. Over the past decade the hybrid-cloud model provided the ability to integrate the data-center approach with public cloud solutions, with the ability to leverage the on-demand solutions available in the public cloud space while maintaining critical workloads operations on private data centers. The requirements of flexibility, scalability and geographical availability, further increased in more recent years due to the shift in applications development architectures and models, where Software as a Service (SaaS) resulted to be an efficient delivery model for numerous use cases, virtually in any industries. To satisfy the SaaS model requirements and provide additional benefits such as pay-per use services and infrastructure, organizations can leverage Public Cloud Providers (PCPs) to design and create the infrastructure architecture for scalable applications delivered to end users via the public internet. Figure 1.1 describes the types of clouds used according to the results of Flexera "State of the Cloud Report 2022" [2]. The report presents survey results from 753 respondents, which
are global cloud decision makers and users. Figure 1.1 shows that all organizations are using at least one private or public cloud, and 96 percent of respondents utilize at least one public cloud.

![Figure 1.1: Flexera 2022: Type of Clouds used](image)

In the transition to public cloud, organizations, rely on public cloud providers for hosting Information Technology (IT) infrastructures with on-demand services, an unrivalled level of security, large ability to scale and extensive geographical availability, which would require an enormous investment to replicate with private data-centers, where skilled staff are responsible to configure, maintain and secure proprietary servers on diverse geographical regions.

A prevalent use case for organizations to adopt the Cloud Computing paradigm is to deliver performant and scalable applications developed using a cloud native microservice architecture to decouple the application logic into several smaller specialized services. PCPs offer several on-demand IT services, including compute, networking and storage, that allow for the creation of necessary infrastructure layers and runtime environments for microservices deployments.

Figure 1.2 describes four main components of the Cloud scheme: Services (first block from the top), Application, Platform and Storage. Safanov V. et al [4] describes the cloud scheme components as follows: Services are available via the cloud, Infrastructure for services deployment and use, Platform as a set of tools for using the cloud, and Storage to support storing the users’ data in the data-center(s) implementing the cloud.

The adoption of cloud and particularly public cloud services introduces a number of new challenges that organizations need to face as a consequence of paradigm shift and different model adoption. Cloud native applications are also designed differently than
traditional application where the software code was often built on a single block using a monolith approach with software components strongly coupled, weakly cohesive and few external dependencies. The cloud native approach entails the separation of responsibilities between software components where each service is highly specialized, responsible for a well-defined task, hence highly cohesive. This approach is implemented by the microservices architectures where smaller components are highly coupled with other components to deliver application features and functionalities while maintaining high cohesion within the components scope. One of the biggest challenges with cloud native microservices architectures is the organization and coordination of communications between system services, which also introduces the additional requirement of security implementation since operations that were traditionally executed within the same code-base (monolith), are now executed externally, in a separate location, sometimes on a different host machine. Some of the most important challenges that organizations need to face when transitioning to the cloud include:

- Define and implement the transition to Cloud process
- Vendor lock-in
- Security in shared environments
- Performance
Software applications demand strong security implementations due to the increased risk of cybersecurity attacks and requirements to comply with standards and regulations while maintaining low-latency and high availability. The implementation of security in a PCP environment differs from the implementation of security in a proprietary data-center environment, where typically a firewall was configured at the perimeter of the internal network, and communication within the firewall boundary are deemed as trusted. In a shared cloud environment where applications and infrastructure environments are multi-tenanted, the perimeter approach is not applicable as users operating within the network boundaries of the same system cannot be deemed as trusted, as often those entities are unrelated. According to D. Klein [5], “in today’s hybrid, multi-cloud datacenters, the perimeter as such is no longer relevant. And the threats are no longer outside, but within.”

In recent years a number of approaches, models, and solutions for distributed systems’ security have arisen. Examples include: security in depth to provide multiple layers of security, the least privilege principle to grant only the minimum necessary permission to users/machine in a system, and zero trust which by default does not trust any entities within a system. In a distributed system architecture, particularly in a shared cloud environment, all those security measures are necessary. This research will focus on zero trust security in cloud native systems.

Zero Trust is a relatively new model to secure modern systems by Authenticating, Authorizing and Encrypting all communication between applications and users. The sidecar proxy is an application design pattern which can be implemented as an abstraction layer to enable networking, monitoring and policy enforcement while keeping the logic of the main architecture unchanged. Sidecar proxy can enable Zero Trust by enforcing security policies while intercepting all network requests going to and coming from microservices. In academic research, the performance of zero trust configuration implemented with a sidecar proxy has yet to be analyzed, particularly in a cloud native system. This research focuses on quantifying the performance overhead added by the implementation of a Zero Trust security model in a system infrastructure deployed on PCPs. The assumption is that the performance of a cloud-based software system is affected by the implementation of security due to the additional processing time and physical resources necessary to enable security, in order to validate this assumption, the system performance should be evaluated to quantify the possible overhead and analyze the suitability of the security model based on requirements specific to the type of organization (critical applications, financial, etc.).
1.1 Motivation

The motivation behind this research is primarily based on security in shared untrusted environments, this matter is particularly important due to the increased cybersecurity risks posed by malicious users as recent attacks have shown. In the past couple of years there have been numerous cyber-attacks on large private and public organizations, including SolarWinds, the Irish Health Service Executive (HSE) and many others.

The cyber-attack which targeted software company SolarWinds, generated a large-scale breach which affected US Government Agencies, Microsoft, Google, and several other organizations that downloaded SolarWinds software in the form of a software update. According to Prof Alan Woodward, a cyber-security researcher at the University of Surrey, "Post-Cold War, this is one of the potentially largest penetrations of Western governments that I'm aware of." [6] The SolarWinds attack was generated with the injection of malicious code during pipeline jobs, after the code was compiled and prior to its packaging/distribution.

In May 2021, HSE has suffered a major ransomware cyber-attack that caused all its IT systems nationwide to be shut down. This is the most significant cyber-attack on an Irish state agency, as well as the largest known attack against a health service computer system in history, according to an independent report released from the HSE with lessons learned [7].

1. Current cyber-attacks highlight the necessity to implement higher-levels of cybersecurity
2. Need for a modern network security model for distributed systems – firewall at perimeter of trusted system no longer sufficient

The traditional network security model relies on the firewall networking component which has the purpose of providing network security by preventing unwanted network communication while permitting legitimate communication. Figure fig:dmz-network represents a high-level architecture of a demilitarized zone (DMZ) network which is a physical or logical subnet that separates a local area network (LAN) from untrusted networks, in this example the public internet. DMZ uses a firewall to provide network security by filtering incoming and outgoing network traffic based on a set of user-defined rules.

The traditional network security model (firewall approach) is no longer suitable due to the increased adoption of multi-tenanted deployment models, where unrelated entities share the same physical resources, this means a potential threat could be generated
within the same physical environment. In addition, organizations also avail of services from multiple cloud vendors for cost benefits and to prevent vendor lock-in, this expands the attack surface and complicates the security architecture.

The research aim of this work is to evaluate the suitability, in terms of performance overhead, of zero trust as a mechanism to implement trustworthiness in a multi-vendor cloud environment with the implementation of a sidecar proxy for the enforcement of policies and encryption. To date this evaluation has not been previously conducted as part of academic research. Quantitative data analysis of resource consumption and network latency was conducted to establish the suitability of Zero Trust in a single-cluster and multi-cluster heterogeneous network model.

In order to support the digital transformation and cloud transition, organizations can rely on well-tested frameworks and ecosystems of tools / technologies that the open source community has been contributing to for the past decades. Open source software is becoming increasingly important and utilized among organizations, since technical advancements in open source software are difficult to replicate, given the global collaboration of large pools of expert engineers. Enterprises are also investing on open source by assigning some of their top engineers to specific open source projects, see Linux, Kubernetes, Istio and more. This results in standards being set and maintained with a strong peer-review workflow prior to the release of newer software versions. To follow best practices and be part of the community can lower the risk of reliance on third party vendors, reduce costs and remain agile while adopting cutting edge technologies.

The deployment and configuration of a cloud environment should, according to the cloud native computing foundation, be automated and repeatable. "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic
environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.” [3].

Figure 1.4 describes container tools and technologies according to the latest Flexera State of the Cloud Report 2022 [2]. The report shows the adoption of Docker and Kubernetes for the containerization of applications and orchestration of containers continues to be considerable. Kubernetes can provide a common abstraction for the run-time of applications which can be deployed on heterogeneous underlying infrastructure offered by PCPs. "Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications". [8] Kubernetes is by far the most employed container orchestration tool in the cloud native space and is often synonymous with cloud in the context of cloud native applications. The potential of Kubernetes was identified by the largest PCPs, including Google, Microsoft, Amazon, Oracle and IBM, which all provide a managed Kubernetes service on their respective public cloud platforms. Throughout this research we employed Kubernetes clusters to provide a runtime environment for containers.
The adoption of a single public cloud provider implies reliance on a single vendor, which can result in potential downtime when a cloud provider’s data center is affected by a disaster. In addition, organizations may have compliance requirements, whereby European law data must be residing in a specific geographical region. This could be not possible if a cloud provider of choice does not operate in the required country. Another issue is the number of services accessible to cloud engineers are limited when depending on a single vendor. A possible solution to the aforementioned is the implementation of a multi-cloud strategy where services offered by multiple vendors are part of a heterogeneous system. The multi-cloud problem can partially be solved by creating inter-operability between applications running on separate PCPs, this can be achieved with the deployment of a multi-cluster service mesh.

A service mesh enhances Kubernetes capabilities and provides additional features such as observably and encrypted cross-cluster micro-services communication. In a cloud native system, security policies for authentication, authorization and encryption can be enabled with a service mesh. The side-car proxy is a core component of this architecture, acting as a policy enforcement point and intercepting networking communication from/to applications part of the mesh. The implementation of the side-car proxy can provide additional benefits, including increased observably of the system and traffic control. [9]

One challenge when developing cloud native applications is the implementation of security where the number of communication within the system is larger than those in a traditional monolith software, and run-time environments are often multi-tenanted where access is granted to unrelated / untrusted users. In addition, environments can also be deployed on multi-network domains for high-availability, compliance, and avoidance of vendor lock-in. In this deployment scenario elements within the environment cannot be trusted.

Zero Trust is a security model that may offer a solution to the above security challenges as it defines an approach to network implementation where users, machines and applications are authenticated and authorized with the enforcement of policies and encryption so that the system does not trust any entity by default [10]. The side-car proxy pattern implemented by the service mesh has recently proved to be an enabler of Zero Trust in cloud native microservices deployments by providing a proxy for each microservice to act as a policy enforcer [11][12].

The implementation of security using a service mesh and sidecar proxy does however introduce latency and system overhead due to an additional container in charge of intercepting all network traffic coming from and going to each application container running in the cluster environment. The sidecar proxy is in charge of enforcing security policies and encryption. The performance overhead occurs since physical resources are required
to operate the control plane (controller) and data plane (sidecar), while latency is affected by the enforcement of security policies and encryption.

1.2 Contribution

This research measures the performance overhead when security is enabled via a service mesh with authentication, authorization and encryption policies. Finally, we propose possible mitigation by leveraging an emerging technology: extended Berkeley Packet Filtering (eBPF) operates at the Linux kernel level. This work involves the design and implementation of a multi-cloud solution using infrastructure as code practices, which incorporates the automation of infrastructure, system configuration and application deployments. The solution entails using a combination of observability and secure microservices’ communication in both single and multi-network multi-cluster service mesh. To date no prior work has been conducted to analyze the performance of Zero Trust implemented with side-car proxy on a PCPs environment and to define the suitability of such network security implementation for critical application deployments. As part of the research, we configured a cloud environment consisting in a managed Kubernetes cluster, deployed a cloud native synthetic application, configured service mesh, tested the performance under load and analyzed results to establish the overhead in terms of latency, CPU and memory. The analysis is performed on both data and control plane. In addition, a performance enhancement is suggested with the use of eBPF technology which operates at Linux Kernel level. This research will focus primarily on Platform, Application and external Cloud clients, with particular focus on the secure communications between entities within those scheme blocks.

1.3 Structure of This Document

Chapter 2 provides a background and literature review at the basis of this research, Chapter 3 presents the methodology followed, Chapter 4 describes the implementation details, Chapter 5 illustrates the evaluation of the performance analysis and Chapter 6 is focused on discussion and conclusions for this research thesis.
Chapter 2

Literature Review

This chapter provides the literature review, background and state of the art in Cloud Native architectures with the focus on security and the evolution of security implementation at application level which is required when adopting a cloud native approach. There is a gap in academic research around the area of security in containerized environments and particularly within orchestrators environments. It appears that industry-lead projects and the open source community are defining the state-of-the-art in the context of cloud.

According to the Cloud Native Computing Foundation (CNCF)[13], which is the largest cloud open source foundation and a Linux foundation project, by definition, containers, microservices architecture, container orchestrators as well as service mesh and infrastructure as code are integral part of cloud native architectures [3]. The CNCF aims to build an ecosystem for cloud native software and advance the container technology while aligning the technology industry around the evolution of containers.

The aim of this chapter is to explain why a cloud native architecture is built on containers, the suitability of microservices architecture implemented with containers, and the requirements for container orchestrators, In addition, the chapter covers the evolution and possible extension of container orchestrators with a service mesh and the implementation of secure communication of microservices within such a deployment. Lastly, eBPF is described as an emerging technology that extends the capabilities of the Linux kernel to improve system performance.

The core concepts discussed in the next sections provide the foundation of the test-bed architecture and research analysis presented in the following chapters.
2.1 Cloud Native

In general terms being cloud native refers to software applications which are built for and deployed to the cloud. The most common pattern that supports a cloud native approach is based on a microservices architecture and containerization to allow inter-operability of decoupled independent services[14]. As container technologies, service oriented and microservices architectures have evolved they have been supported by the necessity to distribute applications consistently across heterogeneous environments[15]. The main motivations to transition to a cloud-native solution include: performance, efficiency, cost and scalability, according to D. S. Linthicum [16].

Performance is improved since the services offered by PCP normally avail of load balancing, scaling and other optimization capabilities out of the box. Efficiency is achieved by creating processes to create, configure and destroy resources in the PCP, which is normally via application programming interfaces (APIs) and software development kits (SDK), those processes can also be automated further improving efficiency. Costs is also reduced as a direct consequence of improved efficiency, as well as the access to on-demand services where PCP provide pay-per-use plan to give access to resources on as-needed basis. Scalability is achieved since additional resources can be scheduled to be created once certain thresholds are reached[16].

The CNCF [13] provided a trail map for the adoption of a cloud native approach, and the benefit include: reproducibility, composition, observability and provisioning, where Continuous Integration / Continuous Delivery (CI/CD) pipelines allow for the automation of deployment, IaC enables the programmatic creation and update of the cloud resources, composition achieved with containers that can operate as independent instances part of a larger multi-container system and observability where monitoring logging and tracing are enabled. This can be achieved with container orchestrators where Kubernetes is the leading project in this space [15][3]. Kubernetes was the first open-source project added to the CNCF landscape in 2015.[13] Figure 2.1 represents the Cloud Native Trail Map provides an overview for enterprises starting their cloud native journey.[17]

2.2 Virtualization and Containers

Cloud computing and the transition to a cloud architecture has been enabled by the advancement in virtualization technologies which allow instances of computer systems to run in a layer abstracted from the underlying physical layer.[18]. Virtualization provides an efficient way to make use of computer resources[19]. According to Pahl [20]: ”Historically, virtualization technologies have developed out of the need for scheduling
processes as manageable container units. These processes are the following: the file system, memory, network, and system information”.

### 2.2.1 Hardware Virtualization

Hardware virtualization involves the creation of a Virtual machine (VM) that can run an entire operating system, including the kernel. The component that allow for the creation of VM is called hypervisor, sometimes referred to as virtual machine manager (VMM). There are two types of hypervisors: type 1 and type 2. Type 1 hypervisors don’t use a host OS and executes directly on the CPU, while type 2 use the host OS.

Modern hypervisors directly access the hardware by using kernel modules, examples of hardware virtualization implementations include the following [21].

- VMware ESX: enterprise product using, the hypervisor is a microkernel which creates a service console VM that can administer hypervisor and new VMs.

- Xen: open source, the hypervisor runs paravirtualised guests for high performance, VMs are called domains, dom0 being the most privileged and can administer other domains.

- Hyper-V: Microsoft product, the hypervisor creates partitions for executing guest OS.

- KVM: open source, the hypervisor executes as kernel module, supports hardware-assisted extension and paravirtualisation, uses Quick Emulator (QEMU) user process and a Virtual Machine Manager (VMM).

- Nitro: AWS product, uses KVM with hardware support, and it does not use QEMU proxy, it provides near bare-metal performance to guest VMs.

In the context of virtualization, VMs are typically the backbone of the infrastructure layer. VMs normally store their file system in large files and run on a single large process on the host machine[20]. This can be an issue as physical resources are pre-allocated on the host machine and starting-up the VM can require several minutes. VMs enable the realization of cloud infrastructure-as-a-service (IaaS) model, where multiple users (tenants) can have virtual resources, including CPU, memory and disk storage.

![Figure 2.2: Use of CPU for Apps in a VM](image-url)
Where a software application consists in a few large components, a VM can provide a dedicated environment for each component and its own operating system. However, if the number of components increases, it is likely to have an inefficient physical resource allocation and higher operational overhead due to the necessity to configure each VM individually. Figure 2.2 describes how applications deployed on a VM can access the CPU via the hypervisor.

### 2.2.2 Operating System Virtualization

OS virtualization is the partition of the OS into smaller instances which Linux calls containers; containers are independent of the host and operate as guest servers. Containers are small, efficient and boot quickly [21].

Containers perform system calls directly on the kernel running in the host OS which is the only component sending instructions to the host CPU, containers do not require any virtualization as opposed to VMs which require a hypervisor, also described in figure 2.2.

Containerization provides a lightweight virtualization through the bespoke construction of containers as application packages from individual images (generally retrieved from an image repository) that consume fewer resources and time. Containers also support a more interoperable application packaging needed for portable, interoperable software applications in the cloud [22][20].

Figure 2.3 describes virtualization architecture in two possible scenarios: traditional hyper-visor architecture (VM) and container base architecture. [20]. Figure 2.3 also shows the difference in the management of guest operating system components, and it helps to understand that a process in a container runs in the container’s host OS while a process within a VM runs in the VM’s OS.

Containers allow to run several software applications on the same host machine where
each application is executed in isolation. According to R. K. Barik et al.: “the basic concept of Containerization is to allow different virtual instances and resource to share a single host Operating System (OS) and its dependent libraries, drivers or binaries”.

The use of containers allows for a faster and more efficient deployment, scaling and patching of applications, containers virtualization is particularly suited for the cloud since applications can be abstracted from the environment in which they are executed. Furthermore, containers require less resources and time to be created, so that they have been suggested as a solution for more interoperable application packaging in the cloud.

2.2.3 Container internals and Technologies

The isolation of containers is achieved by two Linux-native features: namespaces and cgroups. Namespaces provide the isolation of network environments through virtualization while cgroups limits and isolates the resource usage for a collection of processes. According to Linux official man page: “A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers”. Linux man page defines for cgroups as follows: “Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel’s cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on)”.

![Figure 2.4: Linux network namespaces](image)
By default, Linux has a single type of namespace to isolate a certain group of resources, namespaces can arbitrarily be added to create further process abstractions. Linux’s namespaces types include: mount (mnt), process id (pid), network (net), inter-process communication (ipc), uts and user id (user). Containers leverage network namespaces to limit the set of network interfaces the container process can see, hence providing isolation from containers running in other namespaces, or processes running on the host machine, which again is part of another namespace. Figure 2.4 describes how the physical network interface holds the root network namespace and how network namespaces A and B do not communicate to other network namespaces unless configured to do so. The resources used by a container are then constrained by the cgroups configuration which limits the amount of physical resources similarly to when processes run on separate machines.[27]

Container technologies support the life-cycle of containers including build, store and run operations. According to M.Luksa [27], Docker was the first container system that made containers easily portable across different machines, by simplifying the process of packing not only the application and its library, but the whole OS filesystem so that the package can be used to provision the application on any other machine running Docker. The main container technologies as of 2022 include Docker, Linux Containers (LXC), Hyper-V containers, rkt (formerly known as CoreOS Rocket), Podman, runC and containerD. The most complete solutions in terms of features are Docker, LXC, Hyper-V and rkt, while the rest of the technologies serve a specific function only. runC and containerD were originally part of the core Docker project and consequently extracted and managed as stand-alone open source projects. The most common container technology in 2022 is Docker. Docker allows building images which contains the necessary tools for the service to run. The image can then be pushed to a public or private registry where images can be pulled when required [28]. Figure 2.5 describes docker architecture which is based on a client/server model and uses a daemon to manage its containers.

Docker-based containers provide a similar level of isolation as VMs, but instead of using large monolithic VM images, container images are much smaller; the main difference is that container images are composed of layers which can be shared and reused across containers, as a consequence, in a host machine running multiple containers, the requirement is to download only the layers that were not already downloaded previously [27].

