

An Exploration of Chaos Engineering Techniques on a Self-Healing Cloud Native Microservice Architecture



Bruno Franco

A dissertation submitted in partial fulfilment of the requirements of
Technological University Dublin for the degree of
M.Sc. in Computing (Advanced Software Development)

January 2024

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advanced Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Technological University Dublin and has not been submitted in whole or part for an award in any other Institute or University.

The work reported in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: _____

Date:

ABSTRACT

This dissertation delves into the realm of cloud-native microservice architectures with a focus on self-healing mechanisms, investigating their response to chaos engineering techniques. In an era where cloud-based applications demand resilience and cost-effectiveness, understanding the behavior of self-healing architectures under chaotic conditions is of paramount importance.

The primary research objective of this study is to explore how a cost-effective self-healing cloud-native microservice architecture reacts when subjected to chaos engineered fault injections. By simulating real-world disruptive scenarios, the key aim is to provide valuable insights into the architecture's ability to maintain operational integrity and recover gracefully.

Key findings from our research indicate that while auto-scaling warm pools have been widely advocated to bolster resilience, their actual impact on aiding a cloud architecture's recovery from chaos engineered fault injections is less impactful than hitherto claimed. This study contributes to the ongoing discourse on self-healing microservice architectures, offering practical implications for architects, developers, and organizations striving to enhance the robustness and reliability of their cloud-native applications.

The results of this research not only deepen our understanding of self-healing cloud architectures but also underscore the need for a comprehensive approach to resilience, encompassing aspects beyond mere scalability. This dissertation serves as a valuable resource for professionals and researchers engaged in cloud-native system design and chaos engineering, providing essential insights for building more resilient, cost-effective, and adaptable systems.

Key words: Chaos Engineering, microservices, cloud services, self-healing system

ACKNOWLEDGEMENTS

I am very grateful to my supervisor Damian Gordon, for his excellent guidance throughout the production of this dissertation. Damian was extremely friendly, approachable, helpful, and flexible during our journey together. It was an absolute pleasure to work with him, and I consider myself very lucky to have had the opportunity to have him as my supervisor.

I would like to thank my colleagues Anmar Hammadi, M. Sc., Nawaz Zai, M. Sc., and Flavio Junior Neves, MBA, who helped me review the experiments and validate my tool choices and the approach for metrics collection and conclusions.

I would also like to thank my wife Ana for her unambiguous support during the late hours of reading and writing, and the many one-sided babbling tech conversations during our breakfasts when she pretended to be interested in the nitty gritty details of my design experiment.

Of course, none of it would have been possible without my mother Alcione, who worked extremely hard in three shifts for many years, to be able to afford good primary and secondary education for her children. Without her herculean effort, I wouldn't be able to be here, in this very privileged position, in the final steps of concluding a master's degree.

TABLE OF CONTENTS

Contents

ABSTRACT	3
ACKNOWLEDGEMENTS	4
TABLE OF CONTENTS	5
TABLE OF FIGURES	8
TABLE OF TABLES	11
1 INTRODUCTION	12
1.1 Project Background.....	12
1.2 Project Description.....	13
1.3 Project Aims and Objectives	15
1.4 Project Evaluation.....	16
1.5 Project Scope	17
1.6 Thesis Roadmap.....	18
2 LITERATURE REVIEW	19
2.1 Introduction	19
2.2 Chaos Engineering Techniques.....	19
2.3 Self-Healing and Self-Adaptive Systems.....	22
2.4 Microservice Architecture	27
2.5 Cloud Infrastructure	29
2.6 Conclusions	33
3 DESIGN AND METHODOLOGY	34
3.1 Introduction	34
3.2 Ethical Considerations.....	35
3.3 Cloud Provider Selection.....	36
3.4 Microservice Application Development.....	37
3.5 Self-healing Architecture Design	38
3.6 API Testing Tool	41
3.7 Fault Injector.....	43
3.8 Expert Interview Design.....	44

3.9	Experiment Design	45
3.10	Conclusions	50
4	DEVELOPMENT PROCESS	51
4.1	Introduction	51
4.2	Java Micro-service Implementation	52
4.3	AWS Cloud Architecture implementation.....	52
4.3.1	VPC Creation.....	53
4.3.2	Key Pair Creation	53
4.3.3	(Windows users) Install Putty	53
4.3.4	EC2 Creation.....	53
4.3.5	Adding Java Application file into the EC2 instance.....	55
4.3.6	Connecting into EC2 through Putty and running the application.....	55
4.3.7	AMI Extraction.....	56
4.3.8	Launch template	57
4.3.9	Auto Scaling group and Elastic Load Balancer.....	57
4.3.10	Warm Pooling	58
4.4	JMeter Test Plan Implementation	59
4.5	AWS Fault Injection Simulator (FIS) Creation	59
4.6	Experiment Execution	60
4.6.1	No Warm Pooling Configuration	62
4.6.2	Stopped Warm Pooling Configuration	63
4.6.3	Running Warm Pooling Configuration	63
4.6.4	Hibernated Warm Pooling Configuration.....	64
4.7	Conclusions	70
5	RESULTS AND EVALUATION	72
5.1	Introduction	72
5.2	Calibration - Results and Evaluation	73
5.3	No Warm pooling Experiment.....	74
5.4	Stopped Warm Pooling Experiment	76
5.5	Hibernated Warm Pooling Experiment	77
5.6	Running Warm Pooling Experiment	78
5.7	Conclusions	80
5.8	Expert Interviews Conclusions	84

6	CONCLUSIONS AND FUTURE WORK	87
6.1	Introduction	87
6.2	Conclusions	88
6.3	Contributions and Impact.....	92
6.4	Future Work.....	93
	BIBLIOGRAPHY	100
	APPENDIX A: MY JOURNEY	103
	APPENDIX B: EXPERIMENT USER GUIDE	105
1.	Java Micro-service Implementation	105
2.	AWS Cloud Architecture implementation.....	107
3.	JMeter Test Plan Implementation	139
4.	AWS Fault Injection Simulator (FIS) Creation	143
5.	Experiment Execution	146
	APPENDIX C: INTERVIEW PRESENTATION.....	148

TABLE OF FIGURES

FIGURE 1.1 RESILIENCE / AWS ARCHITECTURE BLOG	12
FIGURE 1.2 FIS – AWS FAULT INJECTION SIMULATOR	14
FIGURE 2.1 MONOCLE UI FOR RPC DEPENDENCIES WITHIN A CLUSTER.....	20
FIGURE 2.2 CHAOS ENGINE ALGORITHM.....	21
FIGURE 2.3 OVERALL ARCHITECTURE OF CHESS	22
FIGURE 2.4 MUSA OVERALL PROCESS	25
FIGURE 2.5 SELF-ADAPTATION TECHNIQUES RESEARCH QUESTIONS	27
FIGURE 2.6 MICROSERVICE CAPACITY IDENTIFICATION.....	28
FIGURE 2.7 A PROTOTYPICAL IMPLEMENTATION OF THE MICO SYSTEM	31
FIGURE 3.1 HIGH-LEVEL OVERALL EXPERIMENT.....	34
FIGURE 3.2 CLOUD SERVICE PROVIDERS PER MARKET SHARE.....	37
FIGURE 3.3 VPC EXAMPLE	40
FIGURE 3.4 SECURITY GROUPS EXAMPLE	41
FIGURE 3.5 JMETER ARCHITECTURE.....	43
FIGURE 3.6 FIS EXPLAINED	44
FIGURE 3.7 ELASTIC BEANSTALK AUTO SCALING GROUP CONFIGURATION	46
FIGURE 3.8 INITIAL DESIGN	47
FIGURE 3.9 RECOVERY TIME VS. NUMBER OF FAILED MODULES	47
FIGURE 3.10 FINAL DESIGN	48
FIGURE 3.11 JMETER TEST PLAN CONFIGURED	48
FIGURE 3.12 JMETER RESULT IN TABLE VIEW	49
FIGURE 4.1 AWS CLOUD ARCHITECTURE DESIGNED.....	52
FIGURE 4.2 AWS CONSOLE – VPC.....	52
FIGURE 4.3 AWS CONSOLE – KEY PAIRS.....	53
FIGURE 4.4 AWS CONSOLE – EC2 LAUNCH.....	54
FIGURE 4.5 WINSCP CONSOLE	55
FIGURE 4.6 PUTTY CONSOLE – LOGIN	56
FIGURE 4.7 PUTTY CONSOLE – JAR FILE.....	56
FIGURE 4.8 AWS CONSOLE – CREATING IMAGE	57
FIGURE 4.9 AWS CONSOLE – LAUNCH TEMPLATES	57

FIGURE 4.10 AWS CONSOLE – AUTO SCALING GROUP.....	58
FIGURE 4.11 AWS DIAGRAM WITH ALL STEPS	59
FIGURE 4.12 JMETER LOGO	59
FIGURE 4.13 AWS FIS MENU OPTION	60
FIGURE 4.14 AWS FIS CONSOLE	60
FIGURE 4.15 JMETER VIEW RESULTS.....	61
FIGURE 4.16 AWS FIS – STARTING EXPERIMENT	61
FIGURE 4.17 JMETER – LAST SUCCESS BEFORE FAULT	62
FIGURE 4.18 JMETER – FIRST SUCCESS AFTER FAULT.....	62
FIGURE 4.19 AWS WARM POOL CONSOLE	63
FIGURE 4.20 AWS STOPPED WARM POOL WINDOW	63
FIGURE 4.21 AWS STOPPED WARM POOL INSTANCES	64
FIGURE 4.22 AWS RUNNING WARM POOL CONSOLE.....	64
FIGURE 4.23 AWS RUNNING WARM POOL INSTANCES	64
FIGURE 4.24 EBS ENCRYPTION	65
FIGURE 4.25 AWS CONSOLE – KEY CREATION	65
FIGURE 4.26 AWS CONSOLE – AMI CREATION	65
FIGURE 4.27 AMI CREATION DETAILS	66
FIGURE 4.28 AWS CONSOLE – CREATING NEW LAUNCH TEMPLATE.....	66
FIGURE 4.29 AWS CONSOLE – NEW LAUNCH TEMPLATE DETAILS	67
FIGURE 4.30 AWS CONSOLE – EC2 LISTING	67
FIGURE 4.31 AWS CONSOLE – EC2 ERROR LOGS	68
FIGURE 4.32 AWS CONSOLE – AMI CREATION WITH DEFAULT KEY	69
FIGURE 4.33 AWS CONSOLE – EC2 SUCCESSFUL LISTING	70
FIGURE 4.34 AWS HIBERNATED WARM POOL WINDOW	70
FIGURE 5.1 NO WARM POOLING RESULTS GRAPH	75

FIGURE 5.2 JMETER RUNNING WARM POOL OUTLIER	76
FIGURE 5.3 STOPPED WARM POOLING RESULTS GRAPH	77
FIGURE 5.4 STOPPED WARM POOLING JMETER FAILURES	78
FIGURE 5.5 HIBERNATED WARM POOLING RESULTS GRAPH	79
FIGURE 5.6 RUNNING WARM POOLING RESULTS GRAPH	80
FIGURE 5.7 JMETER CONTINUED INTERMITTENT FAILURES	81
FIGURE 5.8 RECOVERY TIME COMPARISON GRAPH.....	82
FIGURE 5.9 RECOVERY TIME COMPARISON CHART.....	82
FIGURE 5.10 AWS RECOVERY TIME COMPARED TO FRINCU, ET AL. (2011)	84

TABLE OF TABLES

TABLE 1.1 CONFIGURED TO FACILITATE WARM POOLING.....	13
TABLE 5.1 BROADBAND DETAILS	73
TABLE 5.2 TIME TO UNRESPONSIVENESS METRICS	74
TABLE 5.3 NO WARM POOLING DETAILS	75
TABLE 5.4 STOPPED WARM POOLING DETAILS	77
TABLE 5.5 HIBERNATED WARM POOLING DETAILS	78
TABLE 5.6 RUNNING WARM POOLING DETAILS	79
TABLE 5.7 RESULT CONVERSION TO SECONDS.....	82
TABLE 5.8 WARM POOLING PERFORMANCE COMPARISON.....	83
TABLE 5.9 WARM POOLING COST/COMPLEXITY COMPARISON.....	84

1 INTRODUCTION

“In all chaos there is a cosmos, in all disorder a secret order.”

- Carl Jung, *The Archetypes and The Collective Unconscious*

1.1 Project Background

Over the past decades, microservice architecture has become the de-facto pattern in the Software Engineering industry (P. Jamshidi, et al. 2018). Microservice architecture is widely accepted as being a fine-grained, loosely coupled collection of independent services communicating through lightweight protocols. When in combination with cloud-provided infrastructure, microservices become easier to provision and scale (either horizontally or vertically). Machines can be provisioned in minutes and billing plans are varied and flexible. Furthermore, cloud providers also offer out-of-the-box highly available and fault tolerant services, facilitating the creation of self-healing architectures, which refers to the ability of systems to detect and remediate issues without human intervention. At the time of this writing AWS is the biggest cloud provider in the market, leading by a large margin (Borge and Poonia, 2020).

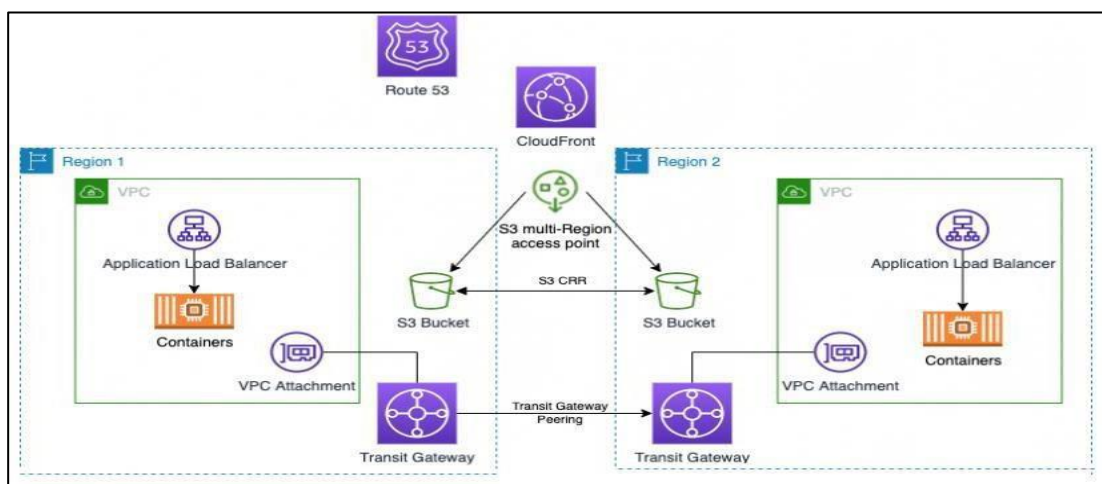


Figure 1.1: Resilience / AWS Architecture Blog (Amazon Blog Posting¹)

Chaos engineering is a discipline within software engineering and system reliability engineering that focuses on proactively testing the resilience and robustness of complex systems by intentionally introducing controlled disruptions or failures into them.

¹ <https://aws.amazon.com/blogs/architecture/creating-a-multi-region-application-with-aws-services-part-2-data-and-replication/>

The primary objective of chaos engineering is to uncover vulnerabilities, weaknesses, or unforeseen behaviors in systems before they cause significant outages or failures in real-world scenarios.

By intentionally introducing chaos or disruptions in a controlled environment, chaos engineering aims to build confidence in the system's ability to withstand unexpected failures, improve its resilience, and enhance overall reliability. It has gained prominence in modern cloud-native architectures, distributed systems, and microservices where complexity and interdependencies between components make systems more susceptible to failures.

1.2 Project Description

This research will explore the use of Chaos Engineered AWS Fault Injector Simulator (FIS) against a Self-Healing Microservice Architecture deployed in AWS. One outcome will be to explore that given reasonable Service Level Objectives (SLO), when configured appropriately, AWS can be a cost-effective, resilient, and reliable cloud provider. This research will assume an SLO of one minute for AWS to self-heal during fault injections, which will be referred to as ‘self-healing SLO’ throughout this paper. Cloud providers, such as AWS, claim to offer highly available and fault tolerant services, but it raises the question, how fault tolerant? If financial resources are not a limitation nor concern, then a company or individual could build a software architecture unnecessarily redundant in multiple availability zones and regions, and be extremely resilient, but when it comes to a cost-effective architecture, how fault tolerant can it be? To answer this question, the author proposes an experiment. Inspired by AWS best practices (Amazon Web Services, 2023), in this research, a self-healing cost-effective microservice architecture will consist of:

- A microservice application serving HTTP GET requests (backend-service app). The service will be running on two general purpose t2.micro instances, for redundancy. An Auto Scaling group with the policy described in Table 1.1 below will also be provisioned.

Desired Capacity	Maximum Capacity	Minimum Capacity
2	4	2

Table 1.1: Configured to facilitate Warm Pooling

Also, an ELB (Elastic Load Balancer) will be configured to redirect incoming requests to

healthy instances. It was decided to run two instances instead of a single instance due to AWS EC2 instances committing to 99.99% availability SLA (Service Level Agreement) per EC2 Region, therefore two instances present eight 9's availability, which corresponds to 3.15 seconds of downtime per year. To challenge the resilience of the architecture described above, faults will be injected via Chaos Engineered AWS Fault Injector Simulator (FIS). A FIS experiment template will be created which will define fault actions against EC2 instances.

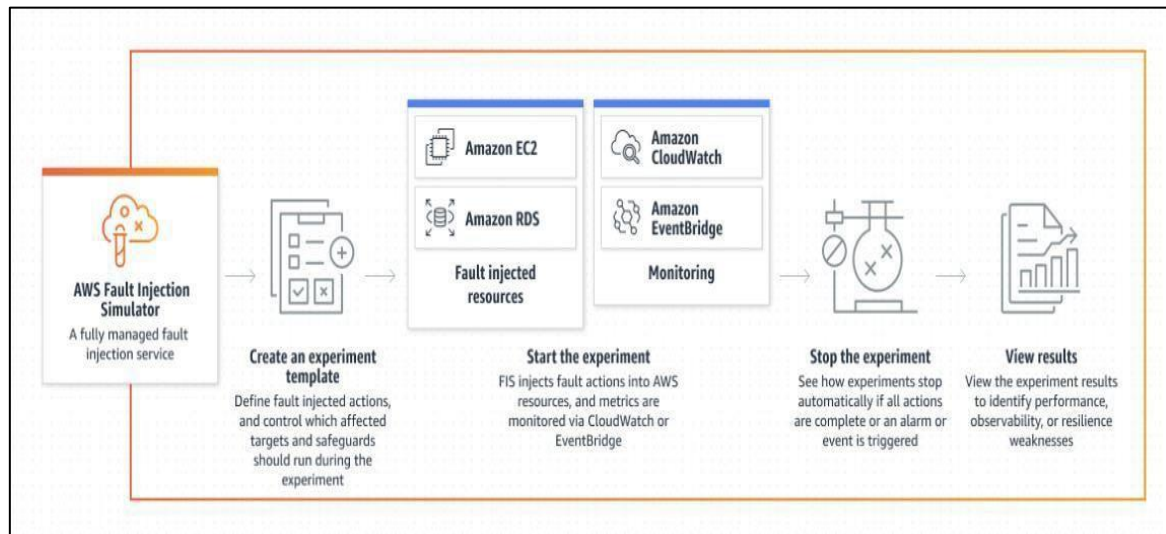


Figure 1.2: FIS – AWS Fault Injection Simulator (Amazon Fault Services²)

Metrics will be monitored via a JMeter test plan, which will test the health of the application every second and produce metrics and visual reports.