Managing containers is a challenge for organizations as microservices running on the containers do not communicate with each other.[23] In large systems where several containerized microservices run concurrently, the orchestration of containers must be organized to monitor their life-cycle, scaling capabilities, networking communications, persistent storage and any other operational capabilities necessary to deliver a functional
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>complete</td>
<td>Greatest adoption, simple architecture</td>
<td>Docker daemon is a single point of failure</td>
</tr>
<tr>
<td>Hyper-V</td>
<td>complete</td>
<td>Higher level of isolation and portability.</td>
<td>Larger infrastructure footprint. Windows only.</td>
</tr>
<tr>
<td>Hyper-V</td>
<td>containers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rkt</td>
<td>complete</td>
<td>Security, no daemon, interoperability.</td>
<td>Not actively developed.</td>
</tr>
<tr>
<td>Podman</td>
<td>engine</td>
<td>Security, no daemon, familiar CLI commands.</td>
<td>Container engine only.</td>
</tr>
<tr>
<td>runC</td>
<td>runtime</td>
<td>Container portability.</td>
<td>Container runtime only.</td>
</tr>
<tr>
<td>containerD</td>
<td>interface/daemon</td>
<td></td>
<td>Container interface only.</td>
</tr>
</tbody>
</table>

Table 2.1: Container Solutions comparison

application. This responsibility is given to container orchestration technologies which are supporting the container model and allow the operability of containers at scale in production environments.

![Docker architecture](image)

**Figure 2.5:** Docker architecture[28]

### 2.3 Microservices

According to Balalaie A. et al. [29]: "A microservices’ architecture is a cloud-native architecture that aims to realize software systems as a package of small services. Each
service is independently deployable on a potentially different platform and technological stack."

Microservices is a software approach that facilitates the development and distribution of workloads to the cloud [30]. This approach differs from the traditional software developed with a monolithic architecture, where according to A. Bucchiarone et al. [31] “... the modularization abstractions rely on the sharing of resources of the same machine (memory, databases, or files), and the components are therefore not independently executable”. Some issues of monolith architectures involve scalability and applications’ changes [31].

The microservices approach promotes loose coupling and high cohesion principles of software development, where a collection of loosely coupled services are managed independently to provide specialized functionalities to the overall software solution. As opposed to a monolithic approach, where the application consists of a single large unit of deployment, microservices are independently deployed, drastically increasing the software development velocity while avoid updating the full application for every incremental updates to the code-base.

<table>
<thead>
<tr>
<th>Microservices advantages</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming language agnostic</td>
<td>Ability to write application code in multiple programming languages</td>
</tr>
<tr>
<td>Single unit of deployment</td>
<td>Smaller independent unit of deployment increasing velocity</td>
</tr>
<tr>
<td>Loose coupling</td>
<td>Independent from other services, enabling interaction with outside applications</td>
</tr>
<tr>
<td>Application resilience</td>
<td>If a service is down the application can still operate</td>
</tr>
<tr>
<td>Resource use optimization</td>
<td>Smaller services can be allocated only the resources necessary</td>
</tr>
</tbody>
</table>

Table 2.2: Microservices advantages

Table 2.2 provides a summary of some main advantages of microservices. Containers support microservices architectures and are a logical way to package, distribute and run microservices applications since they provide the necessary level of isolation and virtualization, while allowing the application code written in any programming language to be executed on heterogeneous host machines.

Microservices, can be defined as the design of Service Oriented Software using a set of small services. M. Fowler et al. describe microservices as “… small applications that
can be deployed independently, with a precise and hardened interface, and easily integrated. Microservices are supported by middleware for communication and a platform for flexible and low-cost deployment.” [30][32].

Microservices are by design independent services which communicate with other services to provide the overall functionality of Software Applications. One of the complexities introduced by microservices is the number of interactions between application components which is substantially greater than a traditional monolith model; microservices are highly scalable, low coupled and are cheaper to run / maintain [33]. Microservices, according to C. Pahl [20] “... must be able to deploy often and independently at arbitrary schedules, instead of requiring synchronized deployments at fixed times. Containerization provides an ideal mechanism for their deployment and orchestration”. In a more recent paper, C. Pahl et al. [22] explains that: “A container holds packaged self-contained, ready-to-deploy parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run the applications.”

2.4 Container Orchestration

A dedicated infrastructure layer is required to provide orchestrating features and a runtime environment for containers.

Open source solutions including Kubernetes, Apache Mesos or Docker Swarm, provide containers runtime environments and offer the necessary networking infrastructure for containerized microservices to communicate with each other and deliver the functionality to end-users by exposing the application to the public [34].

PCP facilitates the creation of such environments by providing managed solutions and Kubernetes is the most widely used technology for the delivery of Cloud native applications.

Containers have the advantage of being decoupled from the infrastructure and be easily portable, however, they require a runtime environment and a system configurable to handle containers networking, data persistence, security, and any environment specific requirement. Typically, containers are scheduled to run within “container clusters”. Figure 2.6 represents a typical container cluster architecture [22].

Kubernetes is an open source container-orchestration system to automate application deployment, scaling and management. Kubernetes is directed by the Cloud Native Computing Foundation and was originally developed by Google, inspired on their internal container management tool. Kubernetes [8] provides a complete container cluster
solution to deploy and orchestrate containers in testing, staging and production environments. Kubernetes managed solutions are offered by major cloud providers to include Amazon Web Services, Microsoft Azure and Google Cloud.

Figure 2.7 represents the Kubernetes components at node and control plane level. Kubernetes clusters consist in a set of worker machines or VMs that run containerized applications. Each worker node has a container runtime environment which handles the execution of containers under the supervision of the API server which is part of the control plane. Kubernetes uses several abstractions to manage workload application, pods are the smallest, most basic deployable objects in Kubernetes, they correspond to a single instance of a running process and contain one or more containers. Multi container pods are managed as a single entity and share all pod’s resources. [23][8]
2.4.1 Kubernetes networking

Kubernetes networking model is based on a few fundamentals rules, where pods should be able to communicate with other pods without Network Address Translation (NAT), programs running on a node should communicate with any pod on the same node without using NAT, each pod has an Internet Protocol (IP) address that can be reached by any other pod in the cluster.[8] This model is referred to as a "flat networking model" and greatly supports microservices architectures by facilitating the communications between containers.

When creating a managed Kubernetes cluster in a PCP, nodes within the cluster get assigned an IP address part of the Virtual Private Cloud (VPC) network, this is the node’s connection to the rest of the cluster. This IP allows connectivity from control components like kube-proxy and kubelet to the Kubernetes API server, as described in figure 2.7. Pods running on each node get an IP from the node IP address pool is a Classless Inter-Domain Routing (CIDR) range specifically assigned to pods and part of the cluster’s configuration. The pod IP address is shared by containers part of the pod and provide communication with other pods in the cluster.[35][27]

![Figure 2.8: Linux network namespaces and pods](image)

When a pod is created, and the pod gets assigned to a node by the Kubernetes scheduler, the pod gets its own Linux network namespace on the node, the container network interface (CNI) assigns an IP address to the pod and attaches the pod to the container network, network packets can then flow to and from the pod. If multiple containers are part of the same pod, all containers will be part of the same network namespace. Linux has the ability to create several virtual device types and veth is one of them. veth are virtual ethernet devices which are created in pairs and are typically used to connect
network namespaces. Figure 2.8 illustrates the final state of a pod creation, where the CNI creates a network namespace and assigns an IP for the pod and then attaches the pod to the host network by creating a veth pair.

Pods are ephemeral by nature, since Kubernetes regularly destroys and recreates them when pods configuration is updated, node pools are upgraded or nodes become unavailable. As a consequence, the IP address assigned to a pod is not reliable. Kubernetes uses another abstraction to provide a stable IP address and ports: services. Each Service has an IP address, called the ClusterIP, which is assigned from the cluster’s VPC network. This IP address is stable and does not change during the lifetime of the service. Services use pod labels to group multiple pods in a logical unit and act as a load balancer for the set of pods that match a given label, this is done using a label selector in the service configuration. [35][27][8]

Figure 2.9 describes how services and pods communicate using labels and labels selectors. The service front-end provides access and load balancing for pod front-end since the pod matches all labels defined in front-end label selector. Similarly, the back-end service load balances the two back-end pods that match all labels of back-end service. If a pod does not match all labels of any service, for example the worker pod in figure 2.9 can still communicate to all services in the cluster, but it is not reachable via any services.

![Figure 2.9: Kubernetes Services and labels](image)

The default networking configuration in Kubernetes avails of **kube-proxy** abstraction which leverages iptables. Iptables is a user space interface to set up, maintain and inspect the tables of IP packet filter (Netfilter) rules in the Linux kernel.[36]. The role of kube-proxy is to assure clients can connect to services defined via Kubernetes API so that connection to the service IP and port is redirected to a backing pod. If more than one pod is backing a service, the kube-proxy performs load balancing across those pods [27]. A kube-proxy runs on each node as a static pod and is not an actual proxy, its function is to update iptables rules to redirect network packets to pods [35].
The name kube-proxy was given based on the original, less performant implementation, where the kube-proxy was acting as a proxy intercepting connection destined to a service IP and configuring iptables rules to redirect the connections to a proxy server [27].

Figure 2.10 describes the role of kube-proxy and how a network packet is sent from PodB-1 to the virtual IP 17.31.0.1 of ServiceA. The kube-proxy watches for changes to services and endpoints via the Kubernetes API server and updates iptables rules accordingly. When the packet is sent from PodB-1, the node kernel checks iptables rules and one of the rules says the virtual IP needs to be replaced with the IP and port of a randomly selected pod, in figure 2.10 the randomly selected pod is PodA-2 which is a pod matching all labels selectors of ServiceA, as previously described in 2.9.

### 2.5 Security and Zero Trust

Security provides organizations with a significant challenge in regard to multi tenant architectures. Organizations moving to the public cloud need to implement security within a runtime environment such as Kubernetes, where the perimeter can no longer be defined due to the fact that multi-tenanted applications are running in the environment and sharing the same physical resources as well as applications running in the cluster. Traditionally, organizations configured trust environments to implement network security; users and services with access to the environment were considered trusted and allowed to execute operations within the system. This security model can also be defined as “Trustworthy” and is suitable for monolith applications, it usually involves the configuration of a firewall which provides the environment perimeter.
A more recent security model defined as “Zero Trust” does not have the concept of environment perimeter, and it is more suitable for cloud applications where systems can be multi-tenant and resources in the same environment are shared by users of different organizations. In a Zero Trust network the authentication, authorization and encryption are enforced at every communication between users, device, and applications. The conceptual typology of Zero-Trust strategy described by S. Mehraj et al. [37] “begins with an assumption that all the data and transactions are required to be deemed as untrusted from the inception”. According to the definition of Zero Trust Architecture (ZTA) of the National Institute of Standards and Technology (NIST): “Zero trust architecture is an end-to-end approach to enterprise resource and data security that encompasses identity (person and nonperson entities), credentials, access management, operations, endpoints, hosting environments, and the interconnecting infrastructure.” [10]. In the same paper, NIST defines the enabler of Zero Trust to be: Enhanced Identity Governance, Micro segmentation or network and software defined perimeters. Software Defined Perimeter (SDP) is a combination of micro segmentation, to provide network isolation, and identity-based access, to ensure authentication and authorization is constantly verified. This research will focus on implementing ZTA using network infrastructure and Software Defined Perimeters since it provides a more complete security approach, which is implemented at both network infrastructure and device/user level.

SDP is a concept proposed by the Cloud Security Alliance (CSA) as a security model and framework with the potential to protect networks in a dynamic manner [38][39][40].

![Diagram of Device Agent/Gateway Model](image)

**Figure 2.11: Device Agent/Gateway Model[10]**

With Software Defined Perimeter, according to NIST [10] “...the Policy Administrator (PA) acts as the network controller that sets up and re-configures the network based on the decisions made by the Policy Engine (PE). The clients continue to request access via Policy Enforcement Points (PEPs), which are managed by the PA component.”
This approach is frequently implemented at the application layer (layer 7) with the deployment model of agent / gateway, as per 2.11.

The implementation of a Zero Trust model improves the security posture of organizations and provides a mean to mitigate threats, the cyber-attack on SolarWinds serves as an example: FedScoop, a platform delivering Government Tech news, reported that about 18,000 organizations were infected and that the implementation of Zero Trust allowed to mitigate the malware’s ability to spread across networks, cyber experts say [41]. According to Sean Frazier, Federal Chief Security Officer at Okta, “If SolarWinds had happened a year ago or two years ago, I think agencies would have had a lot more consternation about it” [41], this is due to the fact that Government agencies are actively adopting a Zero Trust approach.

Zero trust inevitably results in performance overhead due to the requirement of enforcing security policy for the authentication and authorization as well as the encryption of data in transit and at rest.

### 2.6 Sidecar Proxy

The sidecar pattern relates to a motorcycle sidecar where the two components are attached, and each motorcycle has its own sidecar. The sidecar proxy is an application design pattern which can be implemented as an abstraction layer to enable networking, monitoring and policy enforcement while keeping the logic of the main architecture unchanged. This pattern allows single-purpose containers to cooperate in a high cohesive loosely coupled architecture where the sidecar container extends and enhances the capabilities and features of another container without modifying it.

The sidecar pattern can be applied to microservices so that a sidecar proxy container is deployed alongside an application container. Figure 2.12 represents the sidecar pattern in his simplest form, where two applications containers are communicating indirectly via sidecars containers that are attached to each application container and act as proxy.

The implementation of the sidecar pattern in a microservices’ architecture can provide the ability to enable Zero Trust by enforcing policies for authentication, authorization, and encryption of the communications between services. This can be achieved by allowing a microservice running in a container to collaborate with another (proxy) container which has the responsibility to intercept network traffic coming from and going to the microservice container with application workload.
Traditionally the security implementation logic was part of the responsibility of the software application, however, in microservices architectures, where services can be developed in heterogeneous programming languages, each application would have its own specific implementation, which introduces implementation complexity and operational challenges, for example the management and rotation of client/server certificates for secure communication between services. The sidecar is a step towards a centrally controlled security solution acting as gatekeeper for applications.

Some advantages of the sidecar proxy pattern include the following:

- Separate application business logic from infrastructure operations which can be part of a common abstraction layer to reduce the complexity of microservices code
- Prevent code duplication as several third-party components are shared among microservices
- Decouple the application code and sidecar to allow independent upgrades and minimize dependencies with the underlying platform

2.6.1 Service Mesh

The definition of a service mesh, according to W. Li et al. [42] “A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It’s responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware”. Figure 2.13 provides an example architectural overview for a service mesh [42].

The Service Mesh layer is an implementation of the sidecar proxy pattern at the data plane layer with the addition of a control plane layer to manage the service mesh.
Service Mesh is an infrastructure layer that complements microservices architectures in enabling observability, reliability and security in service-to-service communication. A service mesh uses a sidecar proxy which sits alongside a microservice and routes requests to other proxies within the network mesh. In the context of microservices, a service mesh can complement Kubernetes, enhance some of its functionalities and provide more control over containers.

![Figure 2.13: An architectural overview example of the service mesh approach.][42]

NIST defines a service mesh as "A dedicated infrastructure called the service mesh provides services for the application (e.g., authentication, authorization, routing, network resilience, security monitoring), which can be deployed independently of the application code." [12]

F. Hussain et al. [43] describe a service mesh as an infrastructure layer that “...allows the communication among services, in a microservices’ architecture. Service meshes provide service discovery, load balancing, and authentication capabilities for microservices. The NIST defines a service mesh as two distinct architectural components: data and control plane, where the interconnected set of proxies that control the inter-services communication represents the data plane [11]. According to R. Chandramouli et al. [11]: “The specialized proxy that is created for each service instance (i.e., sidecar proxy) performs the runtime operations needed for enforcing security (e.g., access control, communication-related), which are enabled by injecting policies (e.g., access control policies) into the proxy from the control plane. This also provides the flexibility to dynamically change policies without modifying the microservice’s code”.

---

[42]: #/fig-2.13-arch-overview-service-mesh
2.6.2 Istio Service mesh and Security implementation

Istio is an open source service mesh that layers transparently onto existing distributed application [9]. Istio offers traffic management, observability and security capabilities by leveraging a control-plane / data-plane model, where the sidecar proxy represents the data plane. The data plane is responsible to provide information to the control plane, as well as enforcing behavior as directed by the control-plane. The security implementation of Istio include the secure service-to-service communication in a cluster with Transport Layer Security (TLS) encryption, strong identity-based authentication and authorization [9].

Istio control plane can act as a certificate authority (CA) for key and certificate management, which allow for the implementation of secure multi-cluster configuration. Istio securely provisions strong identities to every workload with X.509 certificates while taking care of key and certificates rotation.

Figure 2.14 describe Istio service mesh [9] security architecture, where the sidecar proxy component acts as a PEP on security policies injected by the control plane component and configured declarative way. Figure 2.14 shows how the application logic is no longer coupled with the security implementation since the sidecar proxy is responsible for intercepting all communication coming from and going to the application containers.

![Figure 2.14: Istio Security Architecture](image-url)
A service mesh like Istio can support in implementing a Zero Trust security model with the following features which are described in details in the official Istio documentation, under the security section [9]

1. Authentication: each user request is authenticated to verify the credentials used for the request. Istio enables request-level authentication with JavaScript Object Notation (JSON) Web Token (JWT)

2. Authorization: provide mesh, namespace, and workload-wide access control for workloads in the mesh

3. Encryption: Istio tunnels service-to-service communication through the client and server-side PEPs, which are implemented as Envoy proxies.

Istio uses mTLS authentication for encryption. mTLS is a type of mutual authentication where the two parties in a connection authenticate each other using the TLS protocol. The difference between TLS and mTLS is that in TLS only the server presents the certificate issued by a CA to establish its identity, while in mTLS the both client and server present a client certificate issued by the CA.

According to the official documentation, [44] JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. Istio uses JWT tokens for authorization of Hypertext Transfer Protocol (HTTP) requests which is enforced by the envoy side-car proxy.

The sidecar proxy implementation results in performance overhead due to an additional proxy process per workload and the interception of all network requests going to and coming from microservices. A possible mitigation to this risk can be offered by programmable networks and particularly the eBPF support in the Linux kernel which prevents compromising execution efficiency and safety.

2.7 Extended Berkley Packet Filtering (eBPF)

Extended Berkley Packet Filtering (eBPF) is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules[45]. According to the official eBPF’s
website [45], eBPF “can run sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules”. Figure 2.15 describes eBPF loader and verification architecture.

eBPF is a recent technology based on Berkeley Packet Filter (BPF) which was originally developed for network packet filtering by deploying a kernel agent called packet filter which instead of passing all packets to user-space for network monitoring, it filters out unwanted packet at kernel level [46], this drastically improved efficiency and performance of network monitoring for Linux systems.

According to M Xhonneux et al. “eBPF programs can call helper functions, which are functions implemented in the kernel. They act as proxies between the kernel and the eBPF program. Using such helpers, eBPF programs can retrieve and push data from or to the kernel and rely on mechanisms implemented in the kernel. A given hook is usually associated with a set of helpers” [48].

2.7.1 eBPF Vs iptables

In the context of Kubernetes, eBPF technology is used for the networking implementation by some container network interfaces (CNI) plugins, including Cilium and Calico. The eBPF approach integrates the packet filtering functions in the eBPF program and bypasses the iptables processing by leveraging the eXpress Data Path (XDP) or Traffic Control ingress/egress (TC) hooks. [36]

Figure 2.16 represents the chain processing of iptables and hook points in the Linux Kernel. Each iptables chain comprises a number of tables, each with a different purpose. There are four kinds of tables: raw, mangle, nat, and filter. The raw table is used to split
the traffic without a need for the connection to be tracked. The mangle table is used to change the QoS settings of packets. The nat table is for network address translation and the filter table is used for packet filtering.[49][36]

![Iptables chain processing and hook points](image)

Figure 2.16: Iptables chain processing and hook points[49]

In a research paper of M. Bertone et al. [49], where the performance of eBPF and iptables were evaluated by attaching to the TC ingress and egress of the host interface and number of rules was increased, test results confirmed that bpf-iptables outperforms iptables by an order of magnitude when a high number of rules is used. This is due to the improved algorithm and different optimizations on the classification pipeline that are allowed by the dynamic code injection of eBPF, with a vanilla Linux kernel.

![Sidecar proxy networking in a Service Mesh](image)

Figure 2.17: Sidecar proxy networking in a Service Mesh

In a service mesh configuration, eBPF networking on Kubernetes nodes can be more efficient in receiving and sending network packets which need to go through the sidecar proxy injected in each pod. Figure 2.17 describes how a network packet travels from one service to the proxy running in the same pod and same host machine.
Figure 2.18 describes how eBPF can prevent network packets sent and received on the same host can prevent traveling through the full Linux networking stack. This can be achieved doing a shortcut between network sockets, copying the data from the service socket to the proxy socket and avoiding the use TCP/IP which is unnecessary given the service and the proxy run on the same host [50].

In a non-academic context, some research has been conducted to analyze the performance variation when a service mesh is configured. An example results summary is provided in the graph of Figure 2.19, where latency measurements in microseconds and ranked by percentile for two containers running in a pod talking to each other via one of the listed proxies without policies, routing rules or iptables [47] where Cilium In-Kernel leverages eBPF for containers networking.
Chapter 3

Zero Trust Performance in Cloud Native Systems

This chapter outlines the plan to deploy and configure a multi-cluster system on PCPs. The first section describes the evolution of this work, since the author has been working on research and implementation for almost three years, combining academic activity and industry experience. The next sections will describe the problem definition, objectives, requirements and the system design.

The final system is equipped with the capability to enable and disable a zero-trust configuration in terms of application level communication between in-clusters services and external clients. In addition, an external component, deployed in the same geographical region is configured to put the system under load and test the system performance. The evaluation is conducted leveraging a latency measurement tool, and a real-time metrics monitoring tool to analyze system resources consumption.

3.1 Project evolution

The purpose of this project evolved out of years spent working with cloud native technologies which led to an opportunity to work on a project, whose objective was to provide a multi-cloud solution for an organization. This led to an opportunity to work with a service mesh infrastructure layer which can complement a distributed system architecture and provide interoperability between systems on heterogeneous networks, as well as security, observability, traffic control and more.

The research conducted generated interest from academics and practitioners alike, both in an academic context and industry setting, resulting in receiving the award of the best
Figure 3.1: Initial project architecture

Computer Science project in terms of innovation and complexity in 2020. Figure 3.1 describes the high-level architecture of "Multi-cloud and IaC, a Cloud Native solution to a multi-cloud problem" which was the original research title.

Based on the interest generated and a desire to complete further examination of the area, it was decided to expand the research focusing on the role of security in a multi-cloud architecture. The initial research findings were formalized in a research paper which was published in the conference proceedings of the IEEE CLOUD 2021 conference with the title "Performance analysis of Zero-Trust multi cloud" [51].

Figure 3.2: Architecture of Zero Trust multi-cloud

Figure 3.2 describes the architecture of multi-cloud Zero Trust solution for which the
system performance was evaluated and results presented in the research paper "Performance analysis of Zero-Trust multi cloud" [51].

### 3.2 Problem Definition

The following describes the problem that our research looks to address, it begins with outlining the process of implementing security in IT systems. The security implementation in IT systems relies upon compute resources to process security operations and increases latency in network communication due to security steps to be performed. In general terms, this can be defined as performance overhead. Quantifying performance overhead is vital to define the suitability of the security solution for a given application.

Key performance indicators (KPIs) allow to measure the performance at a specified level such as system, networking or application. To quantify the performance overhead with Zero Trust enabled, we monitor the system physical resources, including CPU and memory as well as networking response time or latency when the test application is under load. KPIs and measurement units are as follows:

1. CPU - time in second
2. Memory - in MBi
3. Latency - in seconds

This led to the use of the following hypothesis:

**Hypothesis 1** - is “Given the scale of modern cloud computing systems is there a scaling threshold at which a Zero Trust security performance degrades?” Hypothesis: “Does the performance of a Zero Trust system degrade at scale in the cloud? What is the threshold?” Null-Hypothesis: “Zero Trust systems performance will never degrade regardless of scale in the cloud”

**Hypothesis 2** - is “Can the performance overhead of sidecar proxy implementation in a Zero Trust model be mitigated with programmable Linux kernel technologies?” Hypothesis: “Performance overhead can be mitigated using eBPF in a Zero Trust model with sidecar proxy” Null-Hypothesis: “Performance overhead cannot be mitigated using eBPF in a Zero Trust model with sidecar proxy”
3.3 Objectives

Following on from the definition of the problem, we outline the research objectives which consist in the implementation of a repeatable infrastructure solution where the creation and configuration of the environment, as well as applications deployments is programmatic to facilitate the testing of the system under different load and security configuration.

We enumerate the primary objectives as follows:

1. Design and develop a multi-cluster framework on PCPs
2. Design and development of a test environment
3. Evaluate the performance at scale of the Zero Trust system
4. Evaluate the performance of a Zero Trust system with the addition of eBPF

Addressing our first objective which is the design and development of a multi-cluster framework on PCPs we do the following: deploy, configure, and manage a multi-cloud system where infrastructure resources are programmatically created using Infrastructure as Code (IaC), Git Operations (GitOps) and Development Operations (DevOps) techniques. The framework will allow the operator to deploy the infrastructure and configure clusters of microservices where workloads will run. In addition, the configuration of a side-car proxy implementation will be applied to each cluster part of the system to allow for the communication of microservices across clusters, the enforcement of security policy, encryption of communication between services and the enabling of system monitoring. The outcome of the successful execution of the workflow is a Zero Trust system on heterogeneous networks.

The second objective is the design and development of a test environment for the performance analysis of the multi-cluster Zero Trust deployment. This requires the definition of the metrics for the analysis of the performance as well as the creation of a workflow to generate load, access metrics and aggregate metrics for evaluation.

The third objective is to evaluate the performance at scale of the Zero Trust system. The evaluation will include the analysis of requests latency (and any other metrics defined in objective 2) to establish the suitability of this configuration for the deployment of critical applications (autonomous vehicles, high precision machines, etc.) and/or the performance overhead to be considered when configuring a Zero Trust environment. Moreover, a comparative analysis of a Zero Trust and not-Zero Trust configuration will
be conducted to uncover the impact of encryption and policy enforcement on system performance.

The final objective is to evaluate the performance of a Zero Trust system with the addition of eBPF proxy accelerator which leverages eBPF technology in the Linux kernel. This final evaluation is an effort to mitigate the performance issues which are expected to be generated by the security policies enforcement during the evaluation for objective 3.

3.4 Solution requirements

Having outlined our objectives we propose the following solution requirements as part of our research which is to do the following: deploy the infrastructure and test the deployment of service mesh under load. The requirements are categorized as Functional requirements and non-Functional requirements. Functional requirements are mandatory, describe what the solution does and represent features of the solution. Non-Functional requirements are optional, describe how the solution works and represents properties or quality attributes of the solution.

3.4.1 Functional requirements

1. Authentication

2. Authorization

3. Encryption

4. System resources monitoring

5. Persistence of test results

Authentication

Authentication is required for the following reasons:

Description

The solution should authenticate both internal and external clients.

Rationale

The Kubernetes infrastructure is complemented with a service mesh installation where each request in-cluster and cross-clusters are authenticated and authorized with security policies enforced by side-car proxy and managed by Istio control plane.

Fit criterion
Only authenticated requests are successful, HTTP requests respond with 200 code for authenticated requests.

**Authorization**

Authorization is required for the following reasons:

*Description*

The solution should include a mechanism to authorize requests.

*Rationale*

The Kubernetes infrastructure is complemented with a service mesh installation where each request in-cluster and cross-clusters are authenticated and authorized with security policies enforced by side-car proxy and managed by Istio control plane.

*Fit criterion*

Only authorized requests are successful, HTTP requests respond with 200 code for authorized requests.

**Encryption**

Encryption is required for the following reasons:

*Description*

The solution should encrypt network communication within the cluster.

*Rationale*

The service mesh configuration allow the deployment of PeerAuthentication policies to enforce encryption of in-cluster and cross-cluster network traffic.

*Fit criterion*

All traffic within the mesh is encrypted.

**System resource monitoring**

System monitoring is required for the following reasons:

*Description*

The solution should include monitoring of the system hardware components.

*Rationale*

The clusters deployed on PCPs should include a monitoring solution to allow the analysis of system-metrics including CPU and memory.

*Fit criterion*

Clusters are equipped with monitoring solution, CPU and memory consumption can be analyzed at any time.

**Persistence of test results**

Testing is required for the following reasons:
Description
The solution should allow for testing and persistence of test data.

Rationale
The clusters are tested under load, on test completion test results must be aggregated and persisted for data analysis.

Fit criterion
All tests results are persisted and available on test completion.

In addition to functional requirements, we proposed the following non-functional requirements to ensure the usability, and efficiency of the solution.

3.4.2 Non-Functional requirements

The non-functional requirements include the following:

1. Portability
2. Reliability
3. Cost
4. Security

Portability
The solution should be portable, code should be in a repository which can be accessed by multiple contributors. The solution should be running in multiple environments, such as VMs and physical machines using Unix-based operating systems.

Reliability
The solution should be reliable and a fail-over mechanism should be in place to handle failures. Faults should be isolated, when a service becomes unavailable another cluster, configured in a separate CSP, should be available.

Availability
The solution should be highly available with possible multiple regions’ deployment. High availability is obtained by adopting multiple CSPs in the same or diverse regions. Clients across different regions should be able to access services with lower-latency.

Cost
The solution should be cost-effective, since Cloud resources are purchased on-demand from CSPs, cost should be reduced. The solution entails programmatic provision and destroy of resources. This includes the option to select a service according to the task
to be executed, such as computing processing from one CSP and static storage from another CSP, or a combination of both. This can reduce costs as the best offer at a given point in time can be selected.

**Security**

The solution should be secure in terms of communication between microservices, end-to-end encryption is enabled with the use of mutual TLS (mTLS). mTLS protocol verifies the identity of both receiver and transmitter of the package, ensuring the communication is private.

### 3.5 Design

Having outlined our functional and non-functional requirements, we propose the design of the following infrastructure. This section provides a description of the solution architecture, including technologies, programming languages, frameworks and service providers. Figure 3.3 Represent a High-level overview of the multi-cluster system architecture deployed on Google Cloud Platform.

![Google Cloud Platform](image)

**Figure 3.3:** High-level architecture on Google Cloud Platform

The overall architecture is formed by the composition of three distinct layers:

1. **Infrastructure**
2. **System Application**
3. Test Application

The overall system creation and resource management is programmatic to maintain consistency, repeatable components and the ability to manipulate the system capability based on a set of configuration parameter. This can be achieved following Infrastructure as Code practices and specifically using Pulumi Open Source technology which allows leveraging standard programming languages such as Go, Python and NodeJS to deploy, configure and manage the life-cycle of infrastructure projects based on public cloud services. Figure 3.4 describes Pulumi architecture, where Pulumi engine is responsible for the creation, update and delete of cloud resources as well as maintaining a deployed state up to date.[52]

![Pulumi architecture diagram]

The IaC approach facilitates the configuration of multiple environments with diverse security implementation, for example enabling / disabling a service mesh and security policies. This allows to maintain the same system as a benchmark and test the behavior under load when diverse security configurations are applied. Figure 3.5 describes the interactions between Pulumi project, which is handled by the Pulumi engine, and the PCP of choice which is GCP. Pulumi has the responsibility to create cloud resources as well as deploy system and test applications on the Kubernetes cluster.
3.5.1 Infrastructure

The infrastructure is entirely cloud-based with the employment of multiple PCP vendors and services on-demand, such as managed Kubernetes clusters and VMs for testing. The PCPs utilized for this research are Amazon Web Services (AWS) and Google Cloud Platform (GCP) with their respective Kubernetes distributions: Elastic Kubernetes Service (EKS) and Google Kubernetes Engine (GKE). The programmatic creation and management of infrastructure resources is achieved with Pulumi IaC.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Master Version</th>
<th>nodes</th>
<th>vCPU</th>
<th>Mem</th>
<th>image</th>
</tr>
</thead>
<tbody>
<tr>
<td>GKE</td>
<td>v1.18.16-gke.302</td>
<td>2</td>
<td>8</td>
<td>32Gb</td>
<td>E2-standard-4</td>
</tr>
<tr>
<td>EKS</td>
<td>v1.18.9-eks-d1db3c</td>
<td>2</td>
<td>4</td>
<td>16Gb</td>
<td>T2.medium</td>
</tr>
</tbody>
</table>

Table 3.1: Cluster Specification

The configuration of service mesh and implementation of Zero Trust principles of encryption, authentication and authorization can be enabled on a multi-cluster within the same provider or separate provider. For the purpose of this research several implementations were deployed, including multiple clusters in the same provider (AWS, AWS or GCP and GCP), multiple clusters in multiple providers (AWS, GCP) and single cluster on the same provider with the enabling/disabling of service mesh and security policies between tests. The last implementation is the most reliable in terms of resource analysis.
and latency analysis, since the system is deployed within a same network and communication between test server and cluster are not only within the same geographical region but also within the same provider’s network. As a consequence, GCP was chosen as PCP for the most comprehensive set of tests and the cluster configuration is as follows.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Master Version</th>
<th>nodes</th>
<th>vCPU</th>
<th>Mem</th>
<th>image</th>
</tr>
</thead>
<tbody>
<tr>
<td>GKE</td>
<td>v1.20.10-gke.1600</td>
<td>2</td>
<td>4</td>
<td>16Gb</td>
<td>E2-standard-4</td>
</tr>
</tbody>
</table>

Table 3.2: Cluster Specification

### 3.5.2 System Applications

System application include Istio service mesh as the implementation of side-car proxy pattern, Prometheus for the monitoring of system resources and Grafana to visualize the metrics in configurable dashboards.

> "Istio extends Kubernetes to establish a programmable, application-aware network using the powerful Envoy service proxy. Working with both Kubernetes and traditional workloads, Istio brings standard, universal traffic management, telemetry, and security to complex deployments." [9] Figure 3.6 describes the interactions between Istio control plane and data-plane, where control plane is in charge of defining the behavior that
data plane will enforce by intercepting all network communication going to containers running in Kubernetes cluster pods.

Istio is a complex and technology offering several features and for the purpose of this research Istio will be employed as an enabler of Zero Trust with the implementation of side-car proxy pattern to serve as a security policy enforcer.

Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels. Prometheus is used as a data source for Grafana dashboards.[53]

Figure 3.7 represents a Grafana dashboard view for a 3 hours time series where CPU and memory are analyzed by namespace. This view clearly shows when the system is under load and allows to quickly perform a comparison on a baseline resource consumption when the system is not under load.

![Grafana dashboard](image)

**Figure 3.7: Grafana dashboard**

Finally, a performance enhancer is proposed employing eBPF technology offered by Cilium container network interface plugin. This can be achieved by enabling GKE data-plane V2 which is offered as a pre-configured option when creating and configuring a GKE cluster. This option must be enabled at cluster creation and cannot be altered once the cluster is up and running. The difference between the default cluster configuration and data-plane V2 is the networking implementation within the Kubernetes cluster, where default uses kube-proxy and IP tables while V2 uses eBPF.
### 3.5.3 Test Application

The object is to test the system with a set of tests which is comprehensive enough to simulate a production-like environment for a small/medium size organization. For example: the number of HTTP requests should be at least 1000 per second. There are several types of tests that can be performed on a system, including: Load test, Endurance test, Stress test etc. Hypothesis 1 of this research is to discover whether a scaling threshold is present when security is enabled in a multi-cluster system. In order to test this hypothesis it is necessary to perform a load test to see how the system responds to a sudden increase in requests for a set test duration.

The latency performance can be monitored and evaluated with the use of a load generator tool that sends sequential or concurrent requests to an application running in the cluster. Several tools are available for the performance analysis, those include Fortio, Vegeta, Blueperf, Jmeter, Gatlin and more. A comparison analysis of the tools was conducted based on four parameters: open source, documentation, ease of use VS feature set, and project health.

<table>
<thead>
<tr>
<th>Project</th>
<th>Open Source</th>
<th>Ease/Features</th>
<th>Documentation</th>
<th>Health</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortio</td>
<td>yes</td>
<td>4</td>
<td>VeryGood</td>
<td>Good</td>
</tr>
<tr>
<td>Vegeta</td>
<td>yes</td>
<td>2</td>
<td>Ok</td>
<td>Good</td>
</tr>
<tr>
<td>Jmeter</td>
<td>yes</td>
<td>3</td>
<td>Good</td>
<td>Ok</td>
</tr>
<tr>
<td>Gatlin</td>
<td>yes</td>
<td>4</td>
<td>Ok</td>
<td>Ok</td>
</tr>
</tbody>
</table>

*Table 3.3: Load generator tools comparison*

Fortio was chosen as a performance load tool since it is open source, it was originally developed by the Istio team (and used to be part of the Istio project), it has all features required, is easy to use, documentation is comprehensive, and the project is healthy with a group of active contributors from the open source community. Table 3.3 represents the summary of the load generator tools comparison analysis.

For the performance evaluation Fortio is assuming two separate functions: a synthetic application serving as echo server to respond to HTTP requests and load generator tool to specify a set query-per-second load and record latency histograms.

The test executions follow a predefined sequence of steps, where the start of recording of results, application of load, generation and collection of test data must be coordinated and performed sequentially.

Figure 3.8 represent a sequence diagram that depicts the objects involved in a testing scenario and the time-based sequence of messages exchanged between the objects during the test.
3.5.4 Development tools and technologies

In terms of programming languages, NodeJS and TypeScript are used for Pulumi IaC project, and bash scripts for the configuration of the local environment, test VM and supporting functions for the improvement and stability of the overall testing framework.

The following technologies and tools are utilized to support development and test stage:

1. VSCode - include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git

2. Postman - HTTP client for API testing

3. GitHub - provider of Internet hosting for software development and version control using Git
Chapter 4

Testing framework for cloud-native Zero Trust

This chapter describes the implementation architecture and features of the system. A key aspect of the architecture used is the creation and configuration of the cloud environment in preparation for the performance tests. This work leverages cloud native open source technologies and DevOps practices, including Infrastructure as Code to programmatically create a reproducible environment with a custom configuration according to the test requirements.

The next sections include Introduction 4.1, Cloud configuration 4.2, Infrastructure as code 4.3, Testing 4.4 Data Aggregation 4.5 and Data Analysis 4.6.

4.1 Introduction

The implementation of the research work involves several steps and technologies which serves multiple purposes at distinct abstraction levels. This includes the scripting required for the configuration of PCP roles/permissions necessary by the IaC program, coding the IaC program for the creation of the PCP infrastructure including deployment of monitoring, service mesh and test application. In addition, scripts where developed to bootstrap the configuration of test virtual machine and the execution of tests. Finally, programming for the data analysis to clean, group and evaluate statistic test results datasets.

The main technologies in implementing the architecture include: Kubernetes, Istio, Docker, eBPF, Prometheus, Fortio, Helm, Pulumi, NodeJS, Bash, Python and Jupyter.
The technology stack was selected based on an analysis of the most popular and best supported technologies for every abstraction level, for example: Prometheus is the most important open source monitoring solution and offers a great integration with Kubernetes. Similarly, Istio is the most featureful service mesh available in 2022, Docker was used for simplicity and the availability of docker images on public repositories which allowed the deployment of Helm charts without additional configuration changes. Helm is virtually the only real package manager for Kubernetes, and it was selected as it is part of the Kubernetes provider for Pulumi, which provides the ability to deploy Kubernetes applications directly in a Pulumi program. In terms of data analysis, the author was already familiar with Python and its data analysis libraries, hence the decision to choose this technology.

The most important decision that had to be taken was choosing the most appropriate IaC tool. The main options available as of 2022 include Terraform, CloudFormation, Cloud deployment manager, Pulumi, Salt, Ansible. After an initial review, Pulumi was selected since it allows for the management of Cloud resources using the same provider as Terraform which is the industry standard, however, Pulumi uses programming languages to define the desired state of the infrastructure, instead of a template declarative way which is the approach of most of the other technologies. Furthermore, Pulumi is open source and not PCP specific, meaning the program can be extended to support multi-cloud deployments.

Another important decision was the selection of the PCP. This was primarily influenced by key factors or indicators such as performance and usability. In terms of performance, the creation of a Kubernetes cluster with comparable specification of GCP takes about three times less than on AWS. With regard to usability, GCP’s user interface is more intuitive and overall simpler to use than AWS. Hence, the decision to focus on GCP and particularly GKE managed Kubernetes engine for the cloud environment configuration and its performance analysis.

Table 4.1 provides a brief description of the purpose of each technology employed as part of this work.