Key Research Question

Can Chaos Engineering techniques implemented via AWS Fault Injector Simulator (FIS) degrade an ‘Industry standard Self-healing Cloud-based microservice architecture’ beyond a one-minute self-healing SLO?

Key Hypothesis

AWS is widely known for its fault tolerance and high availability. However, when running a cost- effective architecture, chaos engineering techniques can still bring down services and disrupt agreed SLOs.

- **H_A:** On a microservice architecture running redundant instances under an Elastic Load Balancer and Auto Scaling group with warm pooling, AWS Fault Injector

² <https://aws.amazon.com/fis/>

Service can inject enough faults so that the public facing service cannot respond to requests within a one-minute window.

- ***H₀***: The use of ELBs and Auto Scaling groups configured with Warm Pooling over redundant EC2 instances is widely accepted as highly resilient and can cope with multiple failures while still maintaining high availability and response-times within the one-minute window.

1.3 Project Aims and Objectives

Project Objective: To explore how a cost-effective self-healing cloud microservice architecture will react to chaos engineered fault injections.

To achieve the objective, an experiment will be executed on the Amazon Web Services platform. The experiment will require the creation of a microservice (backend-service) connected to an in-memory database. This will be implemented using the Java programming language and the Spring Boot framework. An Elastic Load Balancer will be configured to route requests to the microservice, exposing an API that will be the point of external testing later in the experiment. Internally, the backend-service will run a select statement, retrieve data, and expose the data in JSON format in the response. Crucially, the computational logic of the applications is not relevant in the context of this experiment, given that the experiment is focused on self-healing capabilities, and not performance.

To run this architecture, the author will setup the following components in the AWS platform:

- On the AWS console, navigate to ‘Key Pairs’, create a key pair, download, and save the private key.
- Create a standard security group.
- Deploy the backend-service application into an EC2 instance, configure the instance to spin up the application at start-up via the ‘user-data’ script.
- Create an image of the EC2 instance, so it can be used by the Auto Scaling group.
- Create an Auto Scaling group (setup policy as per Table 1).
- Setup one virtual private cloud (VPC) with the standard accompanying configuration (subnets, route tables and network connections).

- Configure an Elastic Load Balancer
- Create an AWS Fault Injector Simulator setup to terminate both EC2 instances:
 - Setup the IAM policy for FIS so it has the privilege to terminate instances.
 - Attach the new policy to a role (by either creating a new role or re-using an existing one).
 - Assign the role to FIS.
 - Add Action ('aws:ec2:terminate-instance' passing in the EC2 instances ID parameter).

A Test Plan will be created in Apache JMeter with:

- One single Thread Group (as this is not a stress-test).
- HTTP Request Sampler pointing to the Elastic load balancer for Service A.
- Graph and Table listeners.

Once the above configuration is in place, the JMeter test plan will be started, which will continuously test the backend-service API and collect response-times. The AWS FIS setup will then be started. Once FIS has been executed and the targeted EC2 instances are terminated, an additional five minutes will be added to allow for the Auto Scale policy to spin up extra compute instances, then finally the JMeter test plan will be stopped.

Auto Scaling groups allow for different configuration variations, which have effects on recovery time of the service, so they will be explored as part of the experiment:

1. No warm pooling.
2. Warm pooling with one instance on 'Stopped' state.
3. Warm pooling with one instance on 'Running' state.
4. Warm pooling with one instance on 'Hibernated' state.

1.4 Project Evaluation

JMeter metrics will produce the main outcome of the experiment. The following metrics will be provided from the 'View Results in Table' listener:

- Sample#: Identifier of the request in incremental numbers.
- Start Time: Time in which the request has been started (up to milliseconds).
- Thread Name: Name of the thread that initiated the request.

- Label: Label of the request.
- Sample Time(ms): Time between the sending of the request and the receiving of the response back.
- Status: HTTP response status code
- Bytes: Size of the request in bytes
- Sent Bytes: Size of the response in bytes
- Latency: Latency of the connection
- Connect Time(ms): Time required for the handshake between sender and receiver.

The experiment is expected to take no longer than 10 minutes and will be executed eight times per warm pool configuration. Any outliers will be examined, noted, and re-executed. An experiment execution will be considered an outlier if the time between the last successful response and the first successful response after the fault injection has a 50% variation from the previous two executions median (except for the first and second executions, which will be evaluated against the following two executions). A final report will be produced from the compiled results, with the median value per warm pooling configuration. If within the four warm pooling configurations, the median ‘Sample Time (ms)’ goes over the one-minute SLO, then the hypothesis has been proven.

1.5 Project Scope

The focus of this project is to stress and validate the resilience of a cloud architecture, therefore performance metrics will not be collected and are out of scope of this research (i.e., API responses latency, API throttling, or how many threads the solution can handle simultaneously before degrading).

Securing assets in the cloud is also not the focus of this project. Minimum security configuration will be in place for inbound and outbound requests in the load balancer and the Linux instances but will by no means be a state-of-the-art security configuration.

The research is also not focused on the quality of the code deployed to the cloud architecture, therefore some of the best software development practices (i.e., unit tests, integration tests) were omitted.

1.6 Thesis Roadmap

Chapter 2 is the Literature Review chapter, it focuses on four main topics: Chaos Engineering Technique, Self-Healing and Self-Adaptive Systems, Microservice Architecture, and Cloud Infrastructure.

Chapter 3 is the Design Chapter which presents the design of the system and the design of the range of experiments that will be undertaken as part of this research.

Chapter 4 is the Development Chapter which presents the development of the system and the execution of the range of experiments that were undertaken as part of this research.

Chapter 5 is the Results and Evaluation Chapter which presents the results of each of the experiments as well as an analysis and evaluation of these results with respect to each other and the relevant literature.

Chapter 6 is the Conclusions and Future Work Chapter which presents the key findings of this project, highlighting what aspects of the research went well and what aspects did not go well. It also discusses some future directions that the research may take.

2 LITERATURE REVIEW

“Research is to see what everybody else has seen, and to think what nobody else has thought.”

- Arthur Schopenhauer, *1851 Parerga und Paralipomena*

2.1 Introduction

This chapter explores the four key areas of the research, to develop the required foundation of understanding to begin the design and development on the proposed experiment. Those four main areas are as follows:

- Chaos Engineering Techniques
- Self-Healing and Self-Adaptive Systems
- Microservice Architecture
- Cloud Infrastructure

2.2 Chaos Engineering Techniques

Chaos engineering involves conducting systematic experiments on a system to instill confidence in its capacity to endure challenging operational conditions (Rosenthal and Jones, 2020). In software development, it is common to specify the need for a software system to gracefully handle failures while maintaining an acceptable level of service quality. This characteristic, often referred to as *resilience*, is frequently outlined as a critical requirement. Unfortunately, many development teams struggle to fulfill this requirement, often due to constraints like tight deadlines or limited domain expertise. Chaos engineering presents itself as a valuable technique to meet the resilience mandate. It serves as a method for enhancing resilience against a range of potential setbacks, including infrastructure failures, network disruptions, and application glitches (Jernberg, *et al.*, 2020).

Basiri, *et al.* (2016) describe the concepts and benefits of Chaos Engineering (exemplified by its use at Netflix). They also describe Chaos Engineering techniques to run experiments to validate the resilience of a software architecture. Their focus was on bringing awareness, so practitioners and research communities come to recognize Chaos Engineering as its own discipline and continue developing it.

Three years later, in 2019, Basiri, *et al.* published a subsequent paper describing the evolution of the Netflix platform for automatically generating and executing chaos experiments in their production environment. Netflix has built an in-house orchestration system named ChaP (Chaos Automation Platform) which interacts with multiple internal Netflix services to carry out chaos engineered experiments via a fault injection system, also developed in-house, named FIT. In designing ChaP, an additional service was also developed, named Monocle, which has two functions, introspecting services and generating experiments:

Monocle: fake (fake-prod)

Config

Generated Test Cases

Generated Runs

Select New Monocle

Last Updated: an hour ago

For guidance on tuning dependencies, see Hystric and NINWS Tuning

NINWS

Hystric

Service Name

Safe To Fail?

fake_service_1

✖

Client Name	Known Impacts	App Name	Read Timeout Sequence	Max Auto Retries	Max Auto Retries Next Server	% of Requests	Max RPS	Hystric Commands															
▼ fakeservice1-client	SPS	FAKESERVICE1	3000	0	1	0.11	22	<table> <tr> <th>Name</th> <th>Safe To Fail</th> <th>Fallback?</th> <th>Known Impacts</th> <th>Timeout</th> </tr> <tr> <td>FakeService1CommandA</td> <td>✓</td> <td>✓</td> <td></td> <td>3000</td> </tr> <tr> <td>FakeService1CommandB</td> <td>✓</td> <td>✓</td> <td></td> <td>1000</td> </tr> </table>	Name	Safe To Fail	Fallback?	Known Impacts	Timeout	FakeService1CommandA	✓	✓		3000	FakeService1CommandB	✓	✓		1000
Name	Safe To Fail	Fallback?	Known Impacts	Timeout																			
FakeService1CommandA	✓	✓		3000																			
FakeService1CommandB	✓	✓		1000																			

fake_service_2

✖

Client Name	Known Impacts	App Name	Read Timeout Sequence	Max Auto Retries	Max Auto Retries Next Server	% of Requests	Max RPS	Hystric Commands										
▼ fakeservice2-client	Signup	FAKESERVICE2	3000	0	1	19.63	51	<table> <tr> <th>Name</th> <th>Safe To Fail</th> <th>Fallback?</th> <th>Known Impacts</th> <th>Timeout</th> </tr> <tr> <td>FakeService2CommandA</td> <td>✖</td> <td>✖</td> <td></td> <td>2000</td> </tr> </table>	Name	Safe To Fail	Fallback?	Known Impacts	Timeout	FakeService2CommandA	✖	✖		2000
Name	Safe To Fail	Fallback?	Known Impacts	Timeout														
FakeService2CommandA	✖	✖		2000														

Figure 2.1. Monocle UI for RPC dependencies within a cluster (Basiri, *et al.*, 2019)

Their paper demonstrates that it is feasible to generate and run chaos experiments in any environment, including production, automatically and safely.

Migirditch, *et al.* (2022) propose a Chaos Engine that stresses agents by intelligently searching over ‘scenario chaos factors’ reflecting real-world events. Their approach is focused on facilitating resilient strategic military planning. Their chaos engineering methodology prioritizes expediting agent training to establish robust policies. This is achieved through the introduction of the 'Adversarial Architect', a system that explores parametric chaos elements (such as enemy force composition, platform failures, weather, communication interference) to identify situations that result in preventable failure scenarios.

Algorithm 1: Chaos Engine

Input : Learning Agent π_L , Adversary π_A , Environment E , Performance Function P .
Output : Chaos Factors θ , Item Response Model M
Step 1 : $\theta \leftarrow$ Initial Chaos Factors
Step 2 : $M \leftarrow$ Randomly initialized item response model.
Step 3 : **while** M is not converged **do**
Step 4 : $Fitness_{\theta_i} \leftarrow P(\pi_A, E_{\theta_i}) - P(\pi_L, E_{\theta_i})$
Step 5 : $Fitness_{\theta_i} \leftarrow Fitness\ Sharing(\theta)$
Step 6 : $\theta \leftarrow Selection(Fitness_{\theta})$
Step 7 : $\theta \leftarrow Mutation(\theta)$
Step 8 : $M \leftarrow Update(\theta, P(\pi_L, E_{\theta}))$
Step 9 : **return** θ, M

Figure 2.2. Chaos Engine Algorithm (Migirditch, et al., 2022)

Zhang, et al. (2021) introduce an innovative framework named PHOEBE, designed for injecting faults related to system call invocations. PHOEBE offers several distinctive features.

- It grants developers complete visibility into system call invocations.
- It creates error models that closely resemble real-world errors occurring in production environments.
- PHOEBE is capable of autonomously conducting experiments to systematically evaluate the reliability of applications in the context of system call invocation errors during production.

To assess the efficiency and runtime impact of PHOEBE, they conducted evaluations on two actual applications within a production setting, both utilizing the Java software stack. The results demonstrate that PHOEBE effectively generates realistic error models and identifies critical reliability issues associated with system call invocation errors. To the best of their knowledge, the concept of "realistic error injection," which involves basing fault injection on actual production errors, has not been explored previously.

This section presented an introduction to Chaos Engineering, an approach to software development that helps support the graceful degradation of software services in the event of failures occurring. It also outlines some state-of-the-art research in the field of Chaos Engineering.

2.3 Self-Healing and Self-Adaptive Systems

Self-healing systems perform periodic health assessments on various components and autonomously initiate corrective actions, such as redeployment, to restore them to their intended operational conditions (Ghosh, *et al.*, 2007). At the hardware level, this self-healing process involves relocating services from a malfunctioning node to a functioning one while also conducting health evaluations on various components. On the other hand, self-adaptive architecture can modify itself, e.g., adjust its states or behaviors, to satisfy certain objectives (Yang *et al.*, 2013)

Ali Naqvi, *et al.* (2022) propose CHESS, an approach for systematic evaluation of self-adaptive and self-healing systems that builds on Chaos Engineering techniques. An exploratory study was conducted to evaluate a self-healing application to evaluate the limitations and promises of CHESS. The managed system is injected with faults using chaos engineering concepts. The problem detection, fault diagnosis, fault recovery, and knowledge modules of the self-healing system are reflected in a feedback loop that adheres to the MAPE-K reference model. To capture the status of the system being evaluated before, during, and after fault injection for subsequent analysis, the system self-monitoring component offers intensive monitoring and data collecting.

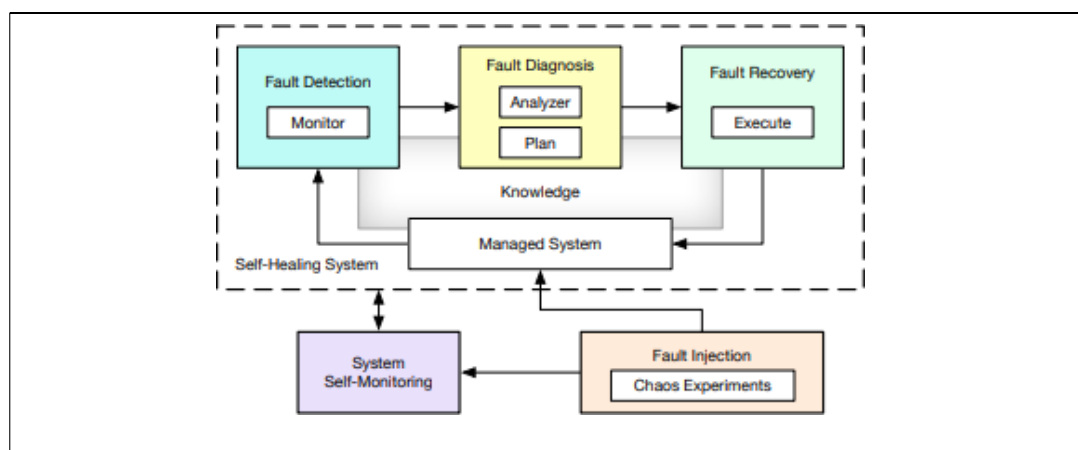


Figure 2.3. Overall architecture of CHESS (Ali Naqvi, *et al.*, 2022)

Motivated by the growth value of private cloud solutions, Petrenko (2021) conducted a thorough comparison between different approaches and technologies that allow for building a resilient cyber-stable private cloud based on well-known and proprietary artificial immune system (AIS) models and approaches, as well as technologies for distributed data processing, container orchestration, logging, security, and others.

Petrenko focuses on five layers in his paper:

- Client application,
- Data Services (Cassandra Postgresql, Ignite, Kafka, Elasticsearch),
- Core Services (Ubuntu Linux, Kubernetes, Ceph),
- Hardware (Compute node, Storage Node, Network),
- Management (Console, Monitoring, Logging).

Dashofy, *et al.* (2002) present an approach and vision for developing self-healing systems, focusing primarily on an event-based developed infrastructure to support the creation and execution of repair strategies.

They have built a substantial infrastructure to support their vision based on:

- xADL 2.0, an extensible architecture description language, describes software architectures and their alterations.
- c2.fw, a flexible framework for constructing event-based systems, is used to instantiate and manage software systems.
- ArchStudio 3 development environment, which is also built on c2.fw, maintains and manages the mapping between architectural descriptions and running systems, as well as hosting design critics, which may be used to examine architecture descriptions or the impact of a change before it is implemented.

Their long-term strategy can be seen as a refinement of the preceding approach, focusing on tools and approaches that enable more flexibility or dependability in system reconfiguration in general than was previously used.

In their paper, Frincu, *et al.* (2011) presented a proposal for a multi agent task scheduling system enhanced with self-healing capabilities. To deliver a distributed, self-healing scheduling platform, they addressed the following issues:

1. Provide fully distributed storage and communication mechanisms by using distributed underlying platforms.
2. Since agents must be fault tolerant and self-adaptive, they implement agents as modular smart control loops.
3. Maintain independence among multiple providers and easy switch scheduling policies by using an inference engine for policy execution.
4. Facilitate flexibility in changing negotiation policy according to specific needs by adding a “negotiator” as an easy-to-integrate plug-in module.

Simulated tests were conducted to minimize, but not reduce, the number of agents involved in cross-service scheduling, and finally, their RMS was tested on a real-world environment to evaluate its healing capabilities. Their tests have demonstrated that the platform's recovery times are within acceptable limits. Their next step is to integrate and test the platform using the future, cloud-based API provided by mOSAIC.

Pedro Dias, *et al.* (2020) present and discuss a set of patterns for self-healing IoT systems that bring improvements in reliability, by providing error detection, recovery, and health-check mechanisms. In their paper they present a collection of 27 patterns, as well as a pattern language, that can enhance the robustness of IoT systems by enabling them to heal themselves. These patterns are grouped into two main categories: *Error Detection* (Probes) and *Recovery & Maintenance of Health*. These patterns are mostly derived from previous work on related fields such as: *Cloud computing*, *Space systems engineering* and *Critical and industrial systems*. These patterns are not new, but their contextualization to IoT systems is introducing new concepts, both in terms of fault-tolerance and self-healing.

Seeger, *et al.* (2019) tackle the challenge of optimally self-healing IoT edge systems with a combination of two key concepts:

- a policy-enabled failure detector that enables adaptable failure detection, and
- an allocation component for the efficient selection of failure mitigation actions.