### 4.2 Cloud configuration

In order to deploy cloud resources and manage infrastructure as code, it is necessary to grant permission and access to the Pulumi program as well as enable APIs to perform create/update/destroy operations on cloud resources. GCP uses projects as an abstraction to represent a set of configuration settings that define how an application interacts
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes</td>
<td>container orchestrator</td>
<td>Run test application, service mesh and monitoring</td>
</tr>
<tr>
<td>Istio</td>
<td>service mesh</td>
<td>Implement sidecar proxy pattern, zero trust security enabler</td>
</tr>
<tr>
<td>Docker</td>
<td>containers</td>
<td>Packages software applications running in Kubernetes</td>
</tr>
<tr>
<td>eBPF</td>
<td>networking</td>
<td>Candidate performance enhancer, substitute to IpTables</td>
</tr>
<tr>
<td>Prometheus</td>
<td>monitoring</td>
<td>Monitor system CPU and Memory during testing</td>
</tr>
<tr>
<td>Fortio test</td>
<td>advanced echo server</td>
<td>Simple echo server employed as test app</td>
</tr>
<tr>
<td>Fortio load</td>
<td>load testing library</td>
<td>Generate specific query-per-second load</td>
</tr>
<tr>
<td>Helm</td>
<td>package manager for Kubernetes</td>
<td>Deploy Kubernetes applications via Pulumi</td>
</tr>
<tr>
<td>Pulumi</td>
<td>Modern Infrastructure as Code</td>
<td>Create, deploy, and manage infrastructure on any cloud</td>
</tr>
<tr>
<td>NodeJS</td>
<td>JavaScript runtime</td>
<td>Pulumi Node.js SDK to write cloud programs in JavaScript</td>
</tr>
<tr>
<td>Bash</td>
<td>shell and command language</td>
<td>Write scripts for configuration and tooling</td>
</tr>
<tr>
<td>Python</td>
<td>programming language</td>
<td>Data analysis using Pandas, Matplotlib and numpy libraries</td>
</tr>
<tr>
<td>Jupyter</td>
<td>interactive data science</td>
<td>Notebook for the interactive execution of data analysis</td>
</tr>
</tbody>
</table>

Table 4.1: Technology stack

with GCP services and what resources the application uses. The following code represents the bash script used for the creation of the GCP project, service account, roles and the enabling of cloud APIs to allow Pulumi IaC to create and manage the resources necessary for the Kubernetes cluster to operate and the deployment of containerized applications running in the cluster.

```bash
#!/bin/bash
set +e
PROJECT_ID=ztmc-srodi-work
SERVICE_ACCOUNT_ID=ztmc-sa
SERVICE_ACCOUNT_DESCRIPTION="Service account for Pulumi IaC"
ROLES=( "roles/compute.instanceAdmin.v1" "roles/container.admin" "roles/iam.serviceAccountUser" )
CLOUD_APIS=( "container.googleapis.com" "compute.googleapis.com" )
```
# default set PROJECT_ID

gcloud config set project $PROJECT_ID

login(){
    # the URL must be opened in a browser copy the verification code and
    # paste in the terminal
    gcloud auth login --no-launch-browser
}

project_create(){
    # Create gcloud project
    gcloud projects create $PROJECT_ID
}

service_account(){
    # create service account
    gcloud iam service-accounts create $SERVICE_ACCOUNT_ID \ 
    --description="$SERVICE_ACCOUNT_DESCRIPTION" \ 
    --display-name="$SERVICE_ACCOUNT_ID"
}

policy_binding(){
    role=$1
    gcloud projects add-iam-policy-binding $PROJECT_ID \ 
    --member="serviceAccount:$SERVICE_ACCOUNT_ID@$PROJECT_ID.iam.gserviceaccount.com" \ 
    --role=$role
}

create_sa_key(){
    # create a key
    gcloud iam service-accounts keys create $SERVICE_ACCOUNT_ID-key \ 
    --iam-account=$SERVICE_ACCOUNT_ID@$PROJECT_ID.iam.gserviceaccount.com
}

[!NOTE] You will have to enable billing for the project in order
        to proceed with enabling Cloud APIs
        Enabling billing MUST be done on the Gcloud console (browser)

cloud_apis(){
    action=$1 # enable / disable
    api=$2 # (i.e. container.googleapis.com, compute.googleapis.com)
    gcloud services $1 $api
}
login
project_create
service_account
create_sa_key

for i in "${ROLES[@]}"; do policy_binding $i; done
for i in "${CLOUD_APIS[@]}"; do cloud_apis enable $i; done

| Listing 4.1: GCP cloud configuration |

### 4.3 Infrastructure as Code

This section presents the technologies and the implementation process of the IaC project required for creating, configuring and testing the hypothesis outlined. The cloud infrastructure is based on GCP and Google Kubernetes Engine which is Google’s managed Kubernetes solution.

Figure 4.1 represents an high level description of the execution of the IaC program that defines the configuration of cloud resources via a static configuration file. The program is initiated by an actor which can be a physical person or a CI/CD server. In particular, figure 4.1 describes the creation of a GKE cluster as well as the deployment of monitoring solution, service mesh and test application. The advantage of using this approach is the opportunity to deploy multiple heterogeneous environments with the only requirement to change a configuration file to define environment specific parameters.

![Diagram of IaC flow](image)

**Figure 4.1: IaC flow**

The same project can also be extended to create a multi-cloud solution on heterogeneous networks employing AWS Elastic Kubernetes Service. However, for the purpose of testing the performance with an implementation of Zero Trust using a service mesh, it was
decided to employ a single CSP to minimize the dependency of networking outside the
cluster network, aiming to reduce the network communication over the public internet
while sending requests from the same geographical region within the same data center.

The following code represents the entry point for the IaC program, where we define
the dependencies (line 1-5) and create resources using modern programming language
constructs and data types, including arrays, objects and literals. This program creates
two instances of class GkeCluster with different configuration parameters, where the
first is deploying a standard GKE cluster (line 7) which serves as a performance testing
benchmark and the second one uses Advance Datapath with data plane Version 2 which
avails of eBPF networking instead of kube-proxy (line 8-17). The program then fetches
static configuration values (line 26-30) to conditionally create resources in the clusters.

```javascript
// testing framework for cloud-native Zero Trust

import { GkeCluster } from "./gke";
import { EchoServer } from "./apps/echoServer";
import { Prometheus } from "./apps/prometheus";
import { Istio } from "./apps/istio";
import { Config } from "@pulumi/pulumi";

const simpleGKE = new GkeCluster("simple-gke");
const ciliumGKE = new GkeCluster("cilium-gke", {
  deployChart: false,
  datapathProvider: "ADVANCED_DATAPATH",
  ipAllocationPolicy: {
    clusterIpv4CidrBlock: "",
    servicesIpv4CidrBlock: "",
  }
});

const clusters = [
  {id: 1, name: simpleGKE.name, provider: simpleGKE.provider},
  {id: 2, name: ciliumGKE.name, provider: ciliumGKE.provider},
];

let out = [];
const chartVersion = "1.12.0-beta.1";
const config = new Config("sys-config");
const deployApps = config.require("deploy-apps");
const loadBalancer = config.require("load-balancer");
const istio = config.require("istio");
const istioPolicies = config.require("istio-policies");
```
if (deployApps === 'true') {
    for (const cluster of clusters) {
        const prometheus = new Prometheus(`prometheus-simple${cluster.id}`, cluster.provider, []);

        if (istio === 'true') {
            const istioSystem = new Istio(`istio-${cluster.id}`, cluster.provider, chartVersion, cluster.name, []);
            const echoServerIstio = new EchoServer(`fortio-istio${cluster.id}`, cluster.provider, [], istioSystem);
            const echoServerGateway = echoServerIstio.createEchoServerIstioGateway(cluster.id, istioSystem, cluster.provider);
            out.push(istioSystem);

            if (istioPolicies === 'true') {
                const echoServerIstioPolicies = echoServerIstio.createIstioPolicies(cluster.id, istioSystem, cluster.provider);
            } else {
                const echoServer = new EchoServer(`fortio${cluster.id}`, cluster.provider, []);

                if (loadBalancer === 'true') {
                    const externalLoadBalancer = echoServer.createEchoServerExternalLB(cluster.provider);
                    out.push(externalLoadBalancer);
                }
            }
        }
    }
}

export const output = out;

Listing 4.2: IaC index.ts file

Within this program, each class represents an application to be deployed in Kubernetes, this includes: EchoServer, Prometheus and Istio. Each class contains a set of attributes which can be exported and outputted by the Pulumi program, an example can be the IP address of the external load balancer created at runtime and used during testing to send HTTP requests to the echo server running in the Kubernetes cluster from outside the cluster. Classes also have a constructor function where all the resources to be created by the Pulumi program are defined.
The following code is a Pulumi class example where attributes are declared and within a constructor a resource namespace is created. The following example refers to the implementation of Echo server class. Echo server act as the test application within the Kubernetes cluster, responding (echoing) to HTTP requests received from the load library tool.

```javascript
import * as k8s from '@pulumi/kubernetes'
import { IstioI } from '../interfaces';
export class EchoServer {

  public name: string;
  public appLabels: {};
  public k8sProvider: k8s.Provider;
  public deployment: k8s.apps.v1.Deployment;
  public service: k8s.core.v1.Service;

  constructor(name: string, k8sProvider: k8s.Provider, dependencies: k8s.helm.v3.Release[], istioSystem?: IstioI) {
    this.name = name;
    this.appLabels = { app: name };
    this.k8sProvider = k8sProvider;

    let dependents: k8s.helm.v3.Release[] = [];
    if(istioSystem){
      dependents = [...dependencies, istioSystem.istioIngress];
    }

    const namespace = new k8s.core.v1.Namespace(name, {
      metadata:{
        name: "fortio",
        labels: {
          "istio-injection": "enabled"
        }
      }
    }, { provider: k8sProvider, dependsOn: dependents, deleteBeforeReplace: true });

    ...

  }
}
```

**Listing 4.3:** Pulumi class example: Echo server implementation

The purpose of the IaC component is to provision a full cloud environment including the applications’ deployment with the ability to update or destroy after test completion.
This is an important requirement as the test environment creation is reproduced multiple times.

4.3.1 Zero Trust Security implementation

The objective in using a service mesh for the purpose of this research is to enforce security policies leveraging the sidecar proxy pattern implemented. Istio the technology of choice, which complements Kubernetes with a networking infrastructure layer consisting in a mesh of data-plane proxies which are configurable via the Istio control plane. Istio is deployed via IaC and the label "istio-injection": "enabled" shown in code example "Pulumi class example: Echo server” line 25-27 instructs Istio control plane to automatically deploy a sidecar proxy for each pod created within this namespace. Once the sidecar is deployed, the pod will be part of the service mesh and security policies can be enforced.

According to NIST [10], one of Zero Trust goals is to prevent unauthorized access to data and services coupled with making the access control enforcement as granular as possible. Hence, the ability to authenticate and authorize all requests is a strong requirement. The security policy configuration is done via EchoServer class in Pulumi program, this is because security policies are in this case only applied to the namespace for which echo server has been deployed (Fortio namespace). Below code snippet describes how policies are created / deployed in EchoServer class.

```javascript
createIstioPolicies(id: number, istioSystem: IstioI, k8sProvider: k8s.Provider) {
    const mtlsPolicy = new k8s.yaml.ConfigFile('PeerAuthentication{id}', {
        file: './apps/yamlsIstioPolicies/PeerAuthentication.yaml',
        transformations: [(obj: any) => {
            obj.metadata.name = obj.metadata.name + id;
        }]
    }, { provider: k8sProvider, dependsOn: [this.service, istioSystem.istioIngress]});

    const authorizationPolicy = new k8s.yaml.ConfigFile('Authorization{id}', {
        file: './apps/yamlsIstioPolicies/Authorization.yaml',
        transformations: [(obj: any) => {
            obj.metadata.name = obj.metadata.name + id;
        }]
    }, { provider: k8sProvider, dependsOn: [this.service, istioSystem.istioIngress]});
```
const authenticationPolicy = new k8s.yaml.ConfigFile('RequestAuthentication{id}', {
  file: './apps/yamlsIstioPolicies/RequestAuthentication.yaml',
  transformations: [(obj: any) => {
    obj.metadata.name = obj.metadata.name + id;
  }]
}, { provider: k8sProvider, dependsOn: [this.service, istioSystem.istioIngress] });

return {
  mtlsPolicy,
  authenticationPolicy,
  authorizationPolicy
}

LISTING 4.4: Istio policies creation in Pulumi EchoServer class

The Istio policies selected for the implementation of Zero Trust via authentication, authorization, and encryption, are: RequestAuthentication, AuthorizationPolicy and PeerAuthentication.

The RequestAuthentication policy is an origin authentication, also known as end-user authentication: verifies the original client making the request as an end-user or device. Istio enables request-level authentication with JWT validation. [9]. The yaml code below represents the policy definition which is applied to the Istio Gateway via label selector and defines the JWT issuer under jwtRules field. This policy will only allow requests to the Istio Gateway when a valid JWT token is presented, the JWT token issuer is specified in the jwtRules section of the policy.

apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
  name: "jwt-auth"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: gateway
  jwtRules:
    issuer: "testing@secure.istio.io"
The AuthorizationPolicy is Role-based Access Control (RBAC) and provides namespace-level, service-level, and method-level access control for services in an Istio Mesh [9]. The yaml code below represents the AuthorizationPolicy definition where requests without a valid token are rejected with a rule specifying a DENY action for requests without request principals, shown as notRequestPrincipals: ["*"]. Request principals are available only when valid JWT tokens are provided. The rule also specifies the destination of the request which in this case is the path */echo*. This rule therefore denies requests without valid tokens to path */echo*.

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "frontend-ingress"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: gateway
  action: DENY
  rules:
  - from:
    - source:
      notRequestPrincipals: ["*"]
    to:
    - operation:
      paths: ["/echo"]
```

The PeerAuthentication policy is used for service-to-service authentication as well as the authentication of the client making the connection [9]. The code below represents the PeerAuthentication policy to globally enable Istio mutual TLS in STRICT mode. While Istio automatically upgrades all traffic between the proxies and the workloads to mutual TLS, workloads can still receive plain text traffic. To prevent non-mutual TLS traffic for the whole mesh, a mesh-wide peer authentication policy is set with the mutual...
TLS mode set to STRICT. The mesh-wide peer authentication policy does not have a selector and is applied in the root namespace.[9]

```yaml
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
```

**Listing 4.7**: PeerAuthentication

### 4.4 Testing

The testing of the application is performed on a server running in the same geographical region as the Kubernetes cluster to minimize latency due to on-the-wire network communication over the public internet. The test server will run scripts to load test the echo server running in GKE, collect and aggregate all test data results, including system resources and HTTP request/response latency. Figure 4.2 describes the main functions of test VM.

![Figure 4.2: Test VM main functions](image)

The first operation involves the creation and configuration of the test server which is a VM running in the same GCP project as GKE cluster. The creation of the test VM is done manually on the GCP dashboard as this is a one off operation and does not require much intervention, while the installing of necessary libraries and tools on the VM is done via bash script once the VM is up and running with SSH access via standard
Testing framework for cloud-native Zero Trust

terminal. The primary dependencies are Pulumi, docker, Python and NodeJS. Bash script below represents the test VM bootstrap script used to download and install all necessary binaries for the execution of tests.

```bash
#!/bin/bash
apt update
curl -fsSL https://get.pulumi.com | sh
apt install -y build-essential
curl -fsSL https://deb.nodesource.com/setup_current.x | bash -
apt install -y nodejs
curl -LO "https://dl.k8s.io/release/$((curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl") install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
apt install -y ca-certificates curl gnu
gpg lsb-release
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/debian $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
apt install -y docker-ce docker-ce-cli containerd.io
groupadd docker
usermod -aG docker $USER
echo "you need to logout after completion of this script.."
echo
apt install -y wget python3 python3-dev python3-venv
wget https://bootstrap.pypa.io/get-pip.py
python3 get-pip.py
pip --version
```

**Listing 4.8:** Test VM bootstrap script

To support the security enabled by Istio service mesh, one must send authenticated and authorized HTTP requests to the echo server in order to receive a successful response.
Testing framework for cloud-native Zero Trust

from the server. This is achieved using JWT tokens as specified in the "RequestAuthentication" Istio policy.

The test execution is triggered by run-tests.sh which is a bash script that runs in the VM that iterative triggers the load-generator.sh script for a number of times according to the script specification. The following code snippet represents an example test run where we simulate 50 clients sending 800, 1000, 1200, 1400 and 1600 requests per seconds for a duration of 5min (300 seconds). This script passes the necessary parameter to the load script so that test results are named descriptively and multiple tests can be run sequentially without human intervention. The test executed by this code takes 2 Hours, 34 Minutes and 10 Seconds to complete since there are 2 timeouts set in the load script to allow external components to be up in sync, for example the monitoring server.

```
#!/bin/bash

cd ~/src/zero-trust-multi-cloud
source get/ctx-gke
source get/ctx-gke

clients=50
test_duration=300

for i in 800 1000 1200 1400 1600
do
    ./load/load-ext.sh $i $clients $test_duration
    default_VS_ciliumNoKubeProxy GKE_CILIUM 34.105.255.223:8080 $CTX_GKE_CLUSTER1
    ./load/load-ext.sh $i $clients $test_duration
    default_VS_ciliumNoKubeProxy GKE_DEFAULT 34.142.36.231:8080 $CTX_GKE_CLUSTER2
    ./load/load-ext.sh $i $clients $test_duration
    default_VS_ciliumNoKubeProxy GKE_CILIUM_ISTIO_NO_POLICIES 35.246.116.1 $CTX_GKE_CLUSTER1
    ./load/load-ext.sh $i $clients $test_duration
    default_VS_ciliumNoKubeProxy GKE_DEFAULT_ISTIO_NO_POLICIES 35.197.197.187 $CTX_GKE_CLUSTER2
    ./load/load-ext.sh $i $clients $test_duration
    default_VS_ciliumNoKubeProxy GKE_CILIUM_ISTIO_WITH_POLICIES 35.246.116.1 $CTX_GKE_CLUSTER1
    ./load/load-ext.sh $i $clients $test_duration
    default_VS_ciliumNoKubeProxy GKE_DEFAULT_ISTIO_WITH_POLICIES 35.197.197.187 $CTX_GKE_CLUSTER2
done
```
The script `run-tests.sh` acts as an intermediary worker to invoke the load generator script which is responsible for the actual execution of the tests, the collection of data from monitored system resources and test results. The next snippet represents the main function of load generator script.

```bash
fortio_test(){
    
    # port forward Prometheus to make it accessible
    # on http://localhost:9090
    kubectl port-forward \
    prometheus-prometheus-stack-kube-prom-prometheus-0 \
    --context $7 \
    -n monitoring 9090 &

    # give it a few sec
    sleep 10

    msg "START FORTIO SERVER FOR LOAD TESTING for $1"
    TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.9/security/tools/jwt/samples/demo.jwt -s)
    URL=$2 && QPS=$3 && THREADS=$4 && TIME=$5 && BENCHMARK=$6
    TODAY=$(date "+%Y-%m-%d")
    TIMESTAMP=$(date "+%Y-%m-%d_%H:%M:%S")
    TEST_DETAILS="QPS=$3_Threads=$4_Duration=$5"
    FILENAME="$1_$TIMESTAMP"
    DIR="test-results/$TODAY/$BENCHMARK/$TEST_DETAILS/$FILENAME"
    mkdir -p $DIR
    echo "$DIR" >> data-analysis/test-list.txt

    # promql queries
    cpu_query="sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate{cluster=""}) by (namespace)"
    memory_query="sum(node_namespace_pod_container:container_memory_rss{cluster=""}, container!=""\") by (namespace)"

    # collect resource data
    ( bash ~/src/zero-trust-multi-cloud/load/gen-results.sh $TIME "$cpu_query" "$CWD/$DIR" cpu ) &
    ( bash ~/src/zero-trust-multi-cloud/load/gen-results.sh $TIME "$memory_query" "$CWD/$DIR" memory ) &

    # load generator
    docker run --name test fortio/fortio load \
    -json - \
```
Testing framework for cloud-native Zero Trust

Listing 4.10: Main function of load generator script

The load generator script performs the following operations:

1. Locally expose the monitoring portal for the system resources analysis at cluster level while running the process in the background (line 19)

2. Set parameters passed to the test and used for the naming of test directories and files (lines 23-30)

3. Create test directory and add name to a list of tests to be used for data-analysis at a later stage (lines 31-32)

4. Define PromQL queries to extract memory and CPU consumption at cluster namespace level (lines 35-36)

5. Run supporting script gen-results.sh, responsible to send HTTP requests to monitoring server and retrieve the queries results. These processes run in the background. (lines 38-39)

6. Run the load test using a docker image for Fortio and passing the necessary parameters, including QPS, threads, time, URL as well as JWT token created in line 24. On completion, output results to a file in the test directory. (lines 42-45)

7. Clean up. First destroy the load test container (line 47), then after a timeout of 60 seconds, kill processes running in the background with the instruction.

Listing 4.11: Kill all background processes

The script accepts seven parameters namely: query per second (QPS), number of simulated concurrent clients (threads), test duration in seconds (duration), name of generic benchmark test (benchmark), URL of the load balancer to send requests to (URL) and Kubernetes cluster context (context) for the authentication to the cluster using Kubernetes command-line tool kubectl.
### 4.5 Data Aggregation

The output of the load test generated by Fortio server is in JSON format and stored to a local file. This is achieved with a standard output redirection, as described on line 36 of the Fortio test function of the load generator script.

Similarly, results of the monitoring server queries to analyze resource consumption at system level are aggregated in single files based on the specific test. This operation is executed by a bash script gen-results.sh which is described in the snippet below.

```bash
#!/bin/bash

if [ $# -ne 4 ]; then exit 127; fi

echo "Collecting resource data and generating results file

duration=$1
query=$2
dir=$3
resource=$4"

DURATION=$1
QUERY=$2
DIR=$3
RESOURCE=$4

python3 data-analysis/query_csv.py http://localhost:9090 "$QUERY" write-header
> "$DIR"/"$RESOURCE".csv

for i in $(seq 1 $1)
do
  # this is important, it must reflect
  # prometheus.prometheusSpec.scrapeInterval and prometheus.
prometheusSpec.evaluationInterval
```

<table>
<thead>
<tr>
<th>qps</th>
<th>threads</th>
<th>duration</th>
<th>benchmark</th>
<th>test_name</th>
<th>url</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>$2</td>
<td>$3</td>
<td>$4</td>
<td>$5</td>
<td>$6</td>
<td>$7</td>
</tr>
</tbody>
</table>
A Python program is used to print the result of a Prometheus query as comma-separated values (CSV) file. The script accepts three parameters: the URL of Prometheus server, the query to send to Prometheus server, and a ”write-header” parameter to instruct the script whether to print the CSV header or not. The output of the Python program is a CSV file which can be again stored locally in the same test directory as the system resource monitoring queries.

Figure 4.3 represents the test results’ directory structure where the root is the test date, followed by the benchmark name, test description including parameters, cluster description. Within the nested structure there are three files with the actual test results: cpu.csv, latency.json and memory.csv.
4.6 Data Analysis

The analysis of data is performed with Python and data analysis libraries including Pandas, NumPy and MatPlotLib which are used to create a DataFrame data structure with test results, process the data as DataFrame, and plot results in tables/graphs. The python program reads the test results files saved in JSON and CSV formats from the test results directory and parses all files extracting and manipulating data based on the analysis requirements. The test parameters and description are included in the directories and file names.

The analysis requirements include the evaluation of the following statistical values:

1. Median (MED)
2. Mean (MEAN)
3. Standard Deviation (STD)

The latency analysis does not take into consideration the median value since the standard deviation is minimal and consistent across all test runs.

Prior to the execution of the data analysis, a dictionary is created where the key corresponds to the test name and the value corresponds to the path of test file. Code snippet below describe this flow as well as the possibility to add filters to analyze only a subset of test results.

```python
with open(test_file_name, "r") as file:
    for line in file:
        stripped_line = line.strip()
        if filtering:
            if key_filter1 in stripped_line and key_filter2 in stripped_line:
                dic[stripped_line] = f'{test_base_dir}{stripped_line}'
            else:
                dic[stripped_line] = f'{test_base_dir}{stripped_line}'

Listing 4.14: Data analysis: test dictionary
```

Consequently, for each key/value part of the test dictionary we extract the test parameters from the test results’ path with the help of function "test specs".

```python
for key, value in dic.items():
    testPath = value
    testName = key
    benchmark, qps, duration, clients, clust, time = test Specs(testName)
```

Consequently, for each key/value part of the test dictionary we extract the test parameters from the test results’ path with the help of function "test specs".
The function "test specs" accepts one parameter (testName) and leverages regex to extract all test parameters part of testName which is the actual file path described in Figure 4.3. The function then returns all parameters in string format.

```python
import re

def test_specs(testName):
    res = re.search('.*/(.*)/(.*)/(.*)/(.*)_2021-12-17_(.*)', testName)
    benchmark = res.group(3)
    qps = re.search('QpS=(.*)_Threads', res.group(4)).group(1)
    duration = re.search('_Duration=(.*)', res.group(4)).group(1)
    clients = re.search('_Threads=(.*)_', res.group(4)).group(1)
    cluster = res.group(5)
    time = res.group(6)
    return benchmark, qps, duration, clients, cluster, time
```

The Python script is responsible for cleaning test results data, the example below represents a code snippet of the CPU data analysis where the data is read from the CPU test results file (line 1), a Pandas DataFrame with a columns’ definition is then created to ease the manipulation of data. The timestamp is then formatted (line 3), some unnecessary data (representing namespaces not part of the performance analysis and data points not representative, explained later) is removed (line 7 and line 8), plots are created and outputted after grouping by namespace (lines 12-13).

```python
data = pd.read_csv(f'{testPath}/cpu.csv')
df = pd.DataFrame(data, columns = ['timestamp','value', 'namespace'])
df['timestamp'] = df['timestamp'].apply(lambda x: datetime.datetime.fromtimestamp(x).strftime('%H:%M'))
# create subplots
fig, ax = plt.subplots()
# remove monitoring namespace
df = df[df.namespace != 'monitoring']
df = df[df.namespace != 'cilium-test']
df = df[['CPU']] # clean data (remove datapoints before and after test)
evaluate(df,1,cluster,'cpu')
# group
for key, grp in df.groupby(['namespace']):
```
The "clean" function represented in line 9 above is responsible for the removal of data points which are positioned between the start of recording and start of the load test or between the end of load test and the end of the recording. In order to quantify the data points falling in this range, a timeout was included prior load test and post load test to maintain consistency across all test runs. The code snippet below describes the "clean" function which takes 2 parameters: the DataFrame and test name (CPU or Latency) and outputs a DataFrame which does not include the non-representative data points. For debug purposes the function also print out the number of data points prior and post cleaning.

```
def clean(df, name):
    print(f'{name} datapoints before cleaning', df.shape[0])
    df = df.iloc[50:-90]
    print(f'{name} datapoints after cleaning', df.shape[0])
    return df
```

The "evaluate" function represented in line 10 above, is responsible to provide the statistical results related to the test under analysis. The code snippet below describes the "evaluate" function which accepts 4 parameters: a DataFrame with the test results, the exponential in the case we want to represent a value on a different scale (seconds, milliseconds, nanoseconds, etc.), the cluster name and resource name (cpu or memory). The function firstly creates 3 DataFrames out of each namespace to be analyzed (Fortio, Istio System and Kube System) in lines 2-4, then for each resulting DataFrame it evaluates the median, average and standard deviation dividing by the given exponential (in this example 1 as we are measuring the CPU time in second and the metric provide us with the value already in seconds).

```
def evaluate(df, exp, cluster, resource):
    df1 = df[df.namespace == 'fortio']
    df2 = df[df.namespace == 'istio-system']
    df3 = df[df.namespace == 'kube-system']
    dataTable.loc[cluster, f'fortio-MEDIAN-{resource}'] = df1['value'].median() / exp
```
For the analysis of latency results, the percentiles are already part of the load server output, hence the requirement is limited to the extraction of the results from the test results files and the plotting of data in a graph.

### 4.6.1 Test results aggregation

The list of test run directories is stored in a text file which is read line by line at the execution of the data-analysis.py script, as describe in Listing 4.14. Pandas data frames are created using a test identifier as index. The test identifier contains the test and cluster configuration as well as a timestamp. Test results for a given test configuration are then aggregated using the following functions.

```python
def add_value_to_row(test_index, index, dfAgg, dfToAdd, columns):
    index = index.replace('GKE_','_')
    index = index.replace('CILIUM', 'eBPF')
    index = index.replace('DEFAULT', 'kp')
    index = index.replace('ISTIO_WITH_POLICIES', 'SM_ZT')
    index = index.replace('ISTIO_NO_POLICIES', 'SM_NP')
    if index not in dfAgg.index:
        for column in columns:
            dfAgg.loc[index, column] = dfToAdd[column][test_index]
    else:
        for column in columns:
            dfAgg.loc[index, column] = dfToAdd[column][test_index] + dfAgg[column][index]
```

Listing 4.19: Data analysis: evaluate() function
if index in count_dict.keys():
    count_dict[index] += 1
else:
    count_dict[index] = 1

def aggregate(df, cluster, resource):
    res = re.search('(.*)/(.*)/(.*)', str(cluster))
    index = str(res.group(1)) + '-' + str(res.group(2))
    columns = ['fortio-MEDIAN', 'fortio-AVG', 'fortio-StDev', 'istio-system-MEDIAN', 'istio-AVG', 'istio-StDev', 'kube-system-MEDIAN', 'kube-system-AVG', 'kube-system-StDev', 'tot']
    if resource == 'cpu':
        add_value_to_row(cluster, index, aggregate_CPU, df, columns)
    if resource == 'memory':
        add_value_to_row(cluster, index, aggregate_Memory, df, columns)
    if resource == 'latency':
        columns = ['Avg', 'std_dev']
        add_value_to_row(cluster, index, aggregate_Latency, df, columns)

Listing 4.20: Data aggregation: aggregate() function

4.6.2 Results plotting

As part of the data analysis some visualizations are created to represent data in bar charts and line charts. An example of the approach followed is shown below, where pandas and MatPlotLib are used for the plotting of a stacked bar chart which represents the overall resource utilization by namespace.

def plot_stacked_hbars(networking, name):
    aggregate_CPU[['fortio-AVG', 'istio-AVG', 'kube-system-AVG']].plot(kind='barh', stacked=True)
    plt.title(f'{name}-Overall CPU usage in s')
    plt.xlabel('s')
    plt.savefig(f'stack-{networking}-cpu-oh.png', bbox_inches='tight', dpi=100)

    aggregate_Memory[['fortio-AVG', 'istio-AVG', 'kube-system-AVG']].plot(kind='barh', stacked=True)
    plt.title(f'{name}-Overall Memory usage in MiB')
    plt.xlabel('MiB')
    plt.savefig(f'stack-{networking}-memory-oh.png', bbox_inches='tight', dpi=100)

    aggregate_Latency['Avg'] = aggregate_Latency['Avg'] * 1000
    aggregate_Latency[['Avg']].plot(kind='barh', stacked=True)
    plt.title(f'{name}-Overall Latency in ms')
Testing framework for cloud-native Zero Trust

Listing 4.21: Result plotting

```python
plt.xlabel('ms')
plt.savefig(f'stack-{networking}-latency-oh.png', bbox_inches='tight', dpi=100)
```
Chapter 5

Testing and Evaluation of cloud-native zero trust

The performance analysis of Zero Trust security configured in a Kubernetes environment deployed on PCP is conducted on a single cloud provider to minimize variance of test results due to the underlining configuration of hardware, virtualization and networking that different PCP offer. The configuration of the cloud environment, cluster, service mesh and deployment of applications is done with IaC practices to leverage definition files and avoid interaction with the PCP dashboards. In addition, IaC provides a repeatable deployment with the ability to create multiple clusters with several configurations based on parameters, loops and data structures. The programmatic creation of the infrastructure can prevent manual intervention and potential misconfiguration issues due to human error.

Initial tests were performed on clusters running in AWS and GCP with the configuration of a service mesh and deployment of security policies to enable Zero Trust while evaluating the resulting performance overhead at system level and application level (HTTP response / latency). Furthermore, the configuration of a cluster with eBPF networking was tested in order to unveil the possibility of enhance system performance avoiding networking requests between side-cars to go through the full Linux networking stack as explained in chapter 2. The results presented in this chapter relate to tests performed on GCP only.

In the next sections we discuss the methodology 5.1, KPI analysis 5.2, summary of test results 5.4 and evaluation of results 5.5.
5.1 Test methodology

Load generation is performed with Fortio [54] load testing library which runs at a specified query per second (QPS) and records an histogram of execution time. As discussed in Test Application subsection of chapter 3.5.3, Fortio was chosen amongst other load generator tools for being lightweight, featureful and provides the ability to act as both server and echo client. Fortio also calculates percentiles (e.g. p99 is the response time such as 99 percent of the requests take less than that number (in seconds, SI unit)). Fortio is configured to run for a set duration, for a fixed number of calls, or until interrupted (at a constant target QPS, or max speed/load per connection/thread).

Testing is performed on a remote VM deployed in the same geographical region as the Kubernetes cluster and each test run requires a number of steps to be performed sequentially. Testing flow is described in list 5.1.

1. Run tests and generate test results
2. Push tests results to remote test branch via git and pull git branch locally
3. Perform data analysis with Python / Jupiter

The test machine (VM) is created and configured once, while the cloud environment configuration and/or the test configuration differs for each test. Table 5.1 summarizes the different cluster configuration, specifying Kubernetes engine version, the networking solution and whether service mesh is enabled / disabled.

<table>
<thead>
<tr>
<th>GKE engine</th>
<th>Networking</th>
<th>Service Mesh</th>
<th>Zero Trust</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.20.10-gke.1600</td>
<td>kube-proxy (iptables)</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>1.20.10-gke.1600</td>
<td>kube-proxy (iptables)</td>
<td>Istio</td>
<td>none</td>
</tr>
<tr>
<td>1.20.10-gke.1600</td>
<td>kube-proxy (iptables)</td>
<td>Istio</td>
<td>yes</td>
</tr>
<tr>
<td>1.20.10-gke.1600</td>
<td>GKE Dataplane V2 (eBPF)</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>1.20.10-gke.1600</td>
<td>GKE Dataplane V2 (eBPF)</td>
<td>Istio</td>
<td>none</td>
</tr>
<tr>
<td>1.20.10-gke.1600</td>
<td>GKE Dataplane V2 (eBPF)</td>
<td>Istio</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 5.1: Primary cluster configuration

Fortio load test library run in a docker container on the test VM and test configuration parameters are passed by the load generator script, as described in Figure 4.2 in the previous chapter. In order to find the optimal configuration in terms of simulated clients (threads) and query per second, we run a benchmark test as describes in the following code snippet. The test parameters for QPS=-1, clients=64 and time=30s are suggested by Fortio official documentation to identify the system saturation without providing a hard limit on QPS, where -1 represents the maximum value achievable.
The test produced results in the range of 600 to 2000 actual QPS. Fortio recommends setting the actual parameters based on 75 percent of the max QPS result of the benchmark test, which is 1500. Based on this result, the actual test configuration parameters are described in Table 5.2.

<table>
<thead>
<tr>
<th>Threads/ Clients</th>
<th>Duration (sec)</th>
<th>Query per Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>300</td>
<td>800</td>
</tr>
<tr>
<td>50</td>
<td>300</td>
<td>1000</td>
</tr>
<tr>
<td>50</td>
<td>300</td>
<td>1200</td>
</tr>
<tr>
<td>50</td>
<td>300</td>
<td>1400</td>
</tr>
<tr>
<td>50</td>
<td>300</td>
<td>1600</td>
</tr>
</tbody>
</table>

Table 5.2: Test configuration parameters

5.2 KPI analysis

The key performance indicators for the evaluation of the system performance are: CPU usage, memory usage and latency. Figure 5.1 represents a latency analysis by percentiles for a test load of 1000 queries per seconds, 50 concurrent clients and duration of 5 minutes. The benchmark is a default configuration of Kubernetes networking using kube-proxy/iptables and the use of eBPF networking to replace iptables.

5.2.1 Latency

The latency analysis is performed in accordance with parameters described in table 5.2 and the following instruction, which is executed by the load generator script in the test VM. This code will run a docker container provision with Fortio library which will perform a load test with the given command line parameters. Furthermore, the test results will be outputted in JSON format and the standard output redirected to a local file which is named based on the test run.

```
docker run --name test fortio/fortio load \
-jjson -qps 1 -c 64 -t 30s http://$URL/echo

>`test-results/$TODAY/$BENCHMARK/$TEST_DETAILS/$FILENAME/latency.json`
The results show lower latency for each percentile and average when using eBPF and zero trust implemented with a service mesh. Figure 5.1 also shows that inversely, the latency is lower for a standard kube-proxy / iptables networking when there is no service mesh configured.

The latency analysis also produces the following test run summary which will be aggregated in 5.4 when the overall test results will be presented. Figure 5.2 represents a test run summary where QPS were set at 1000 and the actual QPS are close to this number. In addition, the total number of requests and total number of successful requests are part of the test run summary. The test run summary include: a timestamp, QPS, duration, threads / clients, actual QPS, requests count, successful requests, maximum latency in seconds, minimum latency in seconds, average latency in seconds and standard deviation.

5.2.2 CPU

The CPU analysis is performed in accordance with parameters described in table 5.2 and the following instruction, which is executed by bash script generate results which
Testing and Evaluation of cloud-native zero trust

Figure 5.2: Latency results per test run

is triggered by the load generator script in the test VM. This command will execute a python program that sends an HTTP request to Prometheus server and outputs the query results in CSV format. The program execution requires three parameters: the URL of Prometheus server, the query to be sent to Prometheus and a header/no-header flag to either print the CSV headers when the CSV file is being generated.

```
python3 data-analysis/query_csv.py http://localhost:9090 "$QUERY" write-header \\
  > "$DIR"/"$RESOURCE".csv
for i in $(seq 1 $1)
do
  # this is important, it must reflect
  # prometheus.prometheusSpec.scrapeInterval
  # and prometheus.prometheusSpec.evaluationInterval
  sleep 2
  python3 data-analysis/query_csv.py http://localhost:9090 "$QUERY" no-header \\
    >> "$DIR"/"$RESOURCE".csv
done
```

Listing 5.3: Bash script triggering Python program that sends HTTP requests to Prometheus server

The data analysis for each test run is performed with a python program for which the code snippet of CPU data analysis is presented in the previous chapter. The results are plotted with Python library MathPlotLib and the function PyPlot. Figure 5.3 represents the results of a test run with 1000 QPS and the cluster configuration was GKE with Dataplane V2 (eBPF) and Istio without policies.

Fortio namespace represents the namespace for the echo server application running in the cluster, istio-system is the Istio controller namespace and kube-system is Kubernetes control plane namespace.
The results described in figure 5.3 show a generally steady CPU utilization for the echo server namespace where a sidecar proxy is deployed without security policies enforced, while the kube-system namespace has an evident increase in CPU time in the first minute of the test, due to the provisioning of additional Istio ingress gateways to support the load. Istio-system namespace is also steadily consuming CPU throughout the test, except for the first minute of the test, the assumption is that the configuration of the ingress gateways require some processing resources while new instances are provisioned.

Similarly to Latency analysis, in section 5.4 all test results will be aggregated, and overall test results will be represented.

5.2.3 Memory

The Memory analysis is performed following the same methodology described in the previous CPU subsection 5.2.2, with different parameters for QUERY and RESOURCE which represents respectively the Prometheus query to analyze Memory consumption by namespace and the identifier MEMORY which is utilized by the scripts to name the results file generated for each test run.

An example result for the memory analysis, generated with MathPlotLib library is described in figure 5.4. The test run was performed on a 1000 QPS load on a cluster configured with standard networking (kube-proxy / iptables) and Istio with security
policies for authentication, authorization and encryption. The plot describes how the utilization of Memory for the Fortio namespace and kube-system namespace is very steady, while for istio-system namespace there is a gradual increase in Memory utilization in the first minute of the test run, which is then maintained steady for the duration of the test. Istio ingress gateway is a load balancer operating at the edge of the mesh that receives incoming HTTP/TCP connections [9] and the assumption is that Memory is required to configure the horizontally scaled instances created during the initial stage of the test.

![Figure 5.4: Memory test run plot (kube-proxy)](image)

The plot in figure 5.5 represents a test run where the cluster is configured to use data plane V2 (eBPF) and Istio policies are enabled. This time the memory consumption for kube-system namespace is higher while the memory consumption for istio-system and fortio remains at a similar level to the test in figure 5.4, where kube-proxy was used instead of eBPF.

### 5.3 Data cleaning

Figures 5.3, 5.4 and 5.5 represent all data-points for each test set of every relevant namespace. However, between the recording of resource usage from Prometheus server and the actual start of the test there is a time frame of nearly a minute, similarly at test completion. This is due to the requirement of starting the monitoring server and load server at the start and the killing of all processes at the end of each test run. As
a consequence, it is required to clean the data-set by removing the first 50 data-points and the last 90 data-points which are not representing the state of the system under load.

Figure 5.6 represents cleaned results of a test run with 1000 QPS where cluster configuration is GKE / Dataplane V2 (eBPF) and Istio without policies. As opposed to the representation of figure 5.3, there is no initial increasing slope and decreasing slope in any of the data-sets representing individual namespaces. As a consequence, it is now possible to analyze the mean value based on a more realistic standard deviation which is representative of the test results set recorded while the system is under load.

It is also important to configure the data resolution after having cleaned the results data-set. This is necessary since there will be no more near zero recording which means that if the data-sets contains small values, for instance smaller than 0.1, the plotting should be represented on the same scale as a data-set containing larger values, such as 1. If different scale are used, the representation is not indicative.

5.4 Summary of test results

The aggregation of all test results is achieved with the grouping based on the cluster networking configuration (kube-proxy or eBPF), service mesh configuration and zero trust policies enabled or no polices.
A total number of 21.6 millions HTTP requests were analyzed. The simulated number of concurrent clients (threads) per each test run is set to 50 and the duration of each test run is 300 seconds, while the query per seconds range between 800 and 1600 as specified in table 5.2.

For the CPU and memory analysis, each table row include the test run number of query per seconds in the first column while the remaining 9 columns represent median, mean and standard deviation values for fortio, istio and kube-system namespaces respectively.