In their paper, they introduce a system aimed at facilitating the autonomous recovery of IoT choreographies. This system is primarily comprised of two key components:

1. A pioneering failure detection concept that offers extensive flexibility in configuring parameters tailored to specific applications and policies. They also provide recommendations on parameter selection.
2. They have devised an Integer Linear Programming (ILP) formulation to achieve optimal task allocation, taking into consideration energy efficiency. Furthermore, they have developed a heuristic approach that allows for real-time allocation computation.

They have conducted evaluations for both the PE-FD failure detector and the performance of the allocation algorithm.

Rios, *et al.* (2017) produced a paper that introduces a novel modeling language and an accompanying tool designed to cater to the specific requirements of multi-cloud

application modeling.

This solution addresses the limitations of current modeling approaches by simplifying two critical aspects:

1. It streamlines the computation of Composite Security Service Level Agreements (SLAs) that encompass security and privacy considerations.
2. It enhances risk analysis and service matchmaking by considering not only the functional and business aspects of cloud services but also their security aspects.

The language and tool discussed in their paper were developed within the scope of the MUSA EU-funded project. These enhancements build upon the existing CAMEL language, which already provided comprehensive meta-models for Requirements, Deployment, Scalability, and Security, covering many requirements for specifying multi-cloud applications. However, the MUSA project identified additional needs for more detailed deployment and security specifications, risk analysis, and the composition of Security SLAs. Consequently, extensions to the CAMEL language were developed to fulfill these requirements. The modeling tool that supports this extended CAMEL meta-model, known as the MUSA Modeller, has been seamlessly integrated with the MUSA framework and is accessible on the MUSA website at www.musa-project.eu.

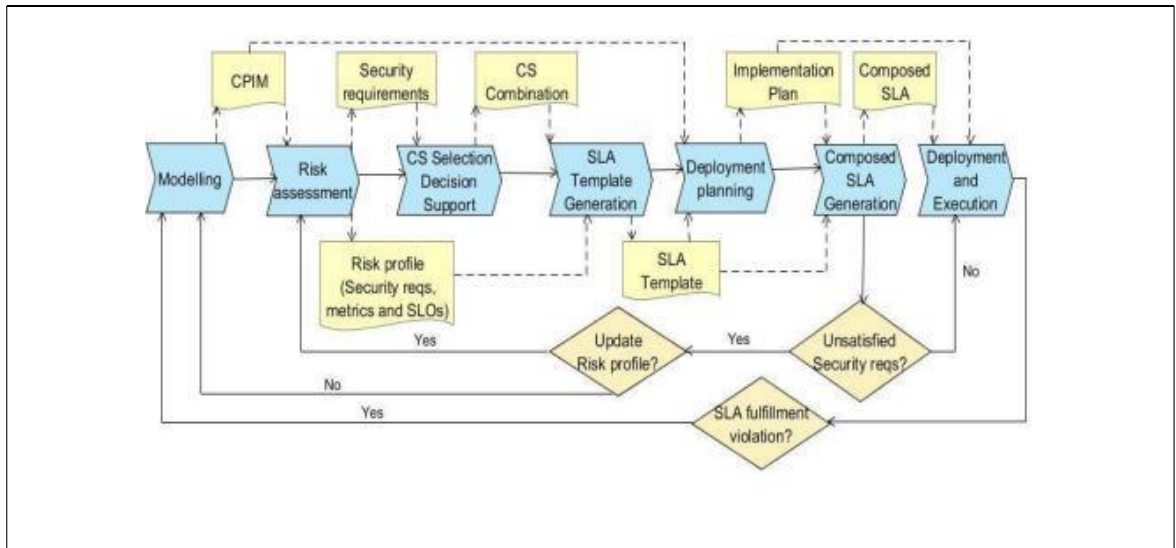


Figure 2.4. MUSA Overall Process (Rios, et al., 2017)

Mendonca, et al. (2018) investigate several representative self-adaptation solutions that have been proposed recently from the perspective of generality and reusability, and propose directions for the following challenges:

1. New Adaptation Mechanisms
2. New Control Loop Deployment Structures
3. New Continuous Delivery Strategies
4. New Testing Approaches
5. New Migration Strategies to Microservices

Colombo, *et al.* (2022) address the challenges of accurately and efficiently monitor Fog environments. They introduced ‘AdaptiveMon’, an adaptive P2P monitoring solution that leverages a knowledge base constantly updated with monitoring information to dynamically modify the system behavior by triggering countermeasures. The experimental results demonstrate that adaptive behaviors improve monitoring accuracy while optimizing the utilization of available resources compared to a non-adaptive solution.

Mendonca, *et al.* (2019) identify key challenges for the development of microservice applications development, delivery, and operations from multiple self-adaptation perspectives. They present the following contributions in their work:

1. We provide a detailed description of a cloud-based intelligent video surveillance application, serving as an illustrative example of a self-adaptive microservice system.
2. In the context of this example application, we highlight and explain various challenges that arise during microservice development, delivery, and operations, considering multiple aspects of self-adaptation.
3. We explore potential avenues for addressing the primary challenges encountered in the development of self-adaptive microservice systems. This exploration involves drawing upon existing microservice practices and technologies to propose new directions for improvement.

A 2021 paper, written by Filho, *et al.* conducted a systematic mapping in which multiple studies on “*self-adaptation techniques and mechanisms in microservice-based systems*” were analyzed considering quantitative and qualitative research questions.

ID	Research Question
RQ1	How has self-adaptation been applied in the context of microservices?
RQ2	What types of research and contribution have been presented?
RQ3	Which phase of a self-adaptation control loop is the focus of the studies?
RQ4	What self-* properties have been addressed?
RQ5	What self-adaptation strategies have been used?
RQ6	What quality requirements have been addressed by the approaches?
RQ7	In which microservice architecture layer were the adaptations applied?
RQ8	What self-adaptation control logic has been addressed?
RQ9	What technologies have been addressed?
RQ10	Has an empirical evaluation been conducted?
RQ10.1	What strategy was used to validate the research?

Figure 2.5. Self-Adaptation Techniques Research Questions (Filho, et al, 2021)

The findings indicate that the majority of research efforts center around the "Monitor" phase, accounting for 28.57% of the adaptation control loop. Additionally, a significant emphasis is placed on achieving the self-healing property (23.81%), employing a reactive adaptation strategy (80.95%), primarily at the system infrastructure level (47.62%), and adopting a centralized approach (38.10%).

This section presented an introduction to Self-Healing systems, which are systems that perform regular checks on the key components, and to autonomously initiate corrective actions. It also outlines some state-of-the-art research in the field of Self-Healing systems.

2.4 Microservice Architecture

Microservices, which is also referred to as the microservice architecture, is an architectural approach that organizes an application into a set of services characterized by the following attributes:

1. Capable of independent deployment.
2. Loosely interconnected.
3. Aligned with specific business functionalities.
4. Managed by small, dedicated teams.

This architectural style empowers an organization to deliver large, intricate applications efficiently and consistently with speed and reliability.

Jindal, *et al.* (2019) address the challenge of identifying a Microservice Capacity (MSC) for a single microservice within a multiple microservice ecosystem. Such challenge was overcome by sandboxing a microservice and creating a performance model via the ‘Terminus’ tool. The tool assesses a microservice's capability under various deployment setups by executing a concise set of load tests and then applying a suitable regression model to the gathered performance data. The assessment of these microservice performance models across four different applications has yielded highly accurate predictions, with a mean absolute percentage error (MAPE) consistently below 10%. These outcomes from the suggested performance modeling of individual microservices serve as a crucial input for the broader microservice application performance modeling.

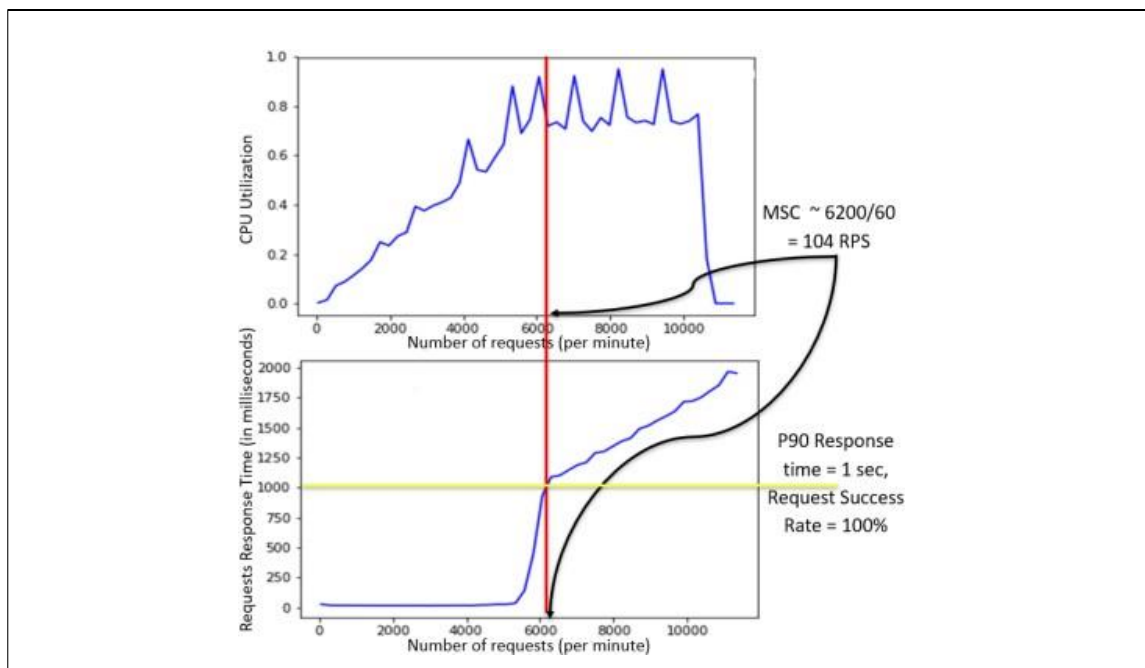


Figure 2.6. Microservice Capacity Identification (Jindal, *et al.*, 2019)

Zhang, *et al.* (2019) investigated the gap between Academia’s ideal vision and real industry’s practices on microservices. A series of industrial interviews was undertaken, encompassing thirteen diverse types of companies. Following this, they systematically structured and coded the acquired data according to prescribed qualitative methods. From the interviews, they validated both the advantages of implementing microservices, which can be gained through practical experience, and the potential challenges that may require additional investment based on their own experiences. Additionally, some of these identified challenges, such as organizational transformation, decomposition, distributed monitoring, and bug localization, could serve as valuable inspiration for researchers to pursue further investigations.

Stubbs, *et al.* (2015) reviewed container technology and introduced Serfnode, a non-intrusive Docker image, as a solution to service discovery challenges. Serfnode also has a self-healing and monitoring mechanism based on Supervisor for resiliency.

Microservices architecture has emerged as a popular approach for organizations looking to modernize their legacy applications. However, there is a significant gap in understanding the key principles required for successfully implementing a microservices architecture. Velepucha, *et al.* (2023) wrote a paper which aims to fill this void by conducting a comprehensive survey of existing literature that delves into the foundational principles of the object-oriented approach and how these concepts relate to both monolithic and microservices architectures.

Furthermore, their investigation covers not only monolithic architectures but also microservices, including an exploration of the design patterns and principles commonly applied in microservices development. Their contribution includes the compilation of a list of patterns commonly used in microservices architecture. They also compare the principles advocated by experts such as Martin Fowler and Sam Neuman in the decomposition of microservices architectures with the pioneering Principle of Information Hiding put forth by David Parnas. Parnas discusses modularization to enhance system flexibility and comprehension.

Additionally, they provide a concise summary of the advantages and disadvantages of both monolithic and microservices architectures, as gleaned from the literature review. The summary in their paper can serve as a valuable reference for researchers in academia and industry, shedding light on the current trends in microservices architecture.

This section presented an introduction to Microservice architectures, an approach to software architecture that organizes an application into a set of services. It also outlines some state-of-the-art research in the field of Microservice architectures.

2.5 Cloud Infrastructure

Cloud infrastructure refers to the essential elements required for cloud computing, encompassing hardware, abstracted resources, storage, and networking components. Consider cloud infrastructure as the foundational building blocks necessary for constructing a cloud environment. To accommodate services and applications within the

cloud, the presence of cloud infrastructure is imperative (Qian, *et al*, 2009).

Kotas, *et al.* (2018) evaluates compute-oriented instances from Amazon Web Services and Microsoft Azure cloud platforms in multiple high-performance computing benchmarks (HPCC and HPCG). Their experiment investigates the performance of various HPC benchmarks on both the AWS and Azure cloud platforms, with a specific focus on the compute-centric c4.8xlarge and H16r instance types. Nevertheless, determining which cloud platform offers the most cost-effective solution for a particular use case hinges on the computational and communication patterns of the application. In the context of the tests conducted at the time of this study, the AWS c4.8xlarge demonstrated a cost advantage in terms of raw computing power, whereas Azure's H16r excelled in providing cost-effective bandwidth. Consequently, applications that heavily rely on communication may find Azure's H16r with its faster network and larger RAM to be a more economical choice, resulting in an overall cost-saving solution. It is worth noting that cloud service providers consistently enhance their offerings. Therefore, the most reliable method for determining an application's performance in the current cloud environment is to conduct testing on the prospective system.

Yussupov, *et al.* (2020) propose a model-driven and pattern-based approach (MICO) for composing microservices, which helps with the transition from architectural models to running deployments. Central to their approach is the MICO meta-model, which harmonizes architectural and deployment considerations, simplifying the transition from integration models to active deployments. In addition to modeling interface-based service integration, the MICO Model empowers the utilization of integration services for modeling messaging-based service interactions. These integration services rely on the implementations of well-known enterprise integration patterns. They promote loosely coupled integration of microservices while abstracting away the technical intricacies linked to the underlying infrastructure deployment prerequisites. To substantiate the validity of their approach, they have conducted a prototypical implementation of the system architecture. This implementation utilized Kubernetes for container orchestration, Apache Kafka as a message-oriented middleware, and Open FaaS for managing the service integration logic. Subsequently, they executed a concrete case study based on a third-party application.

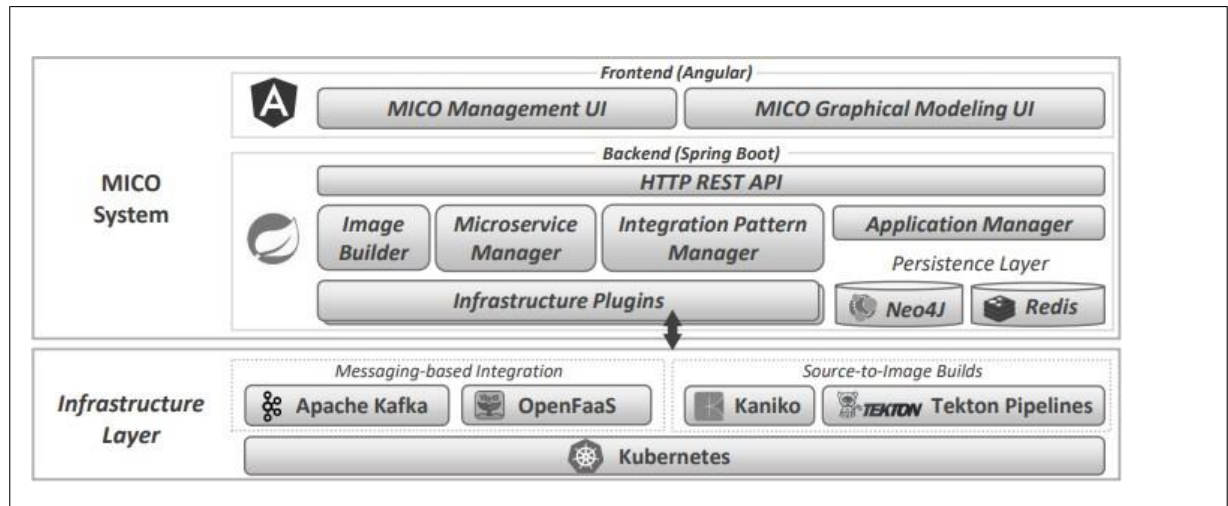


Figure 2.7. A Prototypical Implementation of the MICO system (Yussupov, et al., 2020)

Schleier-Smith, et al. (2021) discuss the evolution of cloud computing and the benefits of Serverless Architecture in terms of lowering costs and simplifying system administration. They conclude their paper with five predictions for serverless computing:

1. The present categories of FaaS and BaaS are anticipated to evolve into a broader spectrum of abstractions, which we classify into two categories: general-purpose serverless computing and application-specific serverless computing. While serverful cloud computing will not vanish, its relative usage within the cloud ecosystem is expected to diminish as serverless computing continues to address its current limitations.
2. Anticipated developments in general-purpose serverless abstractions aim to encompass nearly any conceivable use case. These abstractions will encompass state management and offer optimization possibilities—either user-driven or automatically inferred—resulting in efficiencies that rival, or potentially surpass, those of serverful computing.
3. There is no inherent reason for serverless computing to be more costly than serverful computing. We predict that as serverless technology advances and gains popularity, nearly all applications, whether small or large-scale, will cost no more—and perhaps even less—when utilizing serverless computing.
4. Machine learning is poised to assume a pivotal role in serverless implementations, enabling cloud providers to enhance the execution of extensive

distributed systems while furnishing a user-friendly programming interface.

5. The hardware landscape for serverless computing is expected to exhibit a significantly greater degree of heterogeneity compared to the prevailing x86 servers that currently underpin it.

Blinowski, *et al.* (2022) evaluate the performance and scalability of monolithic versus microservice architecture by running controlled experiments in three different deployment environments (local, Azure App Service and Azure Spring Cloud) using two different implementation technologies (Java versus C# .NET). Their findings are as follows:

1. In terms of performance on a single machine, a monolithic system outperforms its microservices-based counterpart.
2. When dealing with computation-intensive services, the Java platform demonstrates superior utilization of robust hardware, whereas this platform effect is reversed when non-computationally intensive services run on hardware with limited computational capacity.
3. In the Azure cloud environment, vertical scaling proves to be a more economically efficient choice than horizontal scaling.
4. Extending scaling beyond a specific number of instances leads to a decline in application performance.
5. The choice of implementation technology does not significantly affect the scalability performance.

Sen and Skrobot (2021) demonstrate and discuss the process of deployment and provisioning of microservices utilizing DevOps principles and practices in industry standard, more specifically AWS Elastic Container Service (ECS). They concluded that throughout the transition from testing to production phases, AWS ECS serves various environments, offering a substantial reduction in the time and labor required for deploying microservices. This eliminates the need for manual server deployment and configuration. Additionally, AWS ensures a high level of security and reliability without necessitating additional efforts. AWS also seamlessly integrates ECS with its other

services, including Elastic Load Balancer and Identity Access Management, simplifying the deployment of intricate multi-component applications in the AWS Cloud.