5.4.1 Latency test results

The Latency test results are aggregated by cluster networking configuration (kube-proxy versus eBPF) and the parameters analyzed are overall average of mean values and standard deviations.

Tables 5.3 and 5.4 represent the results for kube-proxy and eBPF configuration respectively. Each table row describes the aggregate results for test executed with the number of query per seconds specified in the first column. Table 5.3 summarizes the results based on kube-proxy and iptables cluster networking configuration. Results show a reduction of up to 1ms in average latency as the QPS is increased. In addition, the standard deviation is generally consistent across each configuration, with a maximum variance in standard deviation of 1.5% for test runs at 1200 QPS when Istio is deployed without security policies. The latency is increased by about 9 times when Istio is configured
and a further 10% when security policies are deployed. The standard deviation is also impacted by the configuration of a service mesh, up to 10 times the results obtained from default kube-proxy without service mesh.

<table>
<thead>
<tr>
<th>kube-proxy Service Mesh Zero Trust</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0224</td>
<td>0.0047</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0221</td>
<td>0.0050</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0213</td>
<td>0.0046</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0213</td>
<td>0.0045</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0199</td>
<td>0.0050</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kube-proxy Service Mesh no Policies</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0201</td>
<td>0.0048</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0200</td>
<td>0.0045</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0228</td>
<td>0.0061</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0191</td>
<td>0.0044</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0190</td>
<td>0.0044</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kube-proxy</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0024</td>
<td>0.0005</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0024</td>
<td>0.0005</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0023</td>
<td>0.0005</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0023</td>
<td>0.0004</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0023</td>
<td>0.0004</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Latency mean and standard deviation for Kube-Proxy clusters

Table 5.4 summarizes the Latency test results based on eBPF cluster networking configuration. The mean value is represented in seconds. Results show a variance of 1ms in average latency across aggregated test runs, with the exception for 1200 QPS when service mesh is configured, both with and without policies. In addition, the standard deviation is generally consistent across each configuration, again with the exception for test runs at 1200 QPS when Istio is deployed. The latency is increased by about 3 times when istio is configured and a further 10% when security policies are deployed. The standard deviation is also impacted by the configuration of a service mesh, up to twice bigger than results obtained from eBPF without service mesh. The latency results when service mesh is configured are improved when eBPF is configured, both in terms of average and reduced standard deviation. Kube-proxy is 4 to 7ms less performant then eBPF when Istio is deployed.
### Table 5.4: Latency mean and standard deviation for eBPF clusters

<table>
<thead>
<tr>
<th>eBPF Service Mesh Zero Trust</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>800</td>
<td>0.0155</td>
<td>0.0035</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.0168</td>
<td>0.0032</td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>0.0204</td>
<td>0.0060</td>
</tr>
<tr>
<td></td>
<td>1400</td>
<td>0.0162</td>
<td>0.0028</td>
</tr>
<tr>
<td></td>
<td>1600</td>
<td>0.0153</td>
<td>0.0035</td>
</tr>
<tr>
<td>eBPF Service Mesh no Policies</td>
<td>query per second</td>
<td>average</td>
<td>standard deviation</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>0.0139</td>
<td>0.0036</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.0145</td>
<td>0.0038</td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>0.0124</td>
<td>0.0022</td>
</tr>
<tr>
<td></td>
<td>1400</td>
<td>0.0149</td>
<td>0.0041</td>
</tr>
<tr>
<td></td>
<td>1600</td>
<td>0.0143</td>
<td>0.0036</td>
</tr>
<tr>
<td>eBPF</td>
<td>query per second</td>
<td>average</td>
<td>standard deviation</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>0.0036</td>
<td>0.0017</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.0038</td>
<td>0.0018</td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>0.0038</td>
<td>0.0018</td>
</tr>
<tr>
<td></td>
<td>1400</td>
<td>0.0038</td>
<td>0.0018</td>
</tr>
<tr>
<td></td>
<td>1600</td>
<td>0.0037</td>
<td>0.0018</td>
</tr>
</tbody>
</table>

#### 5.4.2 CPU test results

The CPU value represented in this subsection’s tables corresponds to the total CPU usage in seconds by namespace.

The table column titles represent the different Kubernetes namespaces that are monitored to analyze the service mesh **control-plane** (Istio), service mesh **data-plane** (Fortio) and **Kubernetes control-plane** (kube-system). The test application runs in Fortio namespace and is provisioned with a sidecar for each cluster configuration where Istio is deployed, with or without security policies.

Table 5.5 represents a summary of CPU test results with kube-proxy data-plane networking and service mesh without security policies. Results show an increase in CPU usage for Fortio and Istio namespace as the number of QPS increases. The standard deviation is consistent and low for Fortio namespace, between 0.4 and 0.5%. Istio namespace as a variance of standard deviation of up to 1.5% with results in the range of 2.4 to 3.9%. Kube-systems has a consistent use of CPU for aggregated test runs, with an anomaly shown for 1200 QPS where standard deviation is unusually low at 0.8%; however, the variance in standard deviation is minimal at 0.9% across aggregated test runs.
Table 5.5: CPU results - **kube-proxy networking Service Mesh no Policies**

Table 5.6 represents a summary of CPU test results with eBPF data-plane networking and service mesh without security policies. Results show a steady increase in CPU utilization for Fortio and Istio namespaces with not so stable standard deviation, which has a variance of up to 3.9% for Fortio namespace across aggregated results of test runs. The standard deviation distribution for Istio namespace has lower variance, up to 1.8%. Kube-system namespace produced less predictable results with a standard deviation of up to 10% and a variance in standard deviation of up to 3%. The CPU usage is comparable between kube-proxy and eBPF networking for Fortio and Istio namespace, however, it is up to 45% higher for kube-system namespace when eBPF is configured.

Table 5.7 represents a summary of CPU test results with kube-proxy data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show an increase in CPU usage for Fortio and Istio namespace as the number of QPS increases. The standard deviation is consistent and low for Fortio namespace, between 0.4 and 0.7% with an anomaly at 1000 QPS where standard deviation is 2.8%. Istio namespace as a variance of standard deviation of up to 2.4% with results in the range of 2.6 to 5%. Kube-systems has a consistent use of CPU and standard deviation.

Table 5.8 represents a summary of CPU test results with eBPF data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show an increase in CPU utilization for Fortio and Istio namespaces with a
Testing and Evaluation of cloud-native zero trust

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.369</td>
<td>0.368</td>
<td>0.004</td>
</tr>
<tr>
<td>1000</td>
<td>0.467</td>
<td>0.478</td>
<td>0.028</td>
</tr>
<tr>
<td>1200</td>
<td>0.550</td>
<td>0.550</td>
<td>0.004</td>
</tr>
<tr>
<td>1400</td>
<td>0.631</td>
<td>0.633</td>
<td>0.007</td>
</tr>
<tr>
<td>1600</td>
<td>0.729</td>
<td>0.731</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Table 5.7: CPU results - kube-proxy networking Service Mesh Zero Trust

generally stable standard deviation up to 1600 QPS, where Istio namespace reports nearly double standard deviation than 1400 QPS runs: 7% vs 3.9%. Kube-system namespace produced unpredictable results with a standard deviation of up to 12.6% and a variance in standard deviation of up to 11%. The CPU usage is comparable between kube-proxy and eBPF networking for Fortio and Istio namespace, however, it is again up to 45% higher for kube-system namespace when eBPF is configured.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.375</td>
<td>0.382</td>
<td>0.027</td>
</tr>
<tr>
<td>1000</td>
<td>0.501</td>
<td>0.499</td>
<td>0.021</td>
</tr>
<tr>
<td>1200</td>
<td>0.515</td>
<td>0.541</td>
<td>0.038</td>
</tr>
<tr>
<td>1400</td>
<td>0.662</td>
<td>0.651</td>
<td>0.031</td>
</tr>
<tr>
<td>1600</td>
<td>0.700</td>
<td>0.712</td>
<td>0.035</td>
</tr>
</tbody>
</table>

Table 5.8: CPU results - eBPF networking Service Mesh Zero Trust

Table 5.9 represents a summary of CPU test results with kube-proxy data-plane networking and no service mesh configuration hence no security policies. Results show an increase in CPU usage for Fortio namespace as the number of QPS increases. The standard deviation is consistent at 0.1% for Fortio namespace. Kube-systems has a consistent use of CPU and standard deviation of 0.9 to 1%.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.064</td>
<td>0.064</td>
<td>0.001</td>
</tr>
<tr>
<td>1000</td>
<td>0.082</td>
<td>0.082</td>
<td>0.001</td>
</tr>
<tr>
<td>1200</td>
<td>0.099</td>
<td>0.099</td>
<td>0.001</td>
</tr>
<tr>
<td>1400</td>
<td>0.118</td>
<td>0.117</td>
<td>0.001</td>
</tr>
<tr>
<td>1600</td>
<td>0.137</td>
<td>0.137</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Table 5.9: CPU results - kube-proxy networking (no service mesh)
Table 5.10 represents a summary of CPU test results with eBPF data-plane networking and no service mesh configuration hence no security policies. Results show an increase in CPU usage for Fortio namespace as the number of QPS increases. The standard deviation is consistent at 0.1% for Fortio namespace. Kube-system namespace produced stable results with a standard deviation of up to 1.9% and a variance in standard deviation of up to 0.8%. The CPU usage is lower in Fortio namespace when eBPF networking is configured, however, it is again up to 45% higher for kube-system namespace when eBPF is configured.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio MED</th>
<th>Fortio MEAN</th>
<th>Fortio STD</th>
<th>Istio MED</th>
<th>Istio MEAN</th>
<th>Istio STD</th>
<th>Kube-system MED</th>
<th>Kube-system MEAN</th>
<th>Kube-system STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.048</td>
<td>0.049</td>
<td>0.001</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.079</td>
<td>0.076</td>
<td>0.014</td>
</tr>
<tr>
<td>1000</td>
<td>0.060</td>
<td>0.060</td>
<td>0.001</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.075</td>
<td>0.078</td>
<td>0.014</td>
</tr>
<tr>
<td>1200</td>
<td>0.070</td>
<td>0.070</td>
<td>0.001</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.078</td>
<td>0.080</td>
<td>0.011</td>
</tr>
<tr>
<td>1400</td>
<td>0.081</td>
<td>0.081</td>
<td>0.001</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.078</td>
<td>0.077</td>
<td>0.019</td>
</tr>
<tr>
<td>1600</td>
<td>0.092</td>
<td>0.092</td>
<td>0.001</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.083</td>
<td>0.078</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Table 5.10: CPU results - eBPF networking (no service mesh)

5.4.3 Memory test results

The memory value represented in this subsection’s tables corresponds to the total amount of anonymous and swap cache memory (including transparent huge-pages), and it equals to the value of total Resident Set Size (RSS) from memory.status file in Linux. The RSS is the portion of memory occupied by a process that is held in main memory Random Access Memory (RAM). The memory value is represented in MeBiByte (MiB).

Table 5.11 represents a summary of Memory test results with kube-proxy data-plane networking and service mesh without any security policies. Results show a very steady use of Memory for Fortio and kube-system namespace with a low standard deviation of up to 0.1%. Istio namespace usage of memory is more unpredictable, with a standard deviation in the range of 2.2 to 5%.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio MED</th>
<th>Fortio MEAN</th>
<th>Fortio STD</th>
<th>Istio MED</th>
<th>Istio MEAN</th>
<th>Istio STD</th>
<th>Kube-system MED</th>
<th>Kube-system MEAN</th>
<th>Kube-system STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
<td>0.201</td>
<td>0.190</td>
<td>0.032</td>
<td>0.339</td>
<td>0.339</td>
<td>0.001</td>
</tr>
<tr>
<td>1000</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
<td>0.235</td>
<td>0.217</td>
<td>0.051</td>
<td>0.341</td>
<td>0.341</td>
<td>0.001</td>
</tr>
<tr>
<td>1200</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
<td>0.239</td>
<td>0.235</td>
<td>0.022</td>
<td>0.342</td>
<td>0.342</td>
<td>0.001</td>
</tr>
<tr>
<td>1400</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
<td>0.235</td>
<td>0.226</td>
<td>0.036</td>
<td>0.342</td>
<td>0.342</td>
<td>0.001</td>
</tr>
<tr>
<td>1600</td>
<td>0.051</td>
<td>0.051</td>
<td>0.000</td>
<td>0.234</td>
<td>0.220</td>
<td>0.048</td>
<td>0.341</td>
<td>0.341</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Table 5.11: Memory results - kube-proxy networking Service Mesh no Policies
Table 5.12 represents a summary of Memory test results with eBPF data-plane networking and service mesh without any security policies. Results show a very steady use of Memory for Fortio and kube-system namespace with a low standard deviation of up to 0.3%. Istio namespace usage of memory is generally increasing as per QPS, with a standard deviation in the range of 1.6 to 4.5%. eBPF and kube-proxy use a comparable amount of memory at Fortio and Istio namespace while eBPF uses about 30% more memory at kube-system namespace than kube-proxy networking.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.049</td>
<td>0.049</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.049</td>
<td>0.049</td>
<td>0.000</td>
</tr>
<tr>
<td>1200</td>
<td>0.049</td>
<td>0.049</td>
<td>0.000</td>
</tr>
<tr>
<td>1400</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
<tr>
<td>1600</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 5.12: Memory results - eBPF networking Service Mesh no Policies

Table 5.13 represents a summary of Memory test results with kube-proxy data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show a very steady use of Memory for Fortio and kube-system namespace with a low standard deviation of up to 0.1%. Istio namespace usage of memory is generally steady despite increase in QPS load and the standard deviation recorded is in the range of 0.8 to 4.9%.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.050</td>
<td>0.051</td>
<td>0.000</td>
</tr>
<tr>
<td>1200</td>
<td>0.051</td>
<td>0.051</td>
<td>0.000</td>
</tr>
<tr>
<td>1400</td>
<td>0.051</td>
<td>0.051</td>
<td>0.000</td>
</tr>
<tr>
<td>1600</td>
<td>0.051</td>
<td>0.051</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 5.13: Memory results - kube-proxy networking Service Mesh Zero Trust

Table 5.14 represents a summary of Memory test results with eBPF data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show a very steady use of Memory for Fortio and kube-system namespace with a low standard deviation of up to 0.3%. Istio namespace usage of memory is generally steady, with a standard deviation in the range of 0.5 to 4.2%. eBPF and kube-proxy use a comparable amount of memory at Fortio and Istio namespace.
while eBPF uses about 35% more memory at kube-system namespace than kube-proxy networking.

Table 5.14: Memory test results - **eBPF networking Service Mesh Zero Trust**

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
<tr>
<td>1200</td>
<td>0.049</td>
<td>0.050</td>
<td>0.000</td>
</tr>
<tr>
<td>1400</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
<tr>
<td>1600</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 5.15 represents a summary of Memory test results with kube-proxy data-plane networking and no service mesh configuration hence no security policies. Results show a very steady use of Memory for Fortio and kube-system namespace with a low standard deviation of up to 0.1%.

Table 5.15: Memory test results - **kube-proxy networking (no service mesh)**

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1200</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1400</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1600</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 5.16 represents a summary of Memory test results with eBPF data-plane networking and no service mesh configuration hence no security policies. Results show a very steady use of Memory for Fortio and kube-system namespace with a low standard deviation of up to 0.1%. The use of memory for Fortio namespace is the same for eBPF and kube-proxy, while for kube-system the use of memory is 40% higher when eBPF networking is configured.

Table 5.16: Memory results - **eBPF networking (no service mesh)**

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1200</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1400</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>1600</td>
<td>0.008</td>
<td>0.008</td>
<td>0.000</td>
</tr>
</tbody>
</table>
5.5 Evaluation of test results

The summary of test results' section 5.4 show a low standard deviation for the exception to CPU test results where eBPF networking was enabled where the standard deviation reached 12.6 percent for Kubernetes control-plane on a 1400 query per second test run with zero trust configuration. The second-highest standard deviation is 8.8 percent for Istio control-plane on a 1600 query per second test run with zero trust configuration. The majority of the tests results show standard deviations below 5 percent with the greatest majority around 1 percent or lower.

The CPU consumption at data-plane level (Fortio namespace) is 8 to 9 times higher when a service mesh is configured and the cluster uses kube-proxy networking. The CPU consumption is up to 5 times higher at data-plane level (Fortio namespace) when eBPF networking is configured.

For both kube-proxy and eBPF networking implementation Memory consumption increases by 6 times at data-plane level (Fortio namespace) when a service mesh is configured.

Test results also show minimal variability of Memory consumption at data-plane level and service mesh control plane level, regardless of networking configuration being standard kube-proxy or eBPF networking. However, it appears that Kubernetes control plane is using up to 30 percent more memory when eBPF networking is enabled.

The evaluation of latency performance show that latency increases based on the configuration of a service mesh and further increases with the deployment of Zero Trust security policies. The increase in latency from a standard kube-proxy configuration to a kube-proxy with zero trust enabled can be up to 10 times larger.

In terms of eBPF networking, latency results show that the increase in latency when configuring a service mesh and security policies is not as large as with the kube-proxy, reaching a maximum latency increase of 5 times more when zero trust is enabled.

In accordance with test results, it is not possible to determine a scaling threshold at which a Zero Trust security performance degrades, since both latency results and physical resources results do not show sudden bursts with perpetual increase in resource consumption at any point, despite the higher number of query per seconds.

The latency performance results show a reduction in latency as the QPS increase. This is due to the horizontal pod auto-scaler (HPO) associated to the Ingress Gateway, which is deployed as part of the default Istio Helm chart in Kubernetes. The HPO is responsible for the provisioning of additional pods (horizontal scaling) when the CPU usage of
the gateway pod is above 80% of the CPU requested by the pod as configured in its deployment manifest. The default configuration is 100 m (equivalent of 0.1 CPU). The load balancing between Ingress Gateway pods is taken care of by a Kubernetes service part of the deployment, this is explained in further details in chapter 2 of the thesis under Kubernetes networking 2.4.1. It is worth mentioning that test automation scripts include a 60sec timeout after each test run, and test run have a duration of 5 minutes. This is important as the default value set in the HPO for the scaling-in of pods is 300 seconds. Hence, the autoscaling-in is always completed prior to each test run, which means the starting conditions are the same at each test run.

### 5.5.1 Latency overhead

Figure 5.7 represents a Python Pandas plot of the aggregated average latency utilization based on QPS test parameter and service mesh cluster details with kube-proxy networking: service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The distribution is not linear when service mesh is configured without security policies, there is a peak at 1200 QPS and consequently a decrease of latency for higher QPS (1400 and 1600). The distribution is linear when service mesh and zero trust security policies are deployed with a (despite minimal) reverse trend: the higher the QPS the lower the latency. Figure 5.7 shows a maximum latency overhead of 20ms when service mesh is configured without policies, and minimum overhead of 16ms when service mesh is configured with zero trust security policies.

![Figure 5.7: Overall latency overhead for kube-proxy networking](image)

Figure 5.7: Overall latency overhead for kube-proxy networking
Figure 5.8 represents a Python Pandas plot of the aggregated average latency utilization based on QPS test parameter and service mesh cluster details with cilium networking: service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The distribution is not linear according to the number of QPS. A peak is present at 1200 when zero trust is configured and contrarily, when no security policies are present the lowest point is again at 1200 QPS. Figure 5.8 shows a maximum latency overhead of 16ms when service mesh is configured and zero trust policies deployed, and minimum overhead of 8ms when service mesh is configured without security policies.

![Figure 5.8: Overall latency overhead for eBPF networking](image)

Figure 5.9 represents a Python Pandas plot of the aggregated average latency utilization based on QPS test parameter and service mesh cluster details with eBPF networking. The plot is a horizontal bar chart representation which includes the baseline test run for clusters configured without service mesh, as well as service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The aim of this plot is to visualize and quantify the performance overhead when service mesh is configured. Similarly, figure 5.10 represents a plot with the aggregated average latency utilization based on QPS test parameter and service mesh cluster details with kube-proxy.
Figure 5.9: Comparison overall Latency overhead for eBPF networking

Figure 5.10: Comparison overall Latency overhead for kube-proxy networking

5.5.2 CPU overhead

Figure 5.11 represents a Python Pandas plot of the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details with kube-proxy networking: service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The
Testing and Evaluation of cloud-native zero trust

aggregate includes the average utilization of each namespace: Fortio, Istio and kube-system. The distribution is linear for both service mesh without policies, and service mesh with zero trust security policies, the overhead in seconds increases with the number of QPS. Figure 5.11 shows a maximum overall CPU overhead of 1.4s when service mesh is configured with security policies, and minimum overhead of 0.6s when service mesh is configured without policies.

![Figure 5.11: Overall CPU overhead for kube-proxy networking](image)

Figure 5.12 represents a Python Pandas plot of the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details with eBPF networking: service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The aggregate includes the average utilization of each namespace: Fortio, Istio and kube-system. The distribution is linear for both service mesh without policies, and service mesh with zero trust security policies, the overhead in seconds increases with the number of QPS. Figure 5.12 shows a maximum overall CPU overhead of 1.5s when service mesh is configured with security policies, and minimum overhead of 0.6s when service mesh is configured without policies.