This section presented an introduction to Cloud infrastructure, looking at all the parts of a cloud environment, including software, hardware, and networking components. It also outlines some state-of-the-art research in the field of Cloud infrastructure.

2.6 *Conclusions*

This chapter began by dividing the research area of this dissertation into four separate topics, and then discussed multiple papers to provide the necessary background to proceed into the preparation of the proposed experiment.

As mentioned in this chapter, although much has already been extensively studied, the author has not encountered any recent/relevant paper exploring chaos engineering techniques on a self-healing Microservice Cloud Architecture.

3 DESIGN AND METHODOLOGY

“Cloud is about how you do computing, not where you do computing.”

– Paul Maritz, CEO of VMware

3.1 Introduction

In this chapter, the framework and procedures used to collect and analyze data for study will be outlined. An experiment based on self-healing cloud architecture will be designed to host a microservice application. Then, chaos engineered faults will be injected into such architecture to measure how resilient it is.

First, an infrastructure cloud provider will be selected based on its relevance in the current industry, then a microservice application will be developed. Following that, an architecture within the selected cloud provider will be defined and created to host the microservice application. To collect health-check data from the architecture independently, an API testing tool will be chosen. A chaos engineered fault injection tool will be selected to challenge the architecture. Finally, the experiment will be conducted.

Replicability of the experiment will also be facilitated by a detailed experiment user guide in Appendix B.

Research Question: *Can Chaos Engineering techniques implemented via AWS Fault Injector Simulator (FIS) degrade an ‘Industry standard Self-healing Cloud-based microservice architecture’ beyond a sixty-second self-healing SLO?*

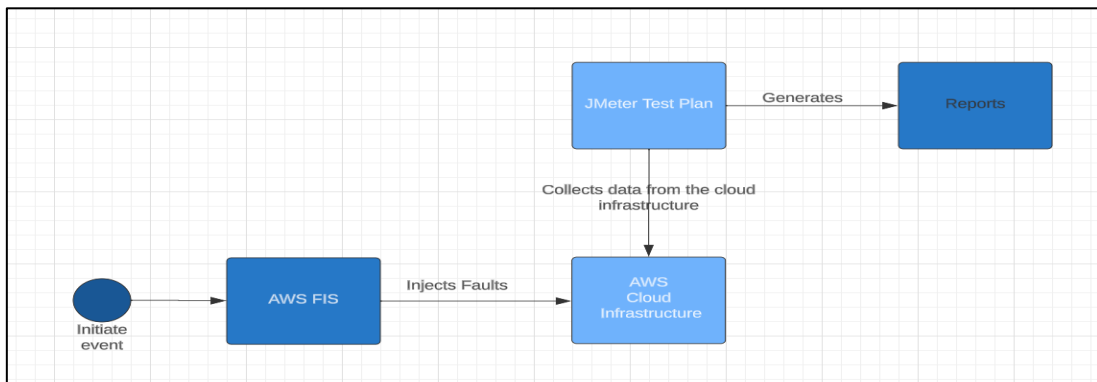


Figure 3.1. High-level Overall experiment (Author)

3.2 *Ethical Considerations*

This study is not motivated to, and will not, promote nor demote any specific technology or cloud provider. The focus is on, given a specific set of parameters, quantitatively assess how resilient a cloud architecture can be. For this research, the AWS platform will be used for the reasons outlined in Section 3.3. Nonetheless, that rationale does not encourage, nor discourage, the choice of using other cloud providers in industry or in the academia.

Regarding data controls and residency, AWS claims that the customer can manage their data effectively through the utilization of robust AWS services and tools. These tools empower the user to specify the data's storage location, implement security measures, and regulate access permissions. For example, AWS Identity and Access Management (IAM) ensures secure control over access to AWS services and resources. Additionally, services like AWS CloudTrail and Amazon Macie aid in compliance, detection, and auditing, while AWS CloudHSM and AWS Key Management Service (KMS) enable the secure creation and management of encryption keys. To further enhance data governance and residency, AWS Control Tower provides the necessary governance and control mechanisms.

Regarding data privacy, AWS claims to consistently enhance privacy protection measures by offering services and features that empower users to establish their own privacy controls, encompassing advanced access, encryption, and logging functionalities. They simplify the process of encrypting data during transit and at rest, allowing users to choose between keys managed by AWS or those they manage entirely on their own. The user can also integrate externally generated and managed keys. Their privacy management procedures are uniformly structured and scalable, governing data collection, utilization, access, storage, and deletion. To assist users in safeguarding their data, AWS provides an extensive array of best practice resources, training, and guidance, including the Security Pillar of the AWS Well-Architected Framework.

AWS claims to only handle customer data, which refers to any personal data the user uploads to their AWS account, following their documented instructions. AWS claims to not access, employ, or disclose user data without their explicit consent, except when necessary to prevent fraud and abuse or to comply with legal requirements, as outlined in the AWS Customer Agreement and AWS GDPR Data Processing Addendum. Many

customers subject to GDPR, PCI, and HIPAA regulations rely on AWS services for such workloads. AWS has attained multiple globally recognized certifications and accreditations, showcasing compliance with rigorous international standards, including ISO 27017 for cloud security, ISO 27701 for privacy information management, and ISO 27018 for cloud privacy.

Regarding AWS billing, AWS has created a User Guide console to be transparent with any charges incurred. The AWS Billing console offers functionalities for settling user AWS invoices and tracking AWS expenses and usage. If the user is part of AWS Organizations, they can also employ the AWS Billing console to oversee their consolidated billing. When registering for an AWS account, Amazon Web Services will automatically bill the credit card supplied. The user has the flexibility to view or modify their credit card details at their convenience, including the option to assign a different credit card for AWS charges. These adjustments can be made through the Payment Methods page within the Billing console.

The author would also like to highlight that no information present in this research regarding chaos engineering techniques can be readily used to deliberately harm/degrade cloud infrastructures.

3.3 Cloud Provider Selection

When deciding regarding a public cloud provider, there are several factors to consider:

- *Required Services*: Ensure that the provider offers the essential services that are needed, including computing, storage, networking, and databases.
- *Feature Set and Capabilities*: Evaluate the provider's features and functionalities that align with the specific requirements, encompassing scalability, security, and performance.
- *Cost Structure*: Verify that the pricing is competitive and that there is a clear understanding of the associated terms and conditions.
- *Support Services*: Confirm that the provider delivers effective support services, enabling users to access assistance whenever necessary.

Presently, AWS stands as the foremost cloud provider, with a global user base in the millions and commanding a market share exceeding 30 percent. Introduced by Amazon

in 2006, it has since evolved into one of the most widely adopted cloud service providers, supported by a wide community of engaged users and developers. Covering 245 countries and territories, AWS operates within 102 availability zones spanning across 32 geographic regions (as of December 2023). Its comprehensive offering encompasses more than 200 fully featured services encompassing compute, storage, networking, databases, analytics, and machine learning. Moreover, AWS regularly introduces new features to meet evolving demands.

AWS includes robust security measures, with over 140 security standards and certifications that cater to the compliance requirements of customers around the world. Microsoft Azure, Google Cloud Platform, Alibaba Cloud, IBM Cloud, Oracle Cloud Infrastructure, Tencent Cloud, DigitalOcean, UpCloud, Akamai, amongst many other cloud providers have grown over the years, but in this research the experiment will be undertaken based on AWS.

#	Cloud Service Provider	Market Share
1	Amazon Web Services (AWS)	32%
2	Microsoft Azure	22%
3	Google Cloud Platform (GCP)	11%
4	Alibaba Cloud	4%
5	Oracle Cloud	3%
6	IBM Cloud (Kyndryl)	2.5%
7	Tencent Cloud	2%
8	OVHcloud	<1%
9	DigitalOcean	<1%
10	Linode (Akamai)	<1%

Figure 3.2. Cloud service providers per market share (DGTL Ingra website³)

3.4 Microservice Application Development

In this experiment, the Java programming language will be used to develop the microservice application that will be deployed in AWS. The Java programming language is characterized by its high-level nature, object-oriented approach, and a deliberate

³ [Top 10 Cloud Service Providers Globally in 2023 - Dgtl Ingra](#)

emphasis on minimizing implementation dependencies. It is a versatile language designed to enable the "write once, run anywhere" (WORA) principle, signifying that compiled Java code can function on any platform that supports Java without requiring recompilation. Typically, Java applications are compiled into bytecode, which is executable on any Java virtual machine (JVM), irrespective of the underlying computer architecture.

While Java's syntax bears some resemblance to C and C++, it offers fewer low-level capabilities compared to both. The Java runtime environment provides dynamic functionalities, such as reflection and runtime code modification, which are typically absent in traditional compiled languages.

In this paper, the Java application will utilize the Spring framework to spin-up an in-memory H2 database containing one table with two columns, inject ten random entries into the table and then expose the entries in JSON format through a REST API via the Spring Web module.

Based on the AWS microservice definition⁴, the Java application proposed in this paper fits the description of being a microservice, as it has a well-defined interface using a lightweight API, it is autonomous, specialized, it can independently run, be updated, deployed and scaled.

3.5 Self-healing Architecture Design

In this research, the design of the cloud architecture is based on the AWS Well-Architected framework, which is widely accepted in the industry. However, given the nature of the experiment, a bigger focus was given to the Reliability Pillar, which states: *“The reliability pillar focuses on workloads performing their intended functions and how to recover quickly from failure to meet demands. Key topics include distributed system design, recovery planning, and adapting to changing requirements.”*⁵

It is also worth mentioning that most of the architectural design choices were focused on the free-tier services, which are offered by AWS to accounts created within the past 12

⁴ <https://aws.amazon.com/microservices/>

⁵ <https://aws.amazon.com/architecture/well-architected>

months. There are three main components to this self-healing architecture:

1. *EC2 instances*: These are the computational power, machines where the Java application will be deployed and executed. In this experiment two instances will be created, due to AWS EC2 instances committing to 99.99% availability SLA (Service Level Agreement) per EC2 Region, therefore two instances present eight 9's. availability, which corresponds to 3.15 seconds of downtime per year.
2. *Auto Scaling Groups*: An Auto Scaling group comprises a set of EC2 instances that are organized as a cohesive entity, serving the purpose of automated scaling and administration. Additionally, an Auto Scaling group facilitates the utilization of Amazon EC2 Auto Scaling capabilities, including health check substitutions and scaling policies. The central functions of the Amazon EC2 Auto Scaling service encompass both the management of instance quantities within an Auto Scaling group and the automatic scaling process. For this experiment, the Auto Scaling group will be set to contain a minimum of two instances and a maximum of four.
3. *Elastic Load Balancers*: Elastic Load Balancing (ELB) autonomously disperses incoming application traffic among numerous targets and virtual appliances situated in one or multiple Availability Zones (AZs).

However, these components are dependent on the lower-level services listed below:

- *VPC (Virtual Private Cloud)*: Amazon Virtual Private Cloud (VPC) empowers the user to deploy AWS resources within a logically segregated virtual network that have been personally configured. This virtual network mirrors the structure of a conventional network the user might manage in their private data center while leveraging the advantages of AWS's scalable infrastructure.

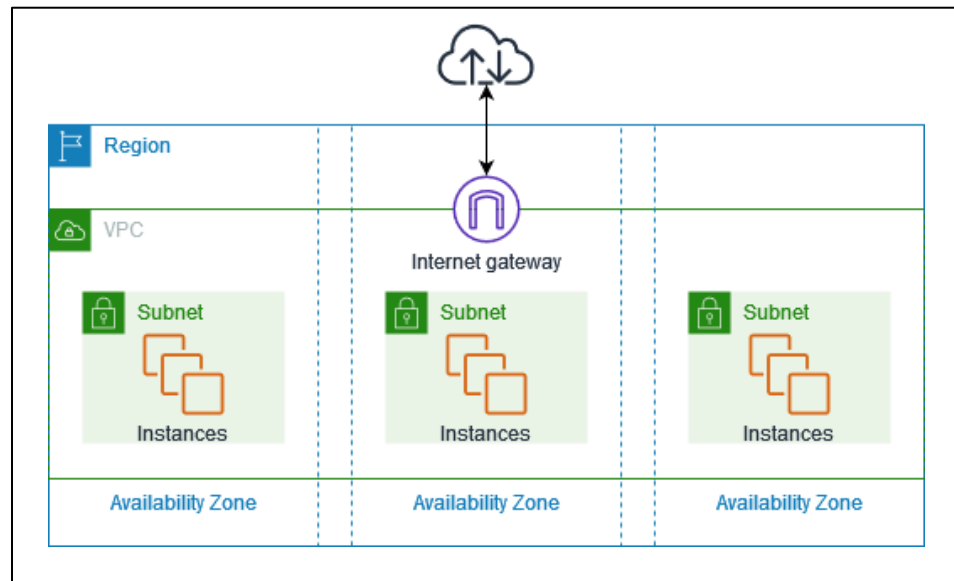


Figure 3.3. VPC example (Amazon VPC⁶)

- **Key-Pair:** A key pair, comprising both a public key and a private key, represents a pair of security credentials employed for verifying the users' identity when establishing a connection to an Amazon EC2 instance. In this setup, Amazon EC2 retains the public key on the users' instance, while they retain control over the private key. In the context of Linux instances, this private key serves as the secure means for SSH access to their instance. Alternatively, in lieu of key pairs, they have the option to utilize AWS Systems Manager Session Manager for connecting to their instance. This method provides an interactive, one-click, browser-based shell, or integration with the AWS Command Line Interface (AWS CLI).
- **AMI:** An Amazon Machine Image (AMI) is a meticulously curated and managed image made available by AWS, containing all the essential data needed to initiate an instance. When launching an instance, specifying an AMI is a mandatory step. If the user needs multiple instances with identical configurations, they can initiate several instances from a single AMI. Conversely, when they need instances with varying configurations, they can utilize distinct AMIs for launching those instances.
- **Launch Template:** It is possible to generate a launch template, which encapsulates the setup details needed for initiating an instance. Launch templates offer a convenient way to store launch parameters, eliminating the need to

⁶ <https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html>

repeatedly specify them when launching instances. For instance, a launch template might include essential information like the AMI ID, instance type, and customary network settings employed for launching instances. When users launch an instance through the Amazon EC2 console, an AWS SDK, or a command line tool, they have the option to designate the specific launch template to employ.

- **Security Groups:** A security group governs the traffic that is permitted to enter and exit the resources it is linked to. For instance, once users associate a security group with an EC2 instance, it takes charge of managing both incoming and outgoing traffic for that instance. It's worth noting that they can only associate a security group with resources located within the same VPC where the security group was established. Upon creating a VPC, a default security group is automatically provided. If needed, they have the flexibility to generate extra security groups for each VPC in their account.

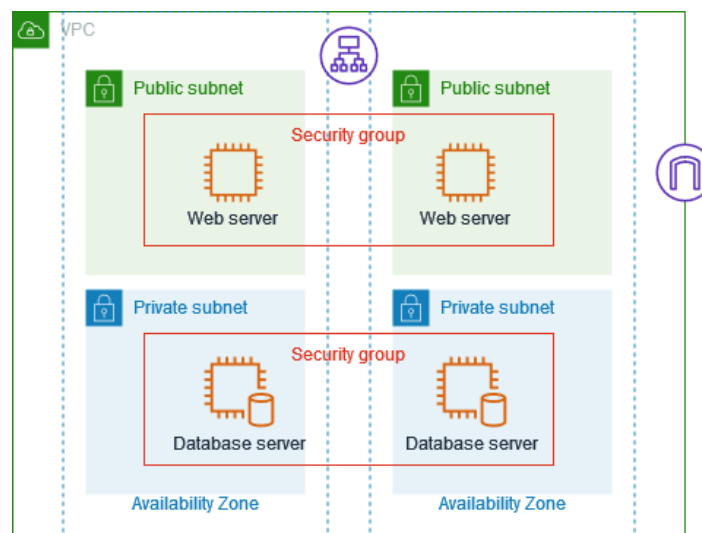


Figure 3.4: Security groups example (Amazon Security Basics⁷)

3.6 API Testing Tool

APIs are pivotal for enabling smooth communication among different applications and services. Given their fundamental role in contemporary software, rigorous testing becomes crucial to ensure their dependability, scalability, and security. This is where the significance of API testing tools becomes apparent.

⁷ <https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html#security-group-basics>

API testing is a procedure used by developers to evaluate the functionality, efficacy, and security of APIs. Before releasing their software, the results of this procedure will inform developers if an API requires problem fixes and patches.

The Apache JMeter software is an open-source, entirely Java-based application from the Apache Software Foundation that is designed to assess performance and stress-test functional behavior. While its initial purpose was testing web applications, it has since evolved to encompass a broader range of testing functions. Apache JMeter may be used to test performance both on static and dynamic resources, Web dynamic applications.

Some of the important characteristics of JMeter to this experiment are as follows:

- It is available freely as open-source software.
- Featuring a user-friendly and intuitive graphical user interface (GUI).
- JMeter is versatile, capable of conducting load and performance tests on various server types, including Web (HTTP, HTTPS), SOAP, Database (via JDBC), LDAP, JMS, Mail (POP3), and more.
- It is a tool that operates seamlessly across different platforms. On Linux/Unix systems, users can initiate JMeter by executing the JMeter shell script, while on Windows, it can be launched by running the jmeter.bat file.
- JMeter offers robust support for Swing and lightweight components (precompiled JAR utilizes javax.swing.* packages).
- Test plans in JMeter are stored in XML format, facilitating the creation and modification of test plans using a simple text editor.
- With its comprehensive multi-threading framework, JMeter enables concurrent sampling by multiple threads and simultaneous sampling of various functions through separate thread groups.
- The extensibility of JMeter allows for the integration of additional functionalities and plugins.
- Beyond load and performance testing, JMeter can also be employed for automated and functional testing of applications.