Figure 5.13 represents a Python Pandas plot of the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details with eBPF networking. The plot is an horizontal stacked bar chart representation which includes the baseline test run for clusters configured without service mesh, as well as service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The stacked bars include all namespaces under analysis: Istio control plane (istio-system), data-plane (Fortio), Kubernetes
Figure 5.12: Overall CPU overhead for eBPF networking

comparison overall CPU overhead for eBPF networking. The aim of the plot is to visualize and quantify the performance overhead overall and by namespace when service mesh is configured. Similarly, figure 5.14 represents a plot with the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details with kube-proxy. Each stack represents a namespace under analysis.

Figure 5.13: Comparison overall CPU overhead for eBPF networking
5.5.3 Memory overhead

Figure 5.15 represents a Python Pandas plot of the aggregated average Memory utilization based on QPS test parameter and service mesh cluster details with kube-proxy networking: service mesh zero trust “SM_ZT” and service mesh no policies “SM_NP”. The aggregate includes the average utilization of each namespace: Fortio, Istio and kube-system. The distribution is linear for both service mesh without policies, and service mesh with zero trust security policies, the overhead in seconds increases with the number of QPS. Figure 5.15 shows a maximum overall memory overhead of 0.28 MiB when service mesh is configured with security policies at 1400 QPS, and minimum overhead of 0.23 MiB when service mesh is configured without policies at 800 QPS.

Figure 5.16 represents a Python Pandas plot of the aggregated average Memory utilization based on QPS test parameter and service mesh cluster details with eBPF networking: service mesh zero trust “SM_ZT” and service mesh no policies ”SM_NP”. The aggregate includes the average utilization of each namespace: Fortio, Istio and kube-system. The distribution is linear for both service mesh without policies, and service mesh with zero trust security policies, the overhead in seconds increases with the number of QPS. Figure 5.16 shows a maximum overall memory overhead of 0.3 MiB when service mesh is configured with security policies at 1200 QPS, and minimum overhead of 0.23 MiB when service mesh is configured without policies at 800 QPS.
Figure 5.15: Overall Memory overhead for kube-proxy networking

Figure 5.16: Overall Memory overhead for eBPF networking

Figure 5.17 represents a Python Pandas plot of the aggregated average Memory utilization based on QPS test parameter and service mesh cluster details with eBPF networking. The plot is a horizontally stacked bar chart representation which includes the baseline test run for clusters configured without service mesh, as well as service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The stacked bars include all
namespaces under analysis: Istio control plane (istio-system), data-plane (Fortio), Kubernetes control-plane (kube-system). The aim of the plot is to visualize and quantify the performance utilization by namespace when service mesh is configured. Similarly, figure 5.18 represents a plot with the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details with kube-proxy.

**Figure 5.17:** Comparison overall Memory overhead for eBPF networking

**Figure 5.18:** Comparison overall Memory overhead for kube-proxy networking
5.6 Evaluation of upgraded system

The results obtained from the original test runs present some anomalies in standard deviation and mean values, this was particularly evident for tests with 1200 QPS load. This was the case for both kube-proxy and eBPF latency results. The latency variance is minimal, within 3 ms with a maximum standard deviation variance of 0.2%. However, further tests under an upgraded system were conducted to unveil any eventual issues with the test bed.

For this set of tests, the framework is exactly the same, the only upgrades applied are the following:

1. GKE version 1.21.15
2. Istio version 1.14.0

5.6.1 Latency test results

As per original test runs, the Latency test results are aggregated by cluster networking configuration (kube-proxy versus eBPF) and the parameters analyzed are mean value and standard deviation.

Tables 5.17 and 5.18 represent the results for kube-proxy and eBPF configuration respectively. Each table row describes the aggregate results for test executed with the number of query per seconds specified in the first column.

Table 5.17 summarizes the results based on kube-proxy and iptables cluster networking configuration. The average latency variability for kube-proxy without service mesh is 0.1ms, 0.9ms for kube-proxy with Service Mesh no policies and 3.4ms for kube-proxy with Service Mesh Zero Trust.

Table 5.4 summarizes the Latency test results based on eBPF cluster networking configuration. The mean value is represented in seconds. The average latency variability for eBPF without service mesh is 0.2ms, 0.7ms for eBPF Service Mesh no policies and 3.5ms for eBPF Service Mesh Zero Trust. There is a minimal difference of up to 0.2ms between kube-proxy and eBPF without service mesh and with service mesh no policies, while the difference is up to 3ms in favor of kube-proxy when security policies are deployed.

The standard deviation and averages for the upgraded system is higher but probably insignificant since the maximum standard deviation recorded is 0.7% and the maximum
<table>
<thead>
<tr>
<th>kube-proxy Service Mesh Zero Trust</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0203</td>
<td>0.0064</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0177</td>
<td>0.0050</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0191</td>
<td>0.0055</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0169</td>
<td>0.0048</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0180</td>
<td>0.0047</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kube-proxy Service Mesh no Policies</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0141</td>
<td>0.0042</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0142</td>
<td>0.0041</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0141</td>
<td>0.0039</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0141</td>
<td>0.0040</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0133</td>
<td>0.0032</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kube-proxy</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0028</td>
<td>0.0021</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0028</td>
<td>0.0021</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0027</td>
<td>0.0020</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0027</td>
<td>0.0020</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0027</td>
<td>0.0020</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eBPF Service Mesh Zero Trust</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0215</td>
<td>0.0070</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0195</td>
<td>0.0057</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0223</td>
<td>0.0066</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0201</td>
<td>0.0063</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0222</td>
<td>0.0070</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eBPF Service Mesh no Policies</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0144</td>
<td>0.0040</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0144</td>
<td>0.0040</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0145</td>
<td>0.0041</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0138</td>
<td>0.0036</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0143</td>
<td>0.0038</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eBPF</th>
<th>query per second</th>
<th>average</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.0030</td>
<td>0.0023</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.0030</td>
<td>0.0022</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>0.0028</td>
<td>0.0022</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0.0028</td>
<td>0.0021</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>0.0028</td>
<td>0.0022</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.17: Latency mean and standard deviation for each test run on Kube-Proxy clusters

Table 5.18: Latency mean and standard deviation for each test run on eBPF clusters
difference corresponds to 0.4% (eBPF Istio Zero Trust). Overall the results are more consistent in both average and standard deviation.

Figure 5.19 represents a Python Pandas plot of the aggregated average Latency utilization based on QPS test parameter and service mesh cluster details where service mesh zero trust is represented as "SM_ZT" and service mesh no policies as "SM_NP". The plot describes both type of networking configuration: eBPF and kube-proxy "kp". The distribution of results is consistent around 10-11ms for service mesh without policies while results are non-linear for service mesh with zero trust security policies, where the overhead in ms decreases as the number of QPS increases up to 1200 QPS and then results are unpredictable. Figure 5.19 shows a maximum overall latency overhead of 19ms when service mesh is configured with security policies on eBPF cluster at 1600 QPS, and minimum overhead of 10ms when service mesh is configured without policies on a kube-proxy cluster at 1600 QPS.

Figure 5.19: Overall Latency overhead

Figure 5.20 represents a Python Pandas plot of the aggregated average Latency utilization based on QPS test parameter and service mesh cluster details for both eBPF and kube-proxy networking. The plot is a horizontal bar chart representation which includes the baseline test run for clusters configured without service mesh, as well as service mesh zero trust "SM_ZT" and service mesh no policies "SM_NP". The aim of this plot is to visualize and quantify the performance overhead when service mesh is configured.
5.6.2 CPU test results

The CPU value represented in this subsection’s tables corresponds to the total CPU usage in seconds by namespace.

The table column titles represent the different Kubernetes namespaces that are monitored to analyze the service mesh control-plane (Istio), service mesh data-plane (Fortio) and Kubernetes control-plane (kube-system). The test application runs in Fortio namespace and is provisioned with a sidecar for each cluster configuration where Istio is deployed, with or without security policies.

Table 5.19 represents a summary of CPU test results with kube-proxy data-plane networking and service mesh without any security policies. Results show a steady increase in CPU utilization as the number of QPS increase for Fortio and Istio namespaces, while kube-system namespace is not affected by the increase in QPS. In terms of standard deviation, Fortio namespace show a variance of 0.4%, Istio namespace 1.2% and kube-system namespace 0.3%.

Table 5.20 represents a summary of CPU test results with eBPF data-plane networking and service mesh without any security policies. Results show a steady increase in CPU utilization as the number of QPS increase for Fortio and Istio namespaces, while
Testing and Evaluation of cloud-native zero trust

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.338</td>
<td>0.339</td>
<td>0.004</td>
</tr>
<tr>
<td>1000</td>
<td>0.410</td>
<td>0.420</td>
<td>0.006</td>
</tr>
<tr>
<td>1200</td>
<td>0.500</td>
<td>0.500</td>
<td>0.005</td>
</tr>
<tr>
<td>1400</td>
<td>0.572</td>
<td>0.572</td>
<td>0.006</td>
</tr>
<tr>
<td>1600</td>
<td>0.648</td>
<td>0.648</td>
<td>0.008</td>
</tr>
</tbody>
</table>

Table 5.19: CPU results - kube-proxy networking Service Mesh no Policies

kube-system namespace is not affected by the increase in QPS. In terms of standard deviation, Fortio namespace show a variance of 0.5%, Istio namespace 0.9% and kube-system namespace 2.7%. It is worth noting that kube-system namespaces uses 80/90% more CPU than with standard kube-proxy networking and both Fortio and Istio namespace show an increase of up to 10% in CPU usage when compared to standard kube-proxy networking. This is due to eBPF programs needed to be translated to byte code, as opposed to iptables which are part of the Linux Kernel and already compiled.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.352</td>
<td>0.352</td>
<td>0.004</td>
</tr>
<tr>
<td>1000</td>
<td>0.435</td>
<td>0.435</td>
<td>0.004</td>
</tr>
<tr>
<td>1200</td>
<td>0.523</td>
<td>0.522</td>
<td>0.006</td>
</tr>
<tr>
<td>1400</td>
<td>0.596</td>
<td>0.597</td>
<td>0.006</td>
</tr>
<tr>
<td>1600</td>
<td>0.676</td>
<td>0.675</td>
<td>0.009</td>
</tr>
</tbody>
</table>

Table 5.20: CPU results - eBPF networking Service Mesh no Policies

Table 5.21 represents a summary of CPU test results with kube-proxy data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show a steady increase in CPU utilization as the number of QPS increase for Fortio and Istio namespaces, while kube-system namespace is not affected by the increase in QPS, this is because kube-system (and kube-proxy) require lower computational effort while leveraging iptables implementation which is code already present in the Linux Kernel, hence code already compiled. In terms of standard deviation, Fortio namespace show a variance of 0.4%, Istio namespace 0.9% and kube-system namespace 0.3%.

Table 5.22 represents a summary of CPU test results with eBPF data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show a steady increase in CPU utilization as the number of QPS increase for Fortio and Istio namespaces, while kube-system namespace is not affected by the increase
Table 5.21: CPU results - **kube-proxy networking Service Mesh Zero Trust**

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th></th>
<th></th>
<th>Istio</th>
<th></th>
<th></th>
<th></th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
<td>MED</td>
<td>MEAN</td>
</tr>
<tr>
<td>800</td>
<td>0.335</td>
<td>0.335</td>
<td>0.003</td>
<td>0.450</td>
<td>0.458</td>
<td>0.028</td>
<td>0.063</td>
<td>0.063</td>
</tr>
<tr>
<td>1000</td>
<td>0.410</td>
<td>0.411</td>
<td>0.004</td>
<td>0.554</td>
<td>0.561</td>
<td>0.024</td>
<td>0.062</td>
<td>0.064</td>
</tr>
<tr>
<td>1200</td>
<td>0.487</td>
<td>0.488</td>
<td>0.004</td>
<td>0.650</td>
<td>0.657</td>
<td>0.026</td>
<td>0.063</td>
<td>0.063</td>
</tr>
<tr>
<td>1400</td>
<td>0.568</td>
<td>0.568</td>
<td>0.005</td>
<td>0.752</td>
<td>0.759</td>
<td>0.023</td>
<td>0.063</td>
<td>0.064</td>
</tr>
<tr>
<td>1600</td>
<td>0.635</td>
<td>0.636</td>
<td>0.007</td>
<td>0.842</td>
<td>0.848</td>
<td>0.026</td>
<td>0.062</td>
<td>0.062</td>
</tr>
</tbody>
</table>

In QPS. In terms of standard deviation, Fortio namespace shows a variance of 0.5%, Istio namespace 3% and kube-system namespace 7%. It is worth noting that kube-system namespaces use up to 90% more CPU than with standard kube-proxy networking and both Fortio and Istio namespace show an increase of about 4 to 10% in CPU usage when compared to standard kube-proxy networking. The reason for this difference is again due to the fundamental difference between eBPF and iptables where iptables functionalities are already pre-compiled and less CPU is required for its execution.

Table 5.22: CPU results - **eBPF networking Service Mesh Zero Trust**

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th></th>
<th></th>
<th>Istio</th>
<th></th>
<th></th>
<th></th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
<td>MED</td>
<td>MEAN</td>
</tr>
<tr>
<td>800</td>
<td>0.347</td>
<td>0.347</td>
<td>0.004</td>
<td>0.489</td>
<td>0.497</td>
<td>0.027</td>
<td>0.096</td>
<td>0.120</td>
</tr>
<tr>
<td>1000</td>
<td>0.424</td>
<td>0.424</td>
<td>0.004</td>
<td>0.607</td>
<td>0.614</td>
<td>0.025</td>
<td>0.094</td>
<td>0.119</td>
</tr>
<tr>
<td>1200</td>
<td>0.509</td>
<td>0.508</td>
<td>0.005</td>
<td>0.714</td>
<td>0.720</td>
<td>0.023</td>
<td>0.093</td>
<td>0.112</td>
</tr>
<tr>
<td>1400</td>
<td>0.590</td>
<td>0.591</td>
<td>0.005</td>
<td>0.824</td>
<td>0.831</td>
<td>0.027</td>
<td>0.094</td>
<td>0.114</td>
</tr>
<tr>
<td>1600</td>
<td>0.664</td>
<td>0.664</td>
<td>0.009</td>
<td>0.937</td>
<td>0.944</td>
<td>0.024</td>
<td>0.095</td>
<td>0.111</td>
</tr>
</tbody>
</table>

Table 5.23 represents a summary of CPU test results with kube-proxy data-plane networking and no service mesh configuration hence no security policies. Results show a steady increase in CPU utilization as the number of QPS increase for Fortio namespace, while kube-system namespace is not affected by the increase in QPS. In terms of standard deviation, Fortio namespace shows no variance, while kube-system namespace has a variance of 0.1%.

Table 5.24 represents a summary of CPU test results with eBPF data-plane networking and no service mesh configuration hence no security policies. Results show a steady increase in CPU utilization as the number of QPS increase for Fortio namespace, while kube-system namespace is not affected by the increase in QPS. In terms of standard deviation, Fortio namespace shows no variance, while kube-system namespace has a variance of 0.1%. The results of kube-proxy and eBPF without service mesh are quite
similar for Fortio namespace, while kube-system namespace is using about 15% more CPU with eBPF networking.

Figure 5.21 is a Python Pandas plot of the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details where service mesh zero trust is represented as “SM\textsubscript{ZT}” and service mesh no policies as “SM\textsubscript{NP}”. The plot describes both type of networking configuration: eBPF and kube-proxy ”kp”. The aggregate include the average utilization for each namespace: Fortio, Istio and kube-system. The distribution of results is linear as the overhead increases with the number of QPS and kube-proxy outperforms eBPF at every single aggregated test run. Figure 5.21 shows a maximum overall CPU overhead of 1.5 s when service mesh is configured with security policies on eBPF cluster at 1600 QPS, and minimum overhead of 0.5 s when service mesh is configured without policies on a kube-proxy cluster at 800 QPS.

Figure 5.22 is a Python Pandas plot of the aggregated average CPU utilization based on QPS test parameter and service mesh cluster details for both eBPF and kube-proxy networking. The plot is an horizontal stacked bar chart representation which includes the baseline test run for clusters configured without service mesh, as well as service mesh zero trust ”SM\textsubscript{ZT}” and service mesh no policies ”SM\textsubscript{NP}”. The stacked bars include all namespaces under analysis: Istio control plane (istio-system), data-plane (Fortio), Kubernetes control-plane (kube-system). The aim of the plot is to visualize

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td>MED</td>
<td>0.046</td>
<td>0.058</td>
<td>0.057</td>
</tr>
<tr>
<td>MEAN</td>
<td>0.046</td>
<td>0.058</td>
<td>0.057</td>
</tr>
<tr>
<td>STD</td>
<td>0.001</td>
<td>nan</td>
<td>0.010</td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.23: CPU results - kube-proxy networking (no service mesh)

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td>MED</td>
<td>0.047</td>
<td>0.059</td>
<td>0.072</td>
</tr>
<tr>
<td>MEAN</td>
<td>0.047</td>
<td>0.059</td>
<td>0.071</td>
</tr>
<tr>
<td>STD</td>
<td>0.001</td>
<td>nan</td>
<td>0.009</td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.24: CPU results - eBPF networking (no service mesh)
and quantify the performance overhead overall and by namespace when service mesh is configured.

The overall CPU overhead represented in Figure 5.21 provides a summary of the performance implications as a result of a service mesh and zero trust service mesh configurations. A discussion on the final results will be presented in the next chapter, however, it is clear that when an organization decides to utilize a service mesh for enhanced security, traffic control and observability, it is indispensable to take into account for additional resources. In terms of CPU, results show that in order to run a service mesh in a Kubernetes environment, 10 times additional CPU time is required. Moreover, in order to enable zero trust with service mesh security policies 15 times additional CPU time is required.

It is important to consider that resources provisioned via a PCP are much better optimized than in a proprietary physical data center. As a consequence, the requirement of additional CPU does not necessarily mean that additional physical CPU needs to be purchased consistently over time, but only when required. This is an important aspect to consider, since the on-demand model of the Public Cloud allow to be more efficient in resource utilization and spending.
5.6.3 Memory test results

The memory value represented in this subsection’s tables corresponds to the total amount of anonymous and swap cache memory (including transparent huge-pages), and it equals to the value of total Resident Set Size (RSS) from memory.status file in Linux. The RSS is the portion of memory occupied by a process that is held in main memory Random Access Memory (RAM). The memory value is represented in MeBiByte (MiB).

Table 5.25 represents a summary of Memory test results with kube-proxy data-plane networking and service mesh without any security policies. The results show a steady utilization of memory for Fortio namespace with a standard deviation of less than 0.1%. The use of memory for Istio namespace increases with the number of QPS. The kube-proxy namespace has a steady utilization as well as standard deviation which is consistent at 0.2%.

Table 5.26 represents a summary of Memory test results with eBPF data-plane networking and service mesh without any security policies. The results show a steady utilization of memory for Fortio namespace with a standard deviation of less than 0.1%. The use of memory for Istio namespace slightly increases with the number of QPS. The kube-proxy namespace has a steady utilization as well as standard deviation which is between
0.2 and 0.3%. The use of memory for Fortio and Istio namespaces is comparable between kube-proxy and eBPF networking, however, the utilization is about 30% higher for kube-system namespace.

Table 5.25: Memory results - kube-proxy networking Service Mesh no Policies

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio MED</th>
<th>Fortio MEAN</th>
<th>Fortio STD</th>
<th>Istio MED</th>
<th>Istio MEAN</th>
<th>Istio STD</th>
<th>Kube-system MED</th>
<th>Kube-system MEAN</th>
<th>Kube-system STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.188</td>
<td>0.179</td>
<td>0.025</td>
<td>0.331</td>
<td>0.331</td>
<td>0.002</td>
</tr>
<tr>
<td>1000</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.209</td>
<td>0.197</td>
<td>0.032</td>
<td>0.330</td>
<td>0.330</td>
<td>0.002</td>
</tr>
<tr>
<td>1200</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.232</td>
<td>0.220</td>
<td>0.034</td>
<td>0.331</td>
<td>0.330</td>
<td>0.002</td>
</tr>
<tr>
<td>1400</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.232</td>
<td>0.214</td>
<td>0.043</td>
<td>0.330</td>
<td>0.330</td>
<td>0.002</td>
</tr>
<tr>
<td>1600</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.231</td>
<td>0.221</td>
<td>0.034</td>
<td>0.331</td>
<td>0.331</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Table 5.26: Memory results - eBPF networking Service Mesh no Policies

Table 5.27 represents a summary of Memory test results with kube-proxy data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show minimal variance in utilization for Fortio namespace, with a standard deviation below 0.1%. Istio memory utilization is slightly reduced as QPS is increased, with a variance in standard deviation of up to 0.9%. The use of memory for kube-system namespace is consistent as well as its standard deviation of 0.2%.