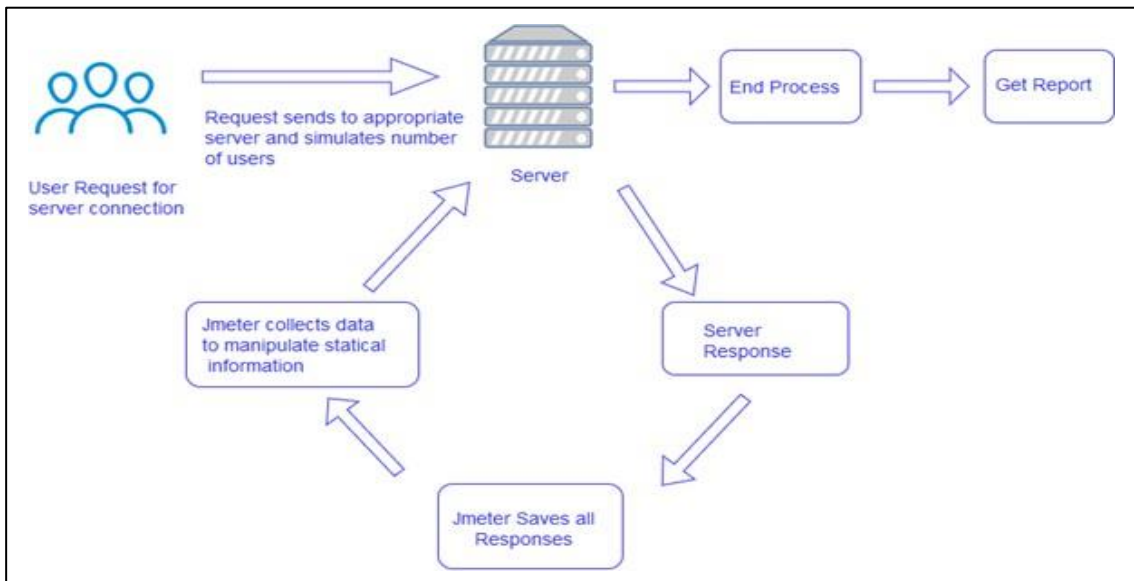


Figure 3.5: JMeter architecture (JMeter Tutorial⁸)

3.7 Fault Injector

As the name suggests, fault injection is a technique for deliberately introducing stress or failure into a system to see how the system responds. Runtime fault injection gained significant traction, particularly within organizations overseeing extensive, intricate, and distributed systems. In 2011, Netflix introduced Chaos Monkey, a tool that intentionally halted compute instances operating in their cloud infrastructure. Chaos Monkey assisted Netflix in confirming the resilience of their workloads to abrupt and unanticipated failures through the random termination of running systems. In 2014, Netflix further advanced this concept with the introduction of their Failure Injection Testing (FIT) platform, which provided a more advanced solution for orchestrating widespread failure scenarios involving multiple teams. These pioneering tools established the foundational principles of modern-day Chaos Engineering.

AWS Fault Injection Simulator (FIS) is a completely managed service that facilitates the execution of fault injection experiments aimed at enhancing an application's performance, visibility, and robustness. FIS streamlines the setup and execution of deliberate fault injection tests spanning various AWS services, enabling teams to gain trust in their application's behavior.

⁸ <https://www.javatpoint.com/jmeter-tutorial>

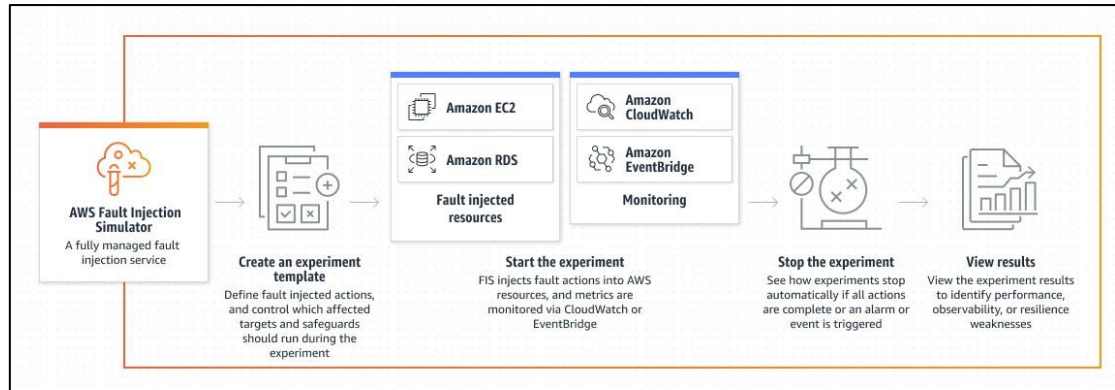


Figure 3.6: FIS explained (Amazon FIS⁹)

It is important to highlight the lack of freely available chaos engineered tools at the time of writing this dissertation. Given the nature of the research and the fault injection requirements, AWS FIS was the only free tool available that could interact with AWS resources in the necessary way (i.e., terminating computing instances to simulate an unexpected fault).

3.8 Expert Interview Design

Three interviews will be conducted with Industry experts in the field of software engineering and cloud infrastructure. The interviews will be conducted separately, so interviewees cannot influence each other and/or steer each other's train of thought in a specific direction. The interviewees will be asked not to speak to each other about this project until after all the interviews are conducted.

The interview process will be as follows, the interviewees will each be presented with an outline of the experiment, and the key results obtained. This will be in a neutral manner. The interviewees will then be asked to comment on the experiment, in terms of how it was designed, how it was implemented, and on the results. And specifically, they will be asked to share their interpretation of the outcomes of the experiment.

The interviewees can be described as follows:

#	Description
I1	Interviewee 1 has over 25 years of experience in the Software Engineering industry, including FinTech, medical systems and the public

⁹ <https://aws.amazon.com/fis/>

	sector. They have also spent over 10 years working directly with cloud infrastructure, including AWS, OpenShift, PCF and Azure.
I2	Interviewee 2 has 8 years of experience in the Software Engineering industry, including FinTech and public sector. They have spent over 3 years working directly with cloud infrastructure, including AWS and OpenShift.
I3	Interviewee 3 has over 10 years of experience in the Software Engineering industry, including banking and public sectors. They have spent over 6 years working directly with cloud infrastructure, including AWS, Azure and OpenShift.

These interviews allowed for reflection and validation of the experiment design, metrics extraction and drawn conclusions from an external perspective, as well as bringing ideas and inspiration for future work. The interviews will be described in detail in chapter 5.8. Zhang, *et al.* (2021) in their paper on ‘Microservice architecture in reality: An industrial inquiry’, conducted a series of interviews with industry experts, therefore this paper found inspiration in their work when planning the interviews. The full presentation used in the interviews and questionnaire can be found in Appendix C, or in Power Point format at <https://github.com/brunorfranco/masterThesis/blob/main/mastersInterviewPresentation.ptx>.

3.9 Experiment Design

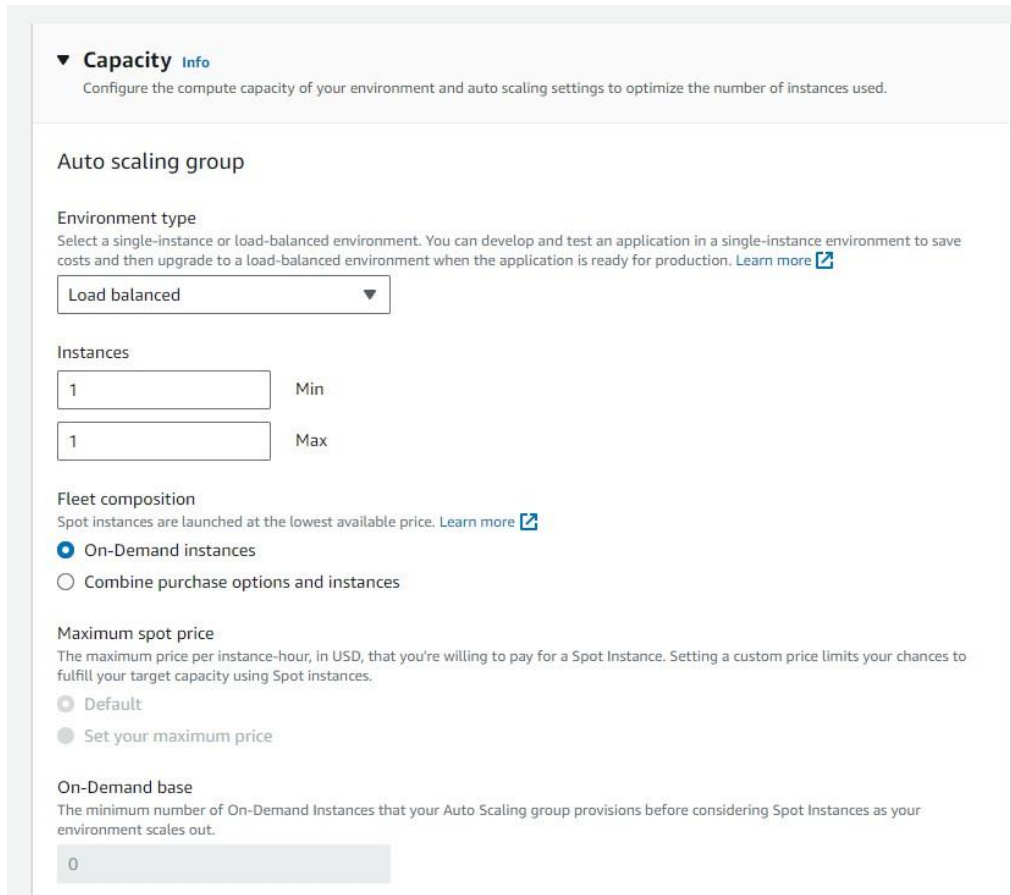
This section will explore the conception of the experiment's design, detailing the encountered challenges throughout the process and culminating in its final state.

3.1.1. Design Challenge

During the implementation of the experiment, multiple challenges were faced, and they will be explored in this section.

Initially, the research intended to use a ‘black box’ AWS service named Elastic Beanstalk, which is a platform designed for the deployment and expansion of web applications and services. By uploading application code, Elastic Beanstalk would seamlessly manage the deployment process, encompassing tasks like capacity provisioning, load balancing, auto scaling, and continuous monitoring of application

health. However, Elastic Beanstalk does not offer fine-grained configuration, including the Auto Scaling configuration required for the ideal experiment, so it could not be utilized in this research. More specifically, Elastic Beanstalk only allows for the minimum and maximum number of instances setup, while the envisioned experiment requires the setup for ‘desired capacity’ and ‘maximum prepared capacity’:



▼ Capacity [Info](#)
Configure the compute capacity of your environment and auto scaling settings to optimize the number of instances used.

Auto scaling group

Environment type
Select a single-instance or load-balanced environment. You can develop and test an application in a single-instance environment to save costs and then upgrade to a load-balanced environment when the application is ready for production. [Learn more](#)

Load balanced ▼

Instances

1 Min

1 Max

Fleet composition
Spot instances are launched at the lowest available price. [Learn more](#)

☒ On-Demand instances

☐ Combine purchase options and instances

Maximum spot price
The maximum price per instance-hour, in USD, that you're willing to pay for a Spot Instance. Setting a custom price limits your chances to fulfill your target capacity using Spot instances.

☐ Default

☒ Set your maximum price

On-Demand base
The minimum number of On-Demand Instances that your Auto Scaling group provisions before considering Spot Instances as your environment scales out.

0

Figure 3.7: Elastic Beanstalk Auto scaling group configuration (Author)

On the applications implementation plan, initially the Author envisioned creating an ‘Industry level standard’ architecture, including a highly resilient and available database in AWS, a Multi AZ RDS MySQL database with an extra read-only replica, but after further consideration, it came to light that for the purpose of this research, this database setup would not help validate nor disprove the hypothesis, and it would incur extra costs on the AWS billing.

A similar scenario was also faced in terms of the application setup. Initially, it was envisioned that two Java applications would be created, a ‘frontend-application’, and a ‘backend-application’, but after further consideration, it was realized that when injecting

faults into instances for the same service, it would not matter having two different services, so the author has elected to proceed with only the ‘backend-application’.

Both applications codes can be found at: [brunorfranco/masterThesis: Folder to hold all the necessary code and configuration for my personal Masters thesis \(github.com\)](https://github.com/brunorfranco/masterThesis-Folder-to-hold-all-the-necessary-code-and-configuration-for-my-personal-Masters-thesis)

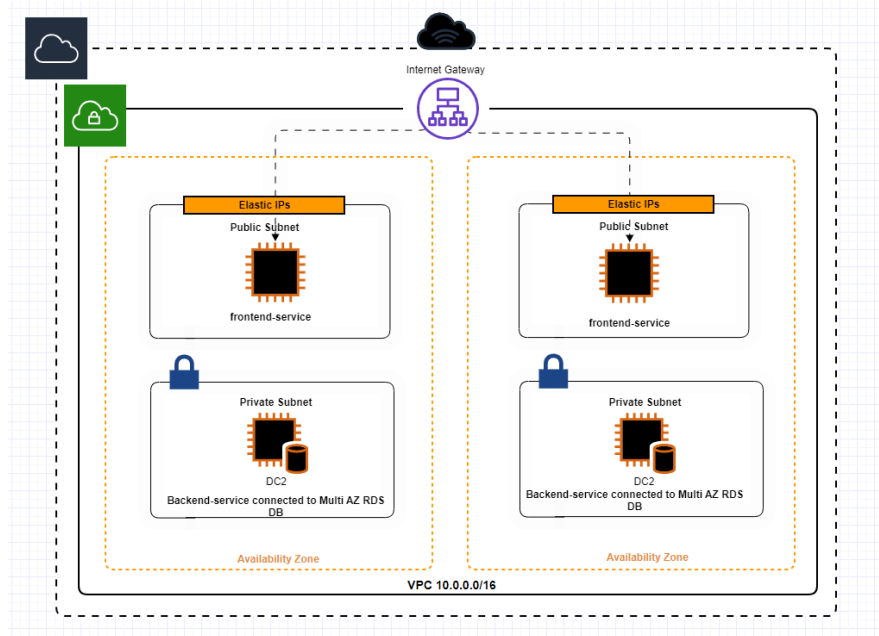


Figure 3.8. Initial Design (Author)

The initial research design had scope that the AWS environment cannot satisfy.

A study by Portent (<https://www.portent.com/blog/analytics/research-site-speed-hurting-everyones-revenue.htm>) discovered that the average website load time in 2023 is 2.5 seconds, therefore the initial research design visioned for AWS to upkeep a self-healing SLO of two seconds. However, after the preliminary test, it became clear that two seconds was too ambitious, so the time expectation was modified to sixty seconds, which was inspired by Frincu, *et al.* (2011), where recovery time findings were around 60s for 10 modules using on-demand deployment.

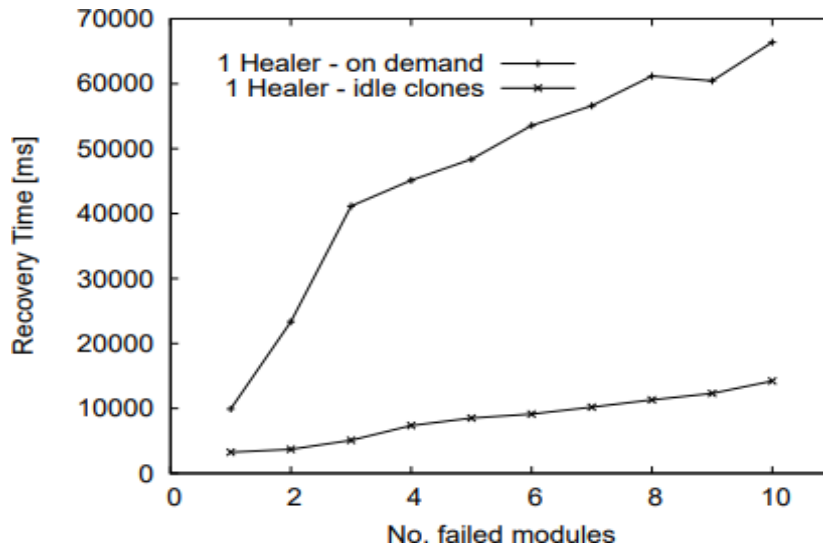


Figure 3.9. Recovery time vs. number of failed modules (Frincu, et al., 2011)

3.9.1 Final Design

Given Elastic Beanstalk limitations and the advice from experts in the field, the overall cloud architecture will manually be created as follows:

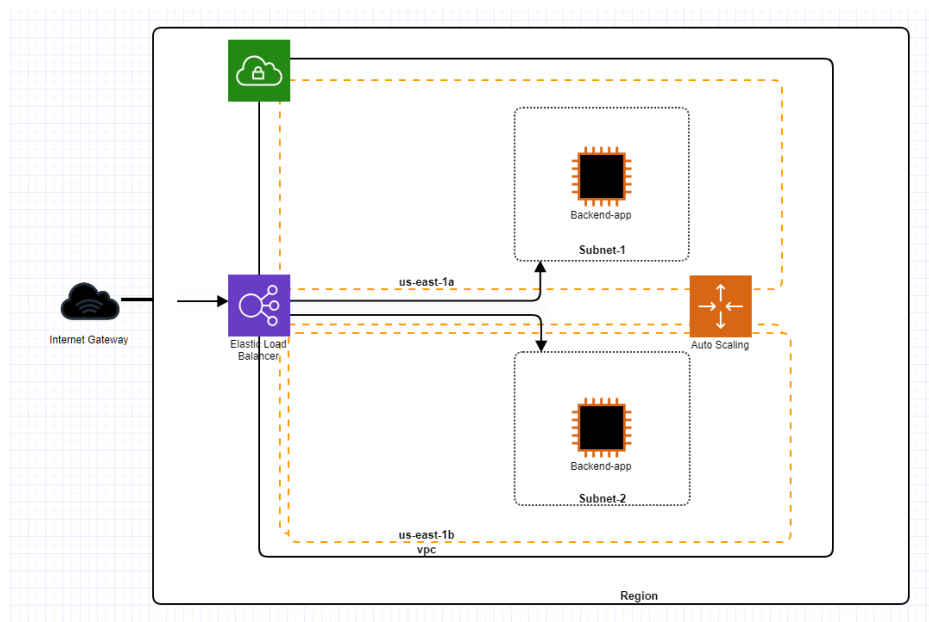


Figure 3.10. Final Design (Author)

For this experiment, a Test Plan will be created in Apache JMeter containing:

- One single Thread Group (as this is not a stress-test),
- HTTP Request Sampler pointing to the Elastic load balancer for the backend-

service,

- Graph and Table listeners.
- Constant Timer with 1000 milliseconds Thread Delay

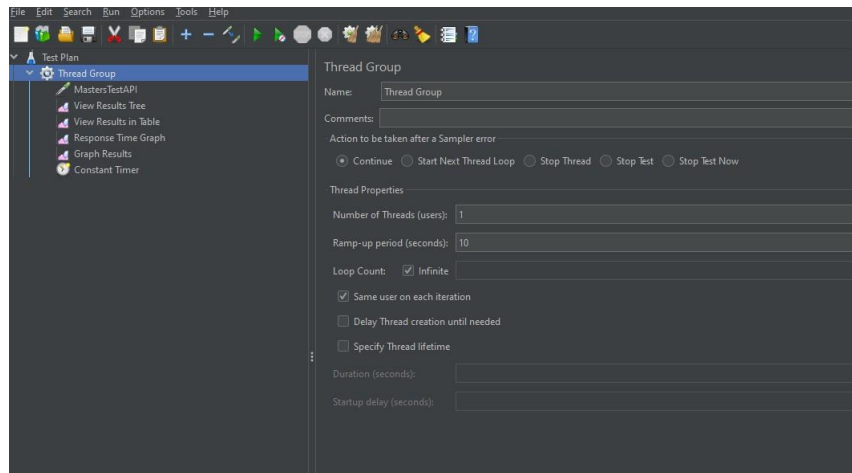


Figure 3.11. JMeter Test Plan Configured (Author)

An AWS Fault Injector setup will also be created to disable both EC2 instances running the backend-service application, and the JMeter Test Plan will then collect information during the fault injection.

The JMeter will collect the time of the last successful request to the API before the fault injection, and the first successful request after the fault injection. The delta of the two will be used as the time that the cloud architecture needed to heal itself.