Table 5.27: Memory results - kube-proxy networking Service Mesh Zero Trust

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio MED</th>
<th>Fortio MEAN</th>
<th>Fortio STD</th>
<th>Istio MED</th>
<th>Istio MEAN</th>
<th>Istio STD</th>
<th>Kube-system MED</th>
<th>Kube-system MEAN</th>
<th>Kube-system STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.230</td>
<td>0.222</td>
<td>0.028</td>
<td>0.329</td>
<td>0.329</td>
<td>0.002</td>
</tr>
<tr>
<td>1000</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.231</td>
<td>0.221</td>
<td>0.031</td>
<td>0.328</td>
<td>0.328</td>
<td>0.002</td>
</tr>
<tr>
<td>1200</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.218</td>
<td>0.206</td>
<td>0.033</td>
<td>0.330</td>
<td>0.330</td>
<td>0.002</td>
</tr>
<tr>
<td>1400</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
<td>0.214</td>
<td>0.207</td>
<td>0.022</td>
<td>0.329</td>
<td>0.329</td>
<td>0.002</td>
</tr>
<tr>
<td>1600</td>
<td>0.050</td>
<td>0.050</td>
<td>0</td>
<td>0.215</td>
<td>0.209</td>
<td>0.022</td>
<td>0.331</td>
<td>0.331</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Table 5.28: Memory results - kube-proxy networking Service Mesh Zero Trust

Table 5.28 represents a summary of Memory test results with eBPF data-plane networking and service mesh with security policies for authentication, authorization and encryption. Results show minimal variance in utilization for Fortio namespace, with a
standard deviation below 0.1%. Istio memory utilization is slightly reduced as QPS is increased, with a variance in standard deviation of up to 1.3%. The use of memory for kube-system namespace is consistent as well as its standard deviation of 0.2%. The effect of security policies deployment with eBPF results in a 45% increased of memory utilization from kube-system namespace when compared to kube-proxy networking configuration. While Fortio and Istio namespaces use a very similar amount of memory for both networking configurations.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.050</td>
<td>0.050</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
</tr>
<tr>
<td>1200</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
</tr>
<tr>
<td>1400</td>
<td>0.049</td>
<td>0.049</td>
<td>0</td>
</tr>
<tr>
<td>1600</td>
<td>0.050</td>
<td>0.050</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 5.28: Memory test results - eBPF networking Service Mesh Zero Trust**

Table 5.29 represents a summary of Memory test results with kube-proxy data-plane networking and no service mesh configuration hence no security policies. Results show no variance in utilization for Fortio namespace. The use of memory for kube-system namespace is consistent as well as its standard deviation at 0.2% for each QPS.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio</th>
<th>Istio</th>
<th>Kube-system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MED</td>
<td>MEAN</td>
<td>STD</td>
</tr>
<tr>
<td>800</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
<tr>
<td>1200</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
<tr>
<td>1400</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
<tr>
<td>1600</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 5.29: Memory test results - kube-proxy networking (no service mesh)**

Table 5.30 represents a summary of Memory test results with eBPF data-plane networking and no service mesh configuration hence no security policies. Results show no variance in utilization for Fortio namespace. The use of memory for kube-system namespace is consistent as well as its standard deviation between 0.1 and 0.2%. The utilization of memory by Fortio namespace is the same for eBPF and kube-proxy while for kube-system is about 40% higher when networking is configured with eBPF.

Figure 5.23 is a Python Pandas plot of the aggregated average Memory utilization based on QPS test parameter and service mesh cluster details where service mesh zero trust is represented as"SM_ZT" and service mesh no policies as "SM_NP". The plot describes
both type of networking configuration: eBPF and kube-proxy "kp". The aggregate include the average utilization for each namespace: Fortio, Istio and kube-system. The distribution of results is linear as the overhead increases with the number of QPS and kube-proxy outperforms eBPF at every single aggregated test run. Figure 5.23 shows a maximum overall Memory overhead of 0.3MiB when service mesh is configured with security policies on eBPF cluster at 1600 QPS, and minimum overhead of 0.2MiB when service mesh is configured without policies on a kube-proxy cluster at 800 QPS.

Table 5.30: Memory results - eBPF networking (no service mesh)

<table>
<thead>
<tr>
<th>Queries</th>
<th>Fortio MED</th>
<th>Fortio MEAN</th>
<th>Fortio STD</th>
<th>Istio MED</th>
<th>Istio MEAN</th>
<th>Istio STD</th>
<th>Kube-system MED</th>
<th>Kube-system MEAN</th>
<th>Kube-system STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.456</td>
<td>0.456</td>
<td>0.002</td>
</tr>
<tr>
<td>1000</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.456</td>
<td>0.456</td>
<td>0.001</td>
</tr>
<tr>
<td>1200</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.455</td>
<td>0.455</td>
<td>0.002</td>
</tr>
<tr>
<td>1400</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.455</td>
<td>0.455</td>
<td>0.002</td>
</tr>
<tr>
<td>1600</td>
<td>0.009</td>
<td>0.009</td>
<td>0</td>
<td>nan</td>
<td>nan</td>
<td>nan</td>
<td>0.457</td>
<td>0.457</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Figure 5.23: Overall Memory overhead

Figure 5.24 is a Python Pandas plot of the aggregated average memory utilization based on QPS test parameter and service mesh cluster details for both eBPF and kube-proxy networking. The plot is an horizontal stacked bar chart representation which includes the baseline test run for clusters configured without service mesh, as well as service
mesh zero trust "SM\textunderscore ZT" and service mesh no policies "SM\textunderscore NP". The stacked bars include all namespaces under analysis: Istio control plane (istio-system), data-plane (Fortio), Kubernetes control-plane (kube-system). The aim of the plot is to visualize and quantify the performance overhead overall and by namespace when service mesh is configured.

**Figure 5.24: Comparison overall Memory overhead**
Chapter 6

Discussion and Conclusions

This section presents a critical evaluation of the project, with a focus on the project objectives, accomplishments and any problems that affected this work. The objective of the research, as stated in chapter 3, included the design and development of a multi-cluster framework on PCPs, the design and development of a test environment, the evaluation of the performance at scale of the Zero Trust system and evaluation of the performance of a Zero Trust system with the addition of eBPF as a possible performance enhancer.

In addition to the research objectives, two hypothesis were formulated: “Given the scale of modern cloud computing systems is there a scaling threshold at which a Zero Trust security performance degrades?” and “Can the performance overhead of sidecar proxy implementation in a Zero Trust model be mitigated with programmable Linux kernel technologies?”.

The next sections outline a discussion of the research results, the conclusion drew from the results analysis and a discussion on the possible scope for the extension of this research with possible challenges to be faced.

6.1 Discussion

The objectives presented as part of the research proposal were achieved, with the primary objective being the evaluation of Zero Trust overhead in a cloud native system.

1. Design and development of a multi-cluster framework
2. Deployment of a test environment
3. Evaluation of performance of Zero Trust configuration at scale

4. Testing and evaluation of a possible performance enhancer with eBPF networking

To support the research objectives a multi-cluster environment was configured on both single PCP and multiple PCPs with the deployment of a multi-cluster service mesh. The performance evaluation was conducted on both multi-cluster multi-network, multi-cluster single-network and single-cluster deployment.

The evaluation presented in chapter 5 refers to a single cluster, single PCP deployment. It is important to note that during the course of the research it was proved that the same zero trust configuration can be extended to include multiple PCP vendors, this is possible using a multi-cluster multi-network service mesh configuration. Furthermore, a performance analysis on multi-cloud was also conducted, but the testing cases and scenarios were not comprehensive enough, as discussed in the conclusion of research paper ”Performance analysis of Zero-Trust multi-cloud” [51]. Nevertheless, this publication served as a stepping stone to gather peer-review feedback and critical analysis to define the requirements to further the research. In addition, the framework presented in the paper served as a starting point for the enhanced testing framework discussed in this dissertation.

The decision to use a single PCP was made in order to prevent variance of underling networking configuration, hardware components and other features that different PCPs offer. This issue was also discussed in the research paper ”Performance Analysis of Zero Trust multi-cloud” [51], where the aim for future tests was to allocate images with similar specifications for clusters memory and CPU cores. In practice, it is not a trivial task to define images with the same specifications from different PCPs, hence the decision to focus on a single cluster and avail of exactly the same baseline configuration.

The performance of the data plane and control plane, including service mesh and Kubernetes system control plane were evaluated as part of this work. It was important to separate control and data plane to identify the performance impact of the configuration of zero trust security policy at both architecture levels. The data plane level include the microservice application under test and the provisioning of a sidecar proxy for the enforcement of security policies. The proxies intercept and control all network communication between microservices. The service mesh (istio-system) control plane represents the components responsible for managing and configuring the data plane proxies. The Kubernetes (kube-system) control plane manages worker nodes and pods in the cluster.

The overall test results show a reasonably low standard deviation for all tests with exception made for eBPF networking at Kubernetes control plane level when a service
mesh is configured. The high standard deviation is believed to be the direct result of the provisioning of additional ingress gateways occurring at the start of the tests. Given that the service mesh configuration was exactly the same, the assumption is that iptables perform better than eBPF when provisioning multiples pods and services in a brief amount of time. Figure A.1 shows the big spikes in CPU consumption at Kubernetes control plane level when networking is eBPF (cilium) and service mesh is configured.

It was confirmed that the envoy proxy is very efficient in terms of resource consumption as the amount of CPU / memory required by the sidecar proxy, whether there are security policies or not, does differ by less than 1 percent on average. This conclusion is valid on all Fortio test results represented in tables 5.5, 5.7, 5.6, and 5.8, where service mesh, hence sidecar proxy was deployed.

The CPU utilization of the service mesh control plane is increased by around 90 percent when zero trust security policies are deployed. This is due to the control plane handling the API server requests and configuring the proxies acting as PEPs in the data plane. This has significant implications when deploying Zero Trust on resource constrained Kubernetes environments.

The memory consumption for the sidecar proxy does not differ whether security policies are deployed or not. However, the memory usage for Fortio namespace (test application) increases by 6 times when the service mesh is configured. This is expected as the additional proxy container needs to hold the proxy configuration and the memory consumption of the proxy depends on the total configuration state the proxy holds [55].

Kubernetes control plane uses about 30 percent more memory when networking configuration is eBPF as opposed to iptables and there is no noticeable variance of memory consumption when service mesh with/without policy is configured or standard no service mesh configuration. This can be viewed on the kube-system column of the memory analysis where networking is configured to be kube-proxy or eBPF. See comparison between no proxy (table 5.15 and table 5.16), comparison between proxy with service mesh (table 5.11 and table 5.12), and comparison between proxy and zero trust (table 5.13 and table 5.14). This indicates that, at least given the current Kubernetes architecture, eBPF networking is less efficient in terms of memory allocation, once again, this is likely to be associated with the fact that eBPF is compiled at execution time as opposed to iptables which is embedded and pre-compiled code part of the Linux kernel.

The latency variability between different tests, as shown in tables 5.3 and 5.4, is minimal. This can signify the amount of query per seconds and / or clients / threads could be further increased to visualize a degrading in performance when a certain threshold
is achieved. As a consequence it is not possible to validate or nullify Hypothesis 1 presented in section 3.2 of chapter 3. However, in testing for this hypothesis a number of important findings were uncovered, including the overall overhead as result of service mesh configuration, as well as the quantification of memory and CPU overhead, which have been discussed in the previous paragraphs.

During the evaluation of test results a number of anomalies were identified, particularly when the QPS was set to 1200. The majority of unexpected results are found in the latency analysis. The latency variance is most likely acceptable since we rely on the PCP’s network on top of the various virtualization layers. Nevertheless, other tests were conducted to rule out any possible oversight. However, one limitation when relying on PCP services and specifically managed Kubernetes distributions, is that PCPs offer support for a specific version only for a limited amount of time after the initial version release. This forced us to upgrade to a most recent GKE version since the original tested version is no longer supported by GCP.

The re-evaluation under the upgraded system version also included an upgraded Istio service mesh version. The results were surprising in that eBPF was no longer outperforming kube-proxy in terms of latency, however, the results were considerably more consistent in terms of reduced standard deviation and reduced variability of results across test runs and aggregated results. The assumption is that both GKE and Istio upgrades resulted in more software stability, better security, at the expense of performance, particularly for latency when eBPF networking is configured. It is important to note that the utilization of memory and CPU was consistently improved, particularly for kube-proxy networking configuration.

### 6.2 Conclusion

The conclusions reported in this section refer to the original evaluation of the system, where clusters version was set to GKE 1.20 and Istio service mesh 1.12.0.

The background and literature review provided us with sufficient elements to confirm the side-car proxy pattern and the service mesh implementation are suitable candidates for enabling zero trust in cloud native systems, particularly in a Kubernetes / containers environment. The opportunities for embracing this approach to improve organizations security postures is large, given the flexibility of both Kubernetes and service mesh, which can be deployed and configured on private clouds, public clouds, hybrid clouds and can operate on containers and or VMs workloads.
The primary conclusion is that latency overhead when zero trust is configured via service mesh with default kube-proxy is between 15ms and 19ms (figure 5.7). eBPF is generally outperforming kube-proxy, but the results show that performance is not predictable with average latencies between 8ms and 16ms (figure 5.8).

The CPU time overhead, in a kube-proxy cluster, is up to 1.4s when security policies are applied and up to 1.1s without security policies. On eBPF cluster, the overhead is up to 1.5s with policies and up to 1.2s without policies. It is also found that the increase in latency overhead is linear as the QPS increases whereby the latency overhead nearly doubles as the QPS doubles (figures 5.11 and 5.12). In addition, the data-plane is utilizing approximately 5 times the amount of CPU when Istio service mesh is configured, the service mesh control plane is utilizing between 0.2s and 0.8s CPU while Kubernetes control plane is not affected by the deployment of a service mesh (figures 5.14 and 5.13).

The memory analysis shows a linear increase in memory required by service mesh whereby the average consumption recorded is between 0.22 MiB and 0.29 MiB. Minimal difference was found between tests run on kube-proxy and eBPF clusters (figure 5.15 and 5.16). In terms of the overhead by namespaces when Istio service mesh is deployed, results show that the data-plane requires approximately 6 times the amount of memory and service mesh control plane utilizes between 0.25 MiB and 0.27 MiB of memory (figures 5.18 and 5.17.)

A secondary conclusion is that the Kubernetes ecosystem and the CNCF are continuously expanding, and currently a particular focus is on eBPF as enabler of security and observability in containerized environments. Multiple open-source projects and startup companies are very active in the areas of service mesh, Kubernetes networking and eBPF. Showing the importance and relevance to solve the security issues related to distributed systems and particularly the de-facto standard of container orchestrators: Kubernetes. To name a few companies: Cilium (open source), Tigera (open source calico, CNI, eBPF), Isovalent (open source Cilium, CNI, eBPF), Tetrate (Istio, service mesh), Solo (Istio, service mesh), Falco (open source / eBPF) and SysDig (open source Falco, eBPF).

In terms of the problem description, the evaluation of system performance was conducted rigorously and data was categorized and analyzed based on system set up to include networking and service mesh configuration. Test results show that the configuration of a service mesh introduces an additional control plane component for the service mesh operation and an additional container per each pod in the cluster to act as PEP, resulting in an addition memory utilization of almost 100% for kube-proxy networking. Also, the recorded CPU time increase is up to 8 times higher.
Discussion and Conclusions

Test results also show that when configuring a service mesh, latency performance results distribution is close to 20ms when the networking configuration is kube-proxy / iptables as opposed to 15ms with eBPF networking. This is an important performance improvement and the validation of Hypothesis 2 presented in section 3.2 of chapter 3: “Can the performance overhead of sidecar proxy implementation in a Zero Trust model be mitigated with programmable Linux kernel technologies?”.

Another important conclusion is that eBPF is less performant in terms of CPU usage when Kubernetes control plane is provisioning new pods and services as opposed to kube-proxy configuration. Additional tests under various scenario are required to quantify the performance loss with eBPF as opposed to kube-proxy. It is hoped the extensive test results presented in this work will inform other practitioners in the field or some such.

6.3 Future Work

In the next sub-section we discuss some possible enhancements and opportunities to expand this research in the future. Future work items could include the following:

1. Evaluate at larger scale
2. Evaluate alternative technologies
3. Evaluate alternative communication protocols

6.3.1 Evaluate at larger scale

The evaluation of the system was conducted on three different deployment models: single cluster single PCP, multi-cluster single PCP and multi-cluster multi PCP. In all the evaluation, the maximum number of worker nodes per cluster was limited to two as it was not necessary for the type of test conducted to have additional nodes since the load was handled comfortably without an evident performance degradation in terms of system resource (i.e. not enough CPU or Memory).

However, the scale of the system could be increased at both application level and infrastructure system level. For example: the analysis could be performed on a real-life microservice application where responsibility for each microservice are well-defined and decoupled from other services. This type on analysis can provide the opportunity to analyze microservices components that run in-cluster and are never exposed externally. In addition to the scale of the microservice application after test, the scale of the underlying infrastructure could be increased to support higher load. Finally, the load test
could be performed with a simulation of greater number of concurrent clients, QPS and possibly for longer time.

In a recent article by Kebe Liu et Al. [56], results show eBPF outperforming kube-proxy when the number of connections/clients was increased to 100, where eBPF was reaching almost 1200 QPS as opposed to kube-proxy reaching 1000 QPS, in addition the latency was about 15 percent lower on the eBPF configuration. The performance analysis of Kebe Liu et Al. [56] did take in consideration only latency. Future work could include the resource analysis for similar configuration, with an increase of clients and no maximum QPS set.

One of the limitations in performing larger scale tests has been the access to PCP services, it is likely necessary to obtain a large amount of PCP credits to perform a more comprehensive analysis.

6.3.2 Evaluation of alternative technologies

There are opportunities to further expand this research with a performance analysis of the system under different environment configurations, to include testing multiple CNIs, multiple service mesh / proxies, multiple managed Kubernetes solutions VS Kubernetes on bare metal. Istio was chosen as a reference service mesh since it was determined to be the best supported, most featureful technology with the greatest level of adoption within the industry. However, multiple service mesh offerings have emerged over the years, this includes Linkerd[57], Kong Mesh[58], Consul[59] and more. In addition, work is currently underway as part of the Cilium project to create an eBPF-based service mesh[60], an interesting project which further validates this research.

Furthermore, the testing framework could be expanded to improve the automation of the testing VM deployment to ease the execution of test runs under multiple load test configuration, particularly, not only increasing the number of QPS with the same concurrent clients (50) and test duration (300s), but also testing the effect on the system when increasing the number of clients or test duration while keeping the same QPS. This could be achieved with automated pipelines which leverage configuration technologies to prepare the test environments and iterations execute tests which could run unsupervised.

6.3.3 Evaluate alternative communication protocols

In addition, the networking aspect within the cluster (east-west traffic) could be further analyzed and experiments with different networking protocols / technologies could be evaluated. In microservices architecture the magnitude of east-west traffic is much larger
than north-south traffic. This is because the application components are broken into smaller services (microservices) which need to communicate creating a highly cohesive and low coupling environment. An important enhance in performance can be gained if the networking communication between microservices in the cluster is more efficient in terms of latency and resource consumption. Google Remote Procedure Call (gRPC) is an open source remote procedure call system based on HTTP/2 and protocol buffers as the interface description language. gRPC can improve the communication between microservices by lowering the bandwidth utilization and latency offered by RESTful APIs[13]. Kumar P.K. et al. [61] proposed a performance analysis of REST, gRPC, and Thrift communication protocols, in terms of their network, memory, CPU utilization, and response time. The analysis show that Thrift and gRPC outperform REST for microservices east-west communication.

Some initial work from Google and Istio community was done to create a gRPC service mesh which does not use a sidecar proxy, resulting in improved performance offered by gRPC communication and further improvement in performance due to the avoidance of the sidecar proxy component. Initial test results where presented in a recent article "gRPC Proxy-less Service Mesh" on the official blog on Istio website [9]. Initial results show three times less usage of memory on both server and client, and practically no use of CPU on both server and client. Furthermore, the latency comparison shows proxy-less gRPC performing 15 to 20 times better than envoy proxy. Test cases where not comprehensive enough to draw conclusions as only mTLS policies were applied. However, the results look very promising. In addition, it is important to note some important limitations of this implementation: some security policies are not yet supported, not all features of xDS are available in gRPC and the application code needs to be instrumented to support gRPC.


Discussion and Conclusions


Discussion and Conclusions


[53] Prometheus Authors. From metrics to insight. [Online]. Available: https://prometheus.io/


[57] Linkerd Authors. Linkerd: A different kind of service mesh. [Online]. Available: https://linkerd.io/


Appendix A

Data analysis

Figure A.1 represents all plots for each system configuration for a test with 1400 query per seconds.
Figure A.1: CPU plots by system configuration at 1400 qps