32	18:16:35.765	Thread Group 1-1	MastersTestAPI	108	651	182	108	0
33	18:16:36.884	Thread Group 1-1	MastersTestAPI	101	651	182	101	0
34	18:16:37.993	Thread Group 1-1	MastersTestAPI	106	651	182	106	0
35	18:16:39.110	Thread Group 1-1	MastersTestAPI	105	651	182	105	0
36	18:16:40.228	Thread Group 1-1	MastersTestAPI	106	651	182	106	0
37	18:16:41.344	Thread Group 1-1	MastersTestAPI	100	651	182	100	0
38	18:16:42.445	Thread Group 1-1	MastersTestAPI	105	651	182	105	0
39	18:16:43.552	Thread Group 1-1	MastersTestAPI	118	651	182	118	0
40	18:16:44.673	Thread Group 1-1	MastersTestAPI	60085	292	182	60085	0
41	18:17:45.765	Thread Group 1-1	MastersTestAPI	900	651	182	900	110
42	18:17:47.673	Thread Group 1-1	MastersTestAPI	104	651	182	104	0

Figure 3.12. JMeter Result in Table view (Author)

Once the FIS setup is completed and the targeted EC2 instances are terminated, an extra five minutes will be allowed for the Auto Scale policy to spin up extra compute instances, then the JMeter test plan will be stopped.

Following Ali Naqvi, *et al.* (2022) experiment, their experiment was executed in two phases, eight times each, our experiment will be executed in four different phases

(differentiated by auto-scaling configurations), eight times each as well:

5. No warm pooling.
6. Warm pooling with one instance on 'Stopped' state.
7. Warm pooling with one instance on 'Running' state.
8. Warm pooling with one instance on 'Hibernated' state.

Therefore, in total, the experiment will be executed thirty-two times, and any outliers will be investigated carefully and re-executed. An experiment execution will be considered an outlier if the cloud healing time has a 50% variation from the previous two executions median (except for the first and second executions, which will be evaluated against the following two executions). A final report will be produced from the thirty-two results, separated by the four configuration variations.

3.10 Conclusions

This chapter has discussed the main technologies and steps necessary to conduct the proposed experiment, collect the data and evaluate it.

The results should produce data to either support or contradict the notion that chaos engineered techniques will degrade the proposed microservice architecture beyond a sixty-seconds self-healing time, as well as compare results between various auto scaling configurations.

Collecting data with an independent tool (JMeter), from outside of the AWS ecosystem, will remove any bias regarding reliability of the results.

A fault injection tool from outside of the AWS ecosystem would have been preferred. However, no free tool that could interact with AWS resources was found, therefore AWS FIS was chosen as the only feasible tool.

4 DEVELOPMENT PROCESS

“Creating new paths requires moving old obstacles.”

– Anthony D. Williams, *Inside the Divine Pattern*

4.1 Introduction

In this chapter, the implementation of the microservice application, the cloud architecture, the fault injection setup, and the API testing will be discussed. These are all based on the designs outlined in Chapter 3. Following this implementation process, four variations of the experiment are executed eight times each. The results these experiments generate will be used for resiliency evaluation purposes. Finally, the difference in results from the four experiment variations will be analyzed. Any problems encountered during the development will be discussed and the technologies used will be judged on their effectiveness during implementation. All of the code written for this dissertation, as well as any generated configuration (user-data file, JMeter test plan, FIS template, etc.) can be found at the following Github location: <https://github.com/brunorfranco/masterThesis>.

The diagram below shows all the necessary steps in the correct sequence to achieve the final state within the AWS platform to proceed with the experiment.

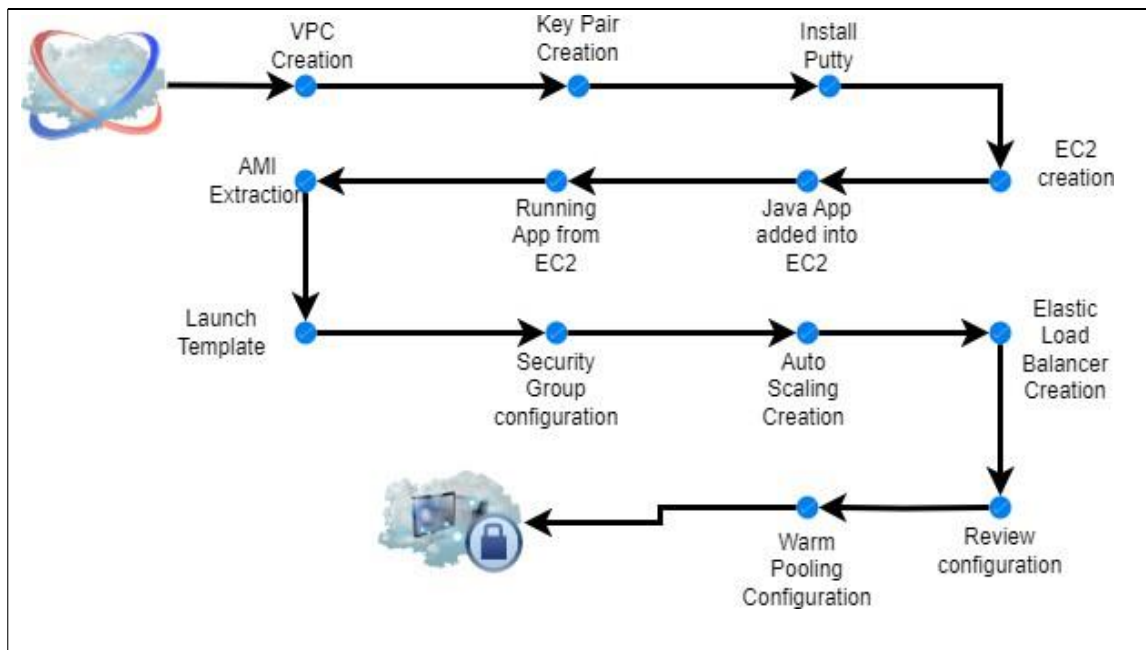


Figure 4.1. AWS Diagram with all steps (Author)

4.2 Java Micro-service Implementation

The ‘backend-service’ Java application was developed and compiled in JavaSE-17, in the Eclipse IDE version 2023-09 (4.29.0). It was built with maven, Spring Boot 3.1.2, Spring Boot Starter Data JPA, Spring boot Starter Web and H2 database. It contains a single model called RandomEntry, which maps to a database table named ‘RandomEntryTable’, with two columns, ‘id’ and ‘randomValue’. The application also exposes a Rest API under port 8082, URL ‘/api/entries’ which returns all the rows of the RandomEntryTable in JSON format. When the application starts up, it executes a command to insert ten random rows into the in-memory database. Once the application is running, the database can be interacted with from the ‘/h2-ui’ URL. The application can be executed by running the ‘BackendServiceApplication.java’ class. The application’s code can be found in its entirety at:

<https://github.com/brunorfranco/masterThesis/tree/main/backend-service>.

4.3 AWS Cloud Architecture implementation

This section will explore the steps to create the self-healing cloud architecture that will be challenged by the fault injection tool of choice and note if such architecture will be able to heal itself within sixty seconds. The main goal in this section is to configure the necessary components mentioned in the diagram below:

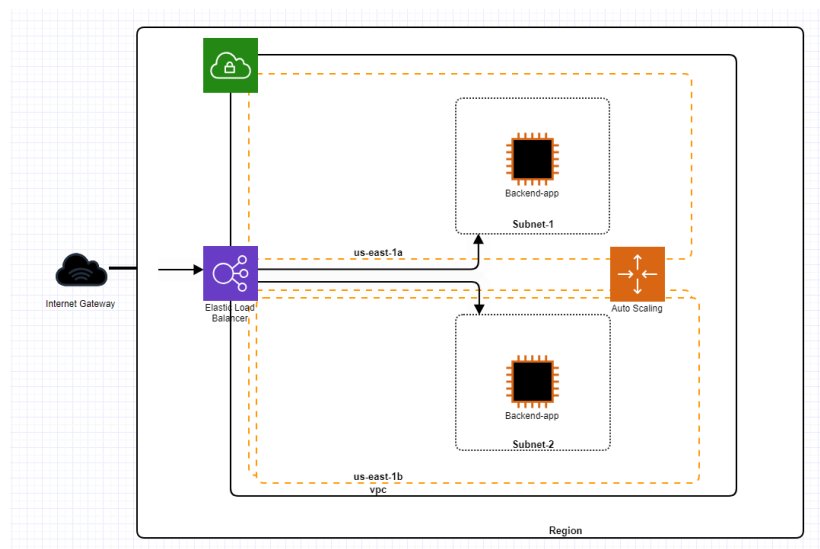


Figure 4.2. AWS Cloud Architecture Designed (Author)

For more details on how to setup any of steps described from section 4.3.1 to 4.3.10, please refer to Appendix B, section 2.

4.3.1 VPC Creation

The objective of this step is to set up a virtual private cloud (VPC), so that other cloud components can be created safely within it.

Once logged into the AWS console, the ‘Create VPC’ wizard steps were followed:

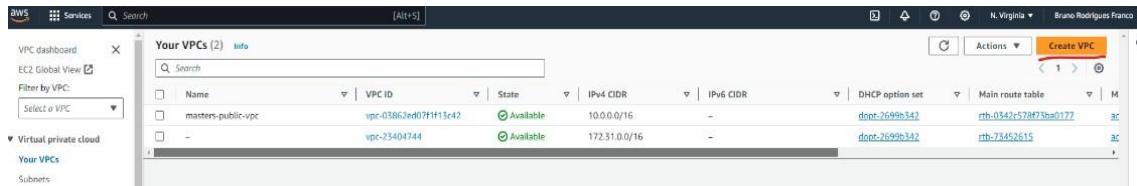


Figure 4.3. AWS Console – VPC (Author)

4.3.2 Key Pair Creation

The objective of this step is to create an AWS Key Pair, so that it can be used for users to safely use the Secure Shell Protocol (SSH) to connect into compute instances. A key Pair is a set of security credentials, consisting of a public key and a private key, and it is used to verify users' identities when they connect to an Amazon EC2 instance. To proceed with the setup, a key pair needs to be created. Once logged into the AWS console, navigate to the ‘Key Pair’ section, and follow the ‘Create key pair’ wizard.

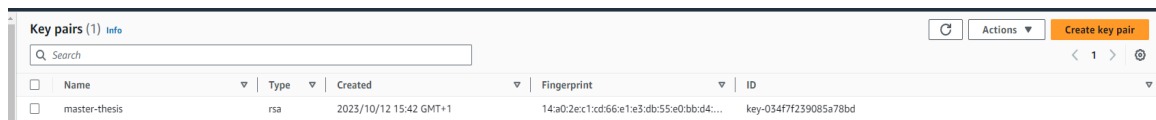


Figure 4.4. AWS Console – Key pairs (Author)

4.3.3 (Windows users) Install Putty

The objective of this step is to download and install Putty, so that it is possible to SSH into compute instances remotely to run the Java application. If the format file in the download was .pem, then Puttygen can be used to convert it to .ppk.

4.3.4 EC2 Creation

The objective of this step is to create EC2 (Elastic Cloud Compute) instances, so that they can be used to deploy and execute the Java Application

From the EC2 dashboard, the ‘Launch Instance’ button was clicked and instructions within the wizard were followed to launch an ‘Amazon Machine Image’:

EC2 > Instances > Launch an instance

Launch an instance [Info](#)

Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags [Info](#)

Name

 [Add additional tags](#)

Figure 4.5. AWS Console – EC2 Launch (Author)

This will spin up an Amazon Linux 2023 x86_64 HVM kernel-6.1. The key pair created on step 4.3.2 was selected under the ‘Key pair (login)’ section. Under the ‘Network settings’, the ‘Create security group’ option was selected to simplify the configuration, given that comes with SSH traffic allowed by default. The tick-box to ‘Allow HTTP traffic from the internet’ was also selected, as a Rest API endpoint will be exposed by the Java application. Finally, the instance was launched.

4.3.5 Adding Java Application file into the EC2 instance.

The objective of this step is to copy the Java application .jar file into the EC2 instance, so that it be deployed. To add the .jar file into the EC2 instance, the WinSCP tool was used, version 6.1.2, build 13797 2023-09-19. For details on how to configure the use of a .ppk key file into WinSCP and SSH into the EC2 instance from the previous step, please refer to Appendix B, section 2.5. The backend-service-0.0.1-SNAPSHOT.jar file from the Java application ‘target’ folder was then copied and pasted into the EC2 instance, under the ‘/home/ec2-user/’ folder:

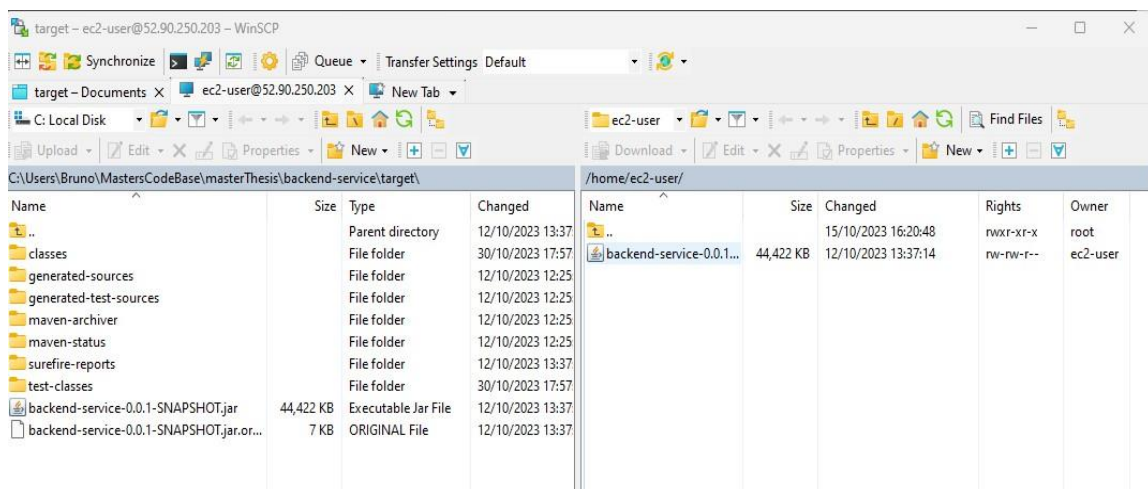


Figure 4.6. WinSCP Console (Author)

4.3.6 Connecting into EC2 through Putty and running the application.

The objective of this step is to connect to the EC2 instance via the command line interface, so that the Java application can be executed. Once the .jar file was loaded into the EC2 instance, then it was time to SSH into the instance through the Putty command line and run it. To achieve that, Putty version 0.79 was used. To open a new session from Putty, the Public IPv4 address from the instance in the AWS console was captured, then inserted into the Putty Host Name with the port 22. Under ‘Connection’, the author opened ‘SSH’, then ‘Auth’, then ‘Credentials’, then browsed and selected the .ppk key file that was generated on step 4.3.2. The SSH connection was established then:

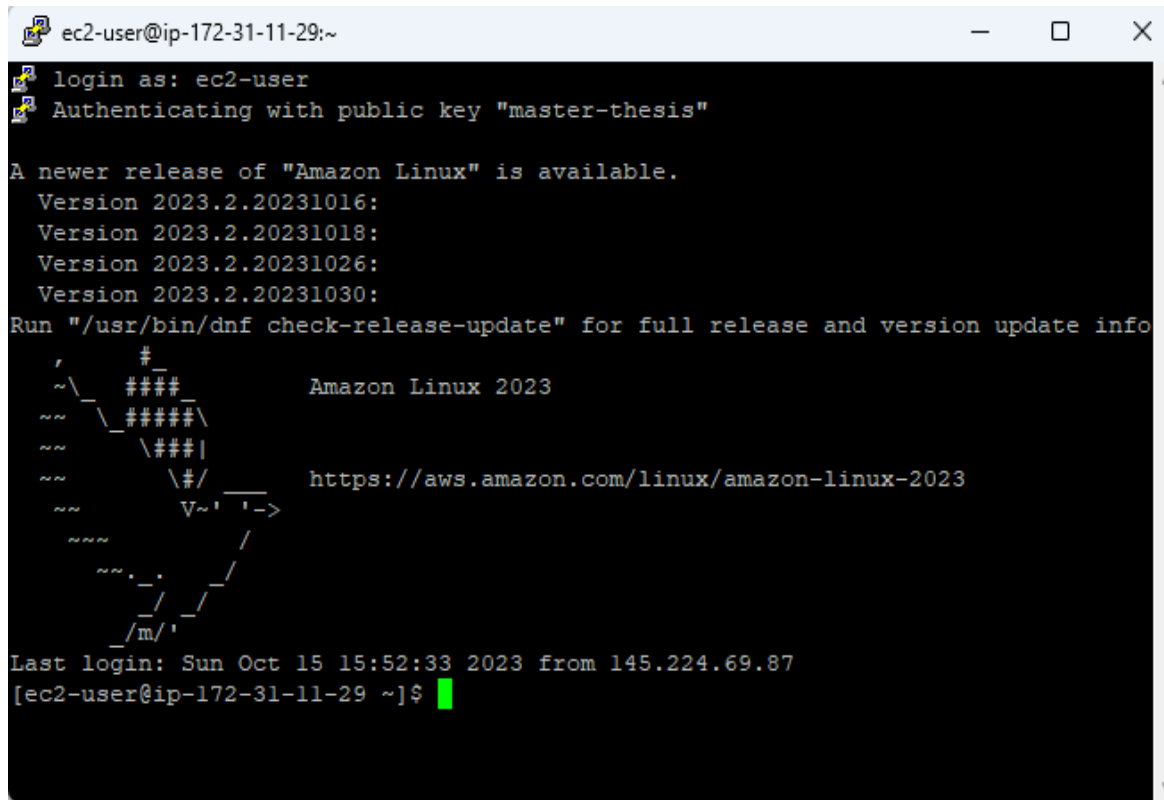


Figure 4.7. Putty Console – Login (Author)

If the connection times out, please refer to Appendix B, section 6 for troubleshooting advice. Once in, it was verified that the .jar file is in place and accessible by executing an ‘ls -l’ command:

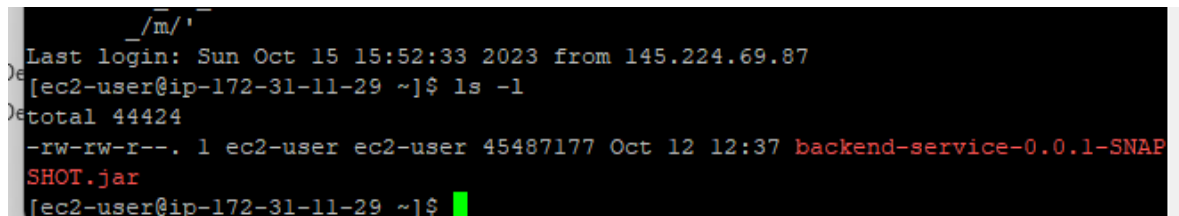


Figure 4.8. Putty Console – Jar File (Author)

After that was confirmed, the application was started by executing “java -jar /home/ec2-user/backend-service-0.0.1-SNAPSHOT.jar &”.

4.3.7 AMI Extraction

The objective of this step is to create a Machine Image, so that the image can be used further on as part of the Auto Scaling group. That way, the Auto Scaling group will be able create pre-configured instances from this AMI when spinning up new on-demand instances.

That was achieved from the ‘EC2 Dashboard’ in the AWS console, by selecting the healthy EC2 instance, then selecting ‘Actions’, then ‘Image and templates’, then ‘Create image’ and following the wizard:

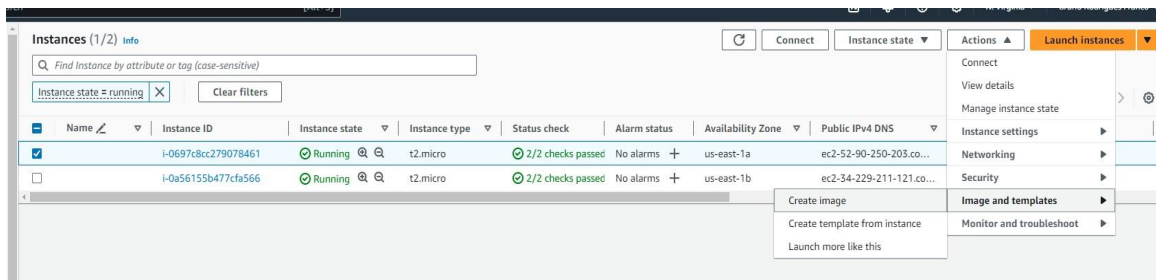


Figure 4.9. AWS Console – Creating Image (Author)

4.3.8 Launch template

The objective of this step is to create a Launch Template, so that it can hold the necessary configuration for the Auto Scaling group. A Launch Template can be created from the EC2 dashboard, then ‘Auto Scaling’, then ‘Auto Scaling Groups’, then the ‘Launch Templates’ option:

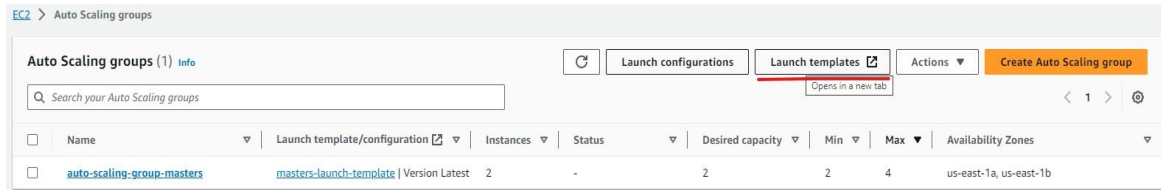


Figure 4.10. AWS Console – Launch Templates (Author)

4.3.9 Auto Scaling group and Elastic Load Balancer

The objective of this step is to create an Auto Scaling group and Elastic Load Balancer, so that AWS can create compute instances on-demand and route incoming traffic from the Load Balancer into said instances. With the Launch template in place, then the Auto Scaling group can be created through the EC2 dashboard, ‘Auto Scaling’, ‘Auto Scaling Groups’, ‘Create Auto Scaling group’ wizard:

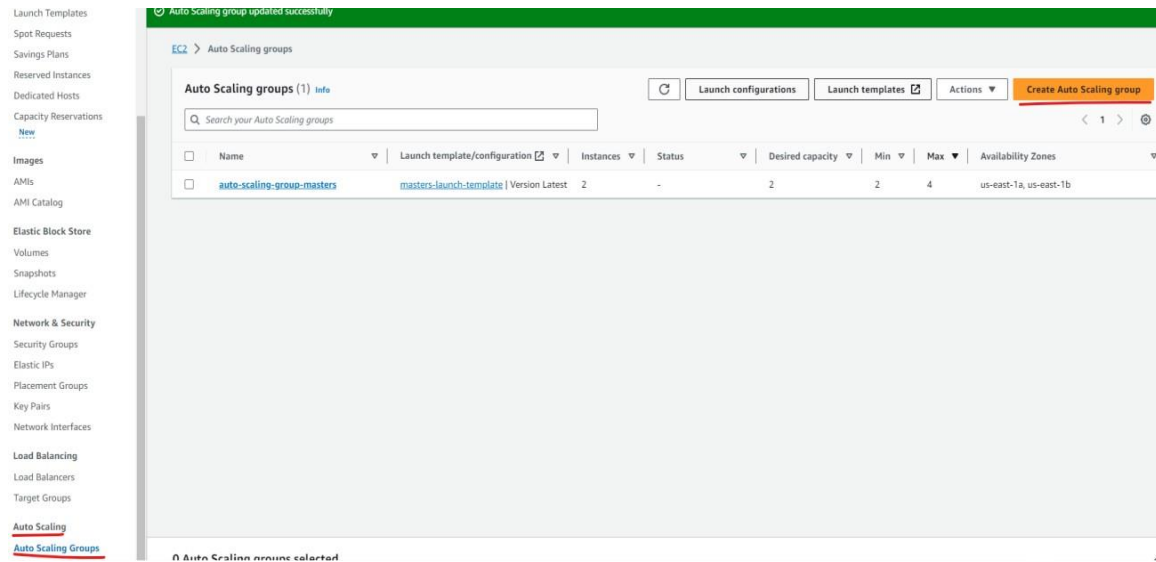


Figure 4.11. AWS Console – Auto Scaling group (Author)

4.3.10 Warm Pooling

The objective of this step is to set up warm pools within the Auto Scaling group, so that higher resilience is achieved.

As mentioned previously, Auto Scaling groups allow for different configuration variations, which have effects on recovery time of the service, so they will be explored as part of the experiment:

1. No warm pooling.
2. Warm pooling with one instance on ‘Stopped’ state.
3. Warm pooling with one instance on ‘Running’ state.
4. Warm pooling with one instance on ‘Hibernated’ state.

They can be configured under the ‘Auto Scaling Group’ section in the EC2 dashboard, under the ‘Instance management’ tab. The details of each warm pooling setup will be discussed under section 4.6 “Experiment Execution”.

4.4 JMeter Test Plan Implementation

For this experiment, a Test Plan was created in Apache JMeter version 5.6.2

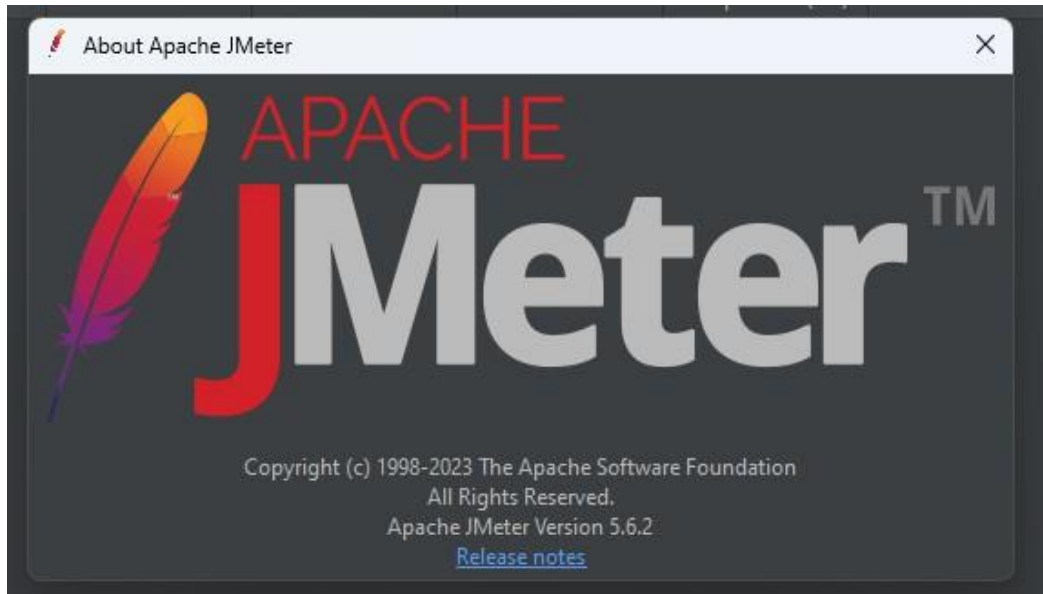


Figure 4.12. JMeter Logo (Author)

This tool is maintained by the Apache Software Foundation and is free and open source. At the time of this writing, the tool can be downloaded from https://jmeter.apache.org/download_jmeter.cgi

The creation and configuration of the Test Plan can be seen on Appendix B, section 3. The full configuration can be found in .jmx format for ease of importing at <https://github.com/brunorfranco/masterThesis/blob/main/JMeterTests/APITesting.jmx>

4.5 AWS Fault Injection Simulator (FIS) Creation

The author has opted to utilize the AWS FIS as the chaos engineering tool of choice. The tool can be accessed through the AWS FIS console:

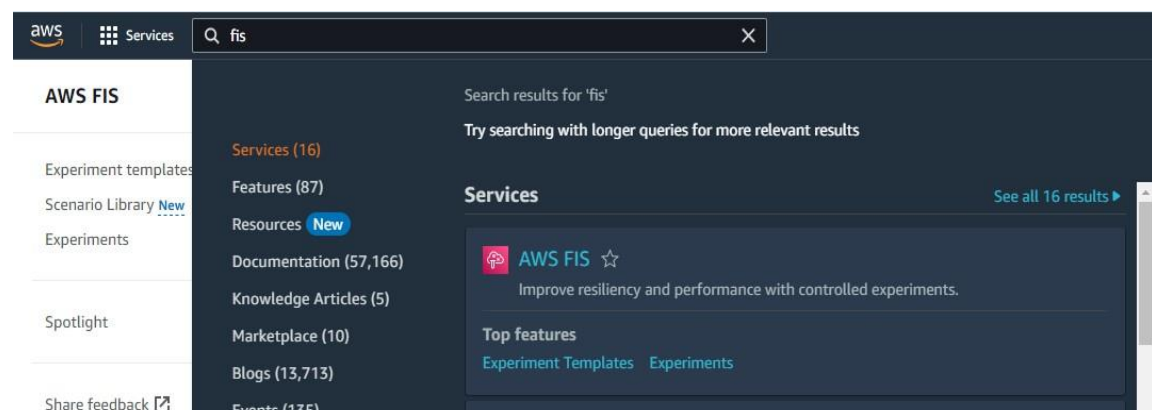


Figure 4.13. AWS FIS Menu option (Author)

The creation of a FIS experiment template was done through the ‘Create experiment template’ wizard:



Figure 4.14. AWS FIS Console (Author)

The details of the FIS setup can be found at Appendix B, section 4.

The command line configuration can be found at the following link to facilitate the creation of the FIS experiment:

<https://github.com/brunorfranco/masterThesis/tree/main/FISExperimentTemplate>

4.6 Experiment Execution

As mentioned earlier, each warm pooling configuration was executed eight times each. Before any experiment executions, it was verified that there were two healthy EC2 instances running and serving requests through the Load Balancer. A timer was initiated. As soon as the timer reached thirty seconds, the JMeter Test Plan was initiated to start collecting metrics:

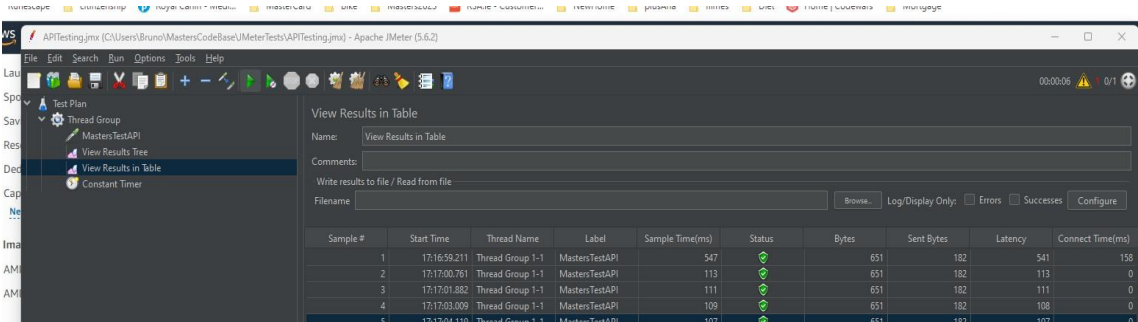


Figure 4.15. JMeter View Results (Author)

Once the timer reached one minute, the FIS experiment was initiated to shut down running EC2 instances:

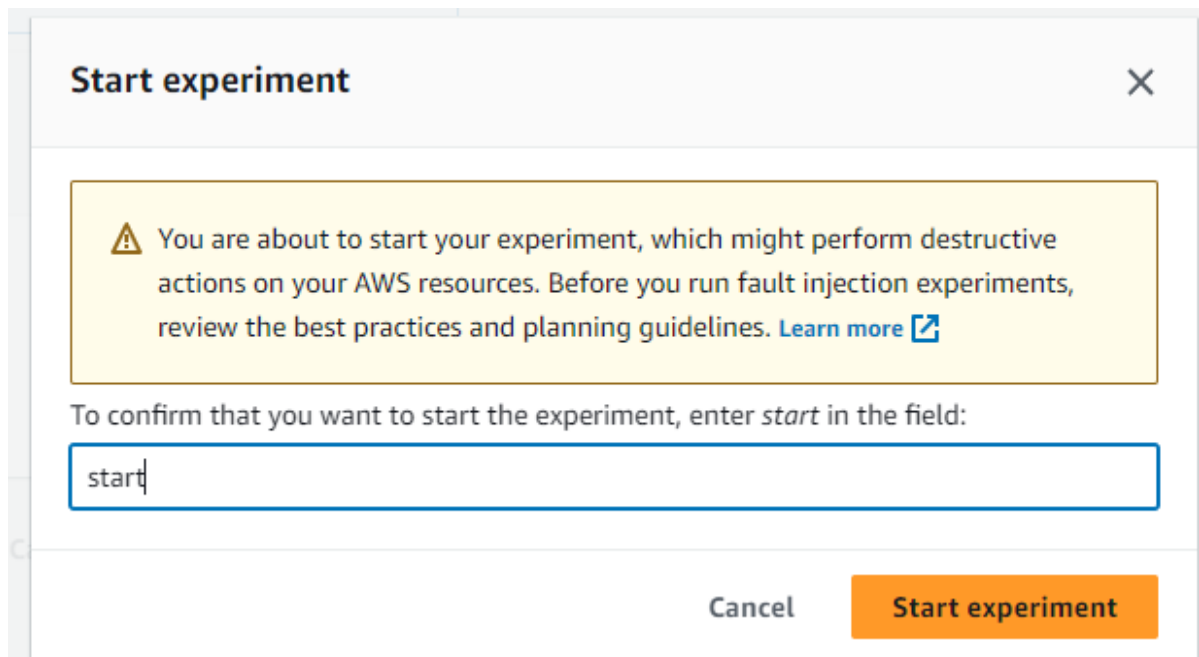


Figure 4.16. AWS FIS – Starting experiment (Author)

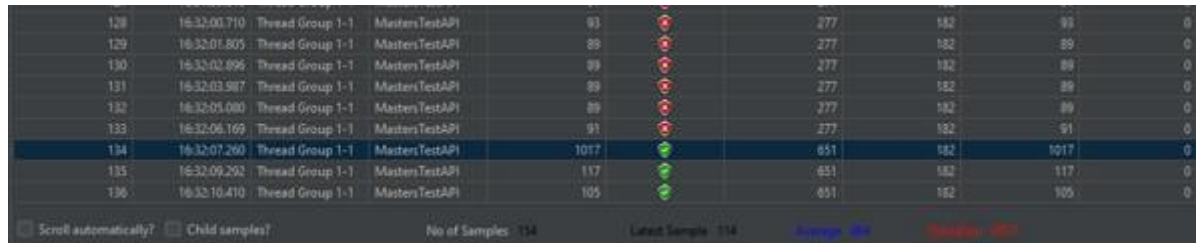
The JMeter Test Plan would soon (~15 minutes after the FIS startup) indicate that the requests were no longer responding successfully, so the author took note up to the millisecond of the last time a request was successful before starting to fail, via the ‘View Result in Table’ in JMeter.

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
29	16:29:11.752	Thread Group 1-1	MastersTestAPI	101	Success	651	182	103	0
30	16:29:12.840	Thread Group 1-1	MastersTestAPI	113	Success	651	182	113	0
31	16:29:13.965	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0
32	16:29:15.085	Thread Group 1-1	MastersTestAPI	102	Success	651	182	102	0
33	16:29:16.201	Thread Group 1-1	MastersTestAPI	99	Success	651	182	99	0
34	16:29:17.300	Thread Group 1-1	MastersTestAPI	103	Success	651	182	103	0
35	16:29:18.403	Thread Group 1-1	MastersTestAPI	104	Success	651	182	103	0
36	16:29:19.511	Thread Group 1-1	MastersTestAPI	108	Success	651	182	108	0
37	16:29:20.631	Thread Group 1-1	MastersTestAPI	99	Success	651	182	99	0
38	16:29:21.733	Thread Group 1-1	MastersTestAPI	104	Success	651	182	104	0
39	16:29:22.846	Thread Group 1-1	MastersTestAPI	107	Success	657	182	107	0
40	16:29:23.966	Thread Group 1-1	MastersTestAPI	110	Success	657	182	109	0
41	16:29:25.085	Thread Group 1-1	MastersTestAPI	109	Success	651	182	108	0
42	16:29:26.203	Thread Group 1-1	MastersTestAPI	108	Success	651	182	108	0
43	16:29:27.325	Thread Group 1-1	MastersTestAPI	100	Success	651	182	100	0
44	16:29:28.433	Thread Group 1-1	MastersTestAPI	111	Success	651	182	101	0
45	16:29:29.551	Thread Group 1-1	MastersTestAPI	60082	Failure	292	182	60082	0
46	16:30:30.642	Thread Group 1-1	MastersTestAPI	205	Failure	337	182	205	115
47	16:30:31.857	Thread Group 1-1	MastersTestAPI	89	Failure	337	182	89	0

Figure 4.17. JMeter – Last success before fault (Author)

AWS auto scaling realized that it did not have the minimum required number of instances as part of its group, so it spun up a healthy instance, then subsequently a second one separately (as a mechanism to avoid spinning up extra unnecessary instances).

As soon as the new instances were assigned to the Load Balancer and the JMeter Test Plan stopped failing and started receiving successful responses back again, then the author also took note of the time up to milliseconds of the first successful response after the fault injection:



No	Time	Thread	Group	Test	Latency	Response	Size	Code
128	16:32:00.710	Thread Group 1-1	MastersTestAPI		93	277	182	93
129	16:32:01.805	Thread Group 1-1	MastersTestAPI		89	277	182	89
130	16:32:02.896	Thread Group 1-1	MastersTestAPI		89	277	182	89
131	16:32:03.987	Thread Group 1-1	MastersTestAPI		89	277	182	89
132	16:32:05.080	Thread Group 1-1	MastersTestAPI		89	277	182	89
133	16:32:06.169	Thread Group 1-1	MastersTestAPI		93	277	182	93
134	16:32:07.260	Thread Group 1-1	MastersTestAPI		1017	651	182	1017
135	16:32:09.292	Thread Group 1-1	MastersTestAPI		117	651	182	117
136	16:32:10.410	Thread Group 1-1	MastersTestAPI		105	651	182	105

Figure 4.18. JMeter – First success after fault (Author)

With both entries noted, the simple following subtraction was used to calculate the time it took for AWS to self-heal its system:

Self-Heal Time = Time of the first successful request after fault injection – Time of the last successful request before fault injection

4.6.1 No Warm Pooling Configuration

Before starting the execution of the experiment, the warm pooling configuration was verified under the EC2 console, Auto Scaling section, Instance management tab:

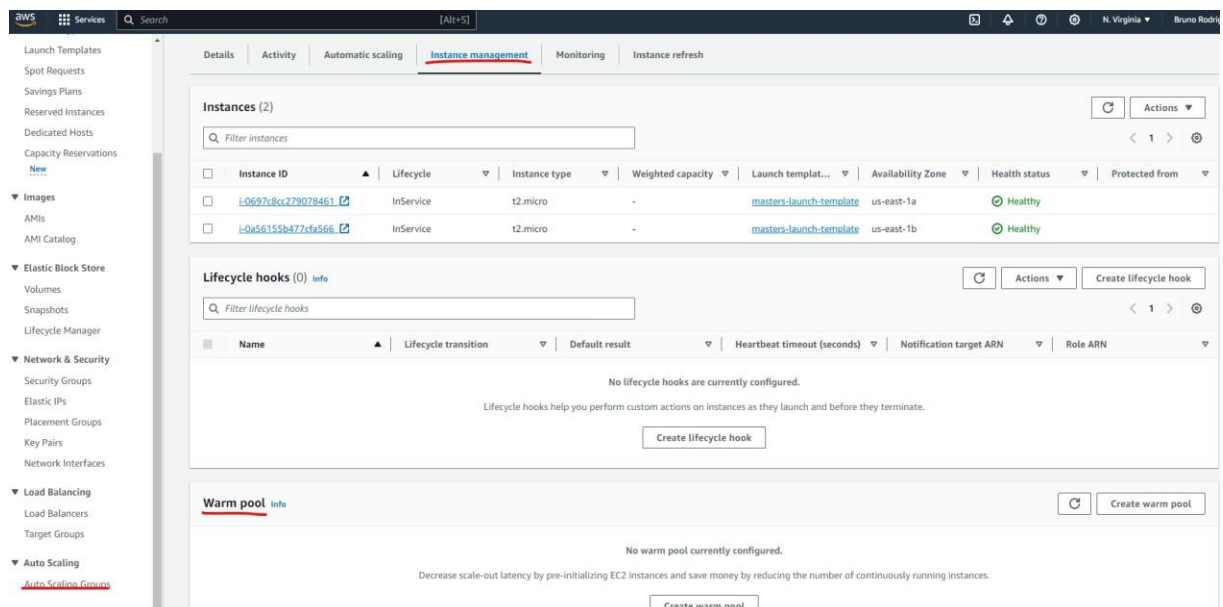


Figure 4.19. AWS Warm Pool Console (Author)

No warm pooling required no extra steps, given that it is the default setup for a newly created

Auto Scaling group.

4.6.2. Stopped Warm Pooling Configuration

From the Instance management tab, the ‘Create Warm Pool’ was clicked, and configured as follows:

Edit warm pool

Warm pool instance state

The state that the warm pool instances will be kept in when they are idle and not in service.

Stopped

Keep a warm pool of stopped instances to reduce your compute cost. You will only be charged fo...

Minimum warm pool size

The minimum number of instances that should always be in the warm pool.

1instance

Max prepared capacity

The maximum number of instances that can be in the warm pool and Auto Scaling group at the same time.

☒ Equal to the Auto Scaling group's maximum capacity

Any updates to the group's maximum capacity will automatically apply to the warm pool's max prepared capacity.

☐ Define a set number of instances

Enter a set number of instances to use for the warm pool's max prepared capacity.

Cancel

Save changes

Figure 4.20. AWS Stopped Warm Pool Window (Author)

After a few minutes, the new instances that were part of the warm pool could be verified under the ‘Warm pool instances’ section:

Warm pool Info

Current warm pool size

2 instances

Minimum warm pool size

1 instance

Max prepared capacity

4 instances (Equal to group's maximum capacity)

Warm pool instance state

Stopped

Status

-

Warm pool instances (2)

Filter warm pool instances

Instance ID	Lifecycle	Instance type	Launch template/configurati...	Availability Zone	Health status
i-073295a7003e9b9b9	Warm:Stopped	t2.micro	masters-launch-template Version	us-east-1a	Healthy
i-0f7325978b3bde81e	Warm:Stopped	t2.micro	masters-launch-template Version	us-east-1b	Healthy

Figure 4.21. AWS Stopped Warm Pool Instances (Author)

4.6.3 Running Warm Pooling Configuration

From the Instance management tab, the ‘Create Warm Pool’ button was clicked, and

configured as follows:

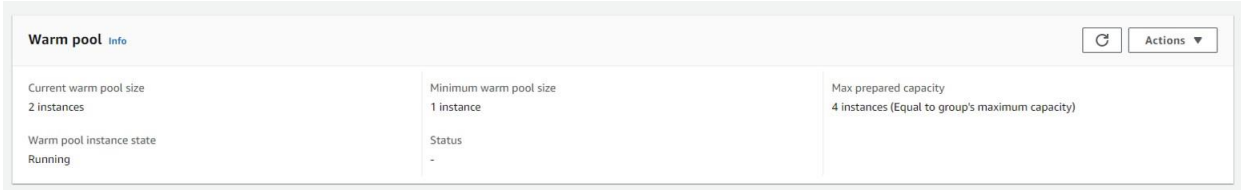


Figure 4.22. AWS Running Warm Pool Console (Author)

After a few minutes, the new instances that were part of the warm pool could be verified under the ‘Warm pool instances’ section:

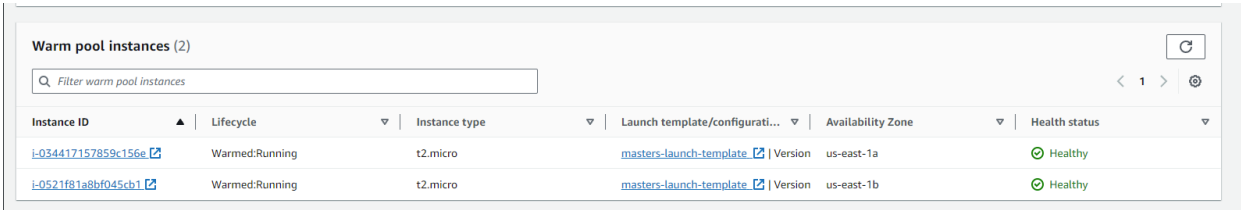


Figure 4.23. AWS Running Warm Pool Instances (Author)

4.6.4 Hibernated Warm Pooling Configuration

Hibernated Warm Pooling was introduced in February 2022, and it required extra security configuration¹⁰.

The AMI behind the auto scaling group is required to have its block storage (ELB) encrypted for it to be able to be added to a hibernated warm pool:

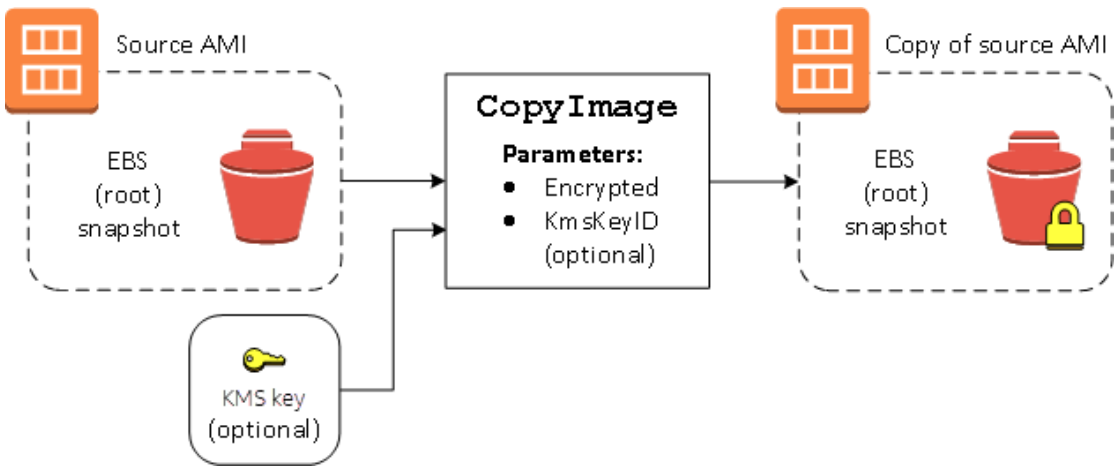


Figure 4.24. EBS Encryption (Amazon Blog Posting¹¹)

¹⁰ <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/hibernating-prerequisites.html>

¹¹ [Amazon EBS encryption - Amazon Elastic Compute Cloud](#)

It was initially decided to create an independent symmetric key under the KMS console (Key Management Service):

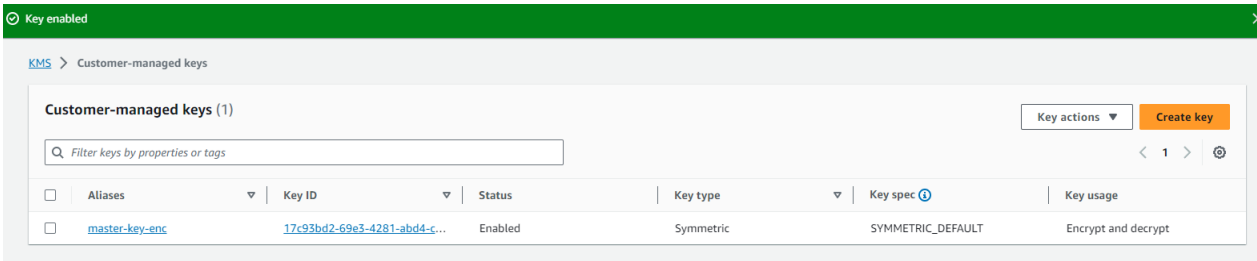


Figure 4.25. AWS Console – Key Creation (Author)

From the AMIs console, the existing AMI that was used throughout the other experiments was selected, then ‘Actions’, then ‘Copy’:

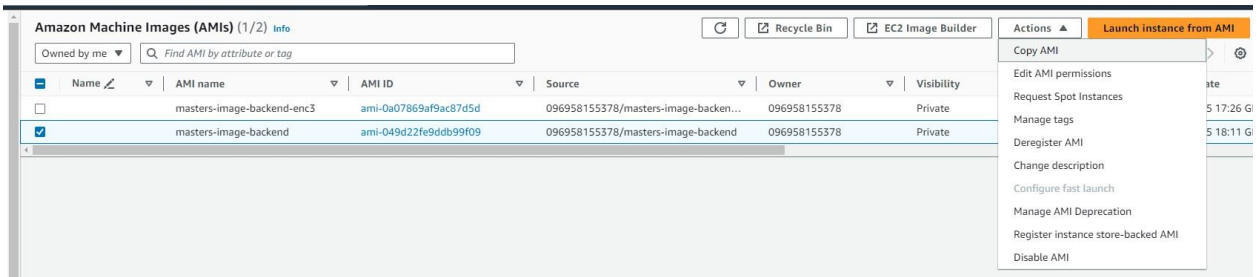



Figure 4.26. AWS Console – AMI Creation (Author)

On the setup page, the ‘Encrypt EBS snapshot of AMI copy’ was then selected and under the KMS key field, the newly created symmetric key was selected:

Copy Amazon Machine Image (AMI)

Original AMI ID
 ami-049d22fe9ddb99f09

AMI copy name

AMI copy description


Destination Region
 A copy of the original AMI will be created in the destination Region.


☐ Copy tags
 Includes your user-defined AMI tags when copying the AMI.


☒ Encrypt EBS snapshots of AMI copy
 Encrypts all snapshots in the AMI copy with the same key.

KMS key
 This is the KMS key used to encrypt the snapshots.

▼ KMS key details

Description
 master-key-enc

Account ID
 096958155378



KMS key ID
 17c93bd2-69e3-4281-abd4-ceca163e086f


KMS key ARN

Figure 4.27. AMI Creation Details (Author)

Once the AMI was created, the Launch Template was updated to use the newly created encrypted AMI:

Launch Templates (1/3) [Info](#)

Launch Template ID	Launch Template Name	Default Version	Latest Version	Create Time	Created By
 It-0e045a76c7f5909e8	enc-test-template	1	2	2023-11-11T18:11:42.000Z	arn:aws:iam:096958155378:root
 It-03c829b1a2606d8a4	Mytemplate	1	1	2023-11-09T19:31:16.000Z	arn:aws:iam:096958155378:root

Actions  Create launch template

- Launch instance from template
- Modify template (Create new version)
- Delete template
- Delete template version
- Set default version
- Manage tags

Figure 4.28. AWS Console – Creating new Launch Template (Author)

▼ Application and OS Images (Amazon Machine Image) Info

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Specify a custom value...

masters-image-backend
ami-017a31c1cc15bcc44
2023-11-11T18:08:05.000Z

Virtualization: hvm
ENA enabled: true
Root device type: ebs

masters-image-backend
ami-049d22fe9ddb99f09
2023-10-15T17:11:42.000Z

Virtualization: hvm
ENA enabled: true
Root device type: ebs

masters-image-backend-enc3
ami-0a07869af9ac87d5d
2023-11-05T17:26:58.000Z

Virtualization: hvm
ENA enabled: true
Root device type: ebs

masters-image-backend
ami-017a31c1cc15bcc44
2023-11-11T18:08:05.000Z

Virtualization: hvm
ENA enabled: true
Root device type: ebs

Description

[Copied ami-049d22fe9ddb99f09 from us-east-1] masters-image-backend

Architecture

AMI ID

x86_64

ami-017a31c1cc15bcc44

▼ Summary

Software Image (AMI)
[Copied ami-049d22fe9ddb99f09 ...read more
ami-017a31c1cc15bcc44

Virtual server type (instance type)
-

Firewall (security group)
-

Storage (volumes)
1 volume(s) - 8 GiB

Free tier: In your first year includes 750 hours of t2.micro (or t3.micro in the Regions in which t2.micro is unavailable) instance usage on free tier AMIs per month, 30 GiB of EBS storage, 2 million I/Os, 1 GB of snapshots, and 100 GB of bandwidth to the internet.

Cancel

Create template version

Figure 4.29. AWS Console – New Launch Template details (Author)

With that in place, the auto scaling started launching unhealthy instances that would terminate as soon as they start up.

Instances (1/8) Info

<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
<input type="checkbox"/>		i-0697c8cc279078461	Terminated	t2.micro	-	No alarms	us-east-1a
<input type="checkbox"/>		i-0ce168024551c802f	Terminated	t2.micro	-	No alarms	us-east-1a
<input type="checkbox"/>		i-08fa19b5c185595ae	Terminated	t2.micro	-	No alarms	us-east-1b
<input type="checkbox"/>		i-08500cbeb694c98ac	Terminated	m1.small	-	No alarms	us-east-1d
<input type="checkbox"/>		i-0a56155b477cfa566	Terminated	t2.micro	-	No alarms	us-east-1b
<input type="checkbox"/>		i-0073fbf53dce7313a	Terminated	t2.micro	-	No alarms	us-east-1b
<input type="checkbox"/>		i-080a1ed373ca099d7	Terminated	t2.micro	-	No alarms	us-east-1a
<input checked="" type="checkbox"/>		i-0ebe38336a911adfe	Terminated	t2.micro	-	No alarms	us-east-1a

Figure 4.30. AWS Console – EC2 listing (Author)

The logs were verified so problem could be better understood, to no avail:

Activity history (100+)					
<input type="text" value="Filter activity history"/>			< 1 2 3 4 5 6 7 8 ... >		
Status	Description	Cause	Start time	End time	
⊖ Cancelled	Launching a new EC2 instance: i-0073fbf53dce7313a. Status Reason: Instance became unhealthy while waiting for instance to be in InService state. Termination Reason: Client.InternalError: Client error on launch	At 2023-11-11T18:23:20Z an instance was launched in response to an unhealthy instance needing to be replaced.	2023 November 11, 06:23:22 PM +00:00	2023 November 11, 06:23:53 PM +00:00	
⊖ Connection draining in progress	Terminating EC2 instance: i-0a56155b477cfa566 - Waiting For ELB Connection Draining.	At 2023-11-11T18:23:20Z an instance was taken out of service in response to an EC2 health check indicating it has been terminated or stopped.	2023 November 11, 06:23:20 PM +00:00		
⊖ Cancelled	Launching a new EC2 instance: i-0ce168024551c802f. Status Reason: Instance became unhealthy while waiting for instance to be in InService state. Termination Reason: Client.InternalError: Client error on launch	At 2023-11-11T18:23:00Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2023 November 11, 06:23:02 PM +00:00	2023 November 11, 06:23:33 PM +00:00	
Launching a new EC2					

Figure 4.31. AWS Console – EC2 error logs (Author)

Unfortunately, the provided information did not have enough details, apart from ‘Client error on launch’. The author suspects that the new symmetric key needed to be loaded as part of the launch template, or the auto scaling group, or maybe there were privilege issues on the IAM, or the security group and they were not able to access the new key when trying to mount the block storage as part of the auto scaling group. Instead of continuing with the investigation of the issue, instead it was decided to not progress with the independent KMS key, but rather use the one provided and managed by AWS when encrypting the AMI.

Create a copy of an Amazon Machine Image in a region

Copy Amazon Machine Image (AMI)

Original AMI ID
ami-049d22fe9ddb99f09

AMI copy name

arn:aws:kms:us-east-1:096958155378:alias/master-key-enc
arn:aws:kms:us-east-1:096958155378:alias/aws/xray
arn:aws:kms:us-east-1:096958155378:alias/aws/dynamodb
arn:aws:kms:us-east-1:096958155378:alias/aws/ebs
arn:aws:kms:us-east-1:096958155378:alias/aws/elasticfilesystem
arn:aws:kms:us-east-1:096958155378:alias/aws/es
arn:aws:kms:us-east-1:096958155378:alias/aws/glue
arn:aws:kms:us-east-1:096958155378:alias/aws/kinesisvideo
arn:aws:kms:us-east-1:096958155378:alias/aws/rds
arn:aws:kms:us-east-1:096958155378:alias/aws/redshift
arn:aws:kms:us-east-1:096958155378:alias/aws/redshifttest
arn:aws:kms:us-east-1:096958155378:alias/aws/s3
arn:aws:kms:us-east-1:096958155378:alias/aws/ssm

Select a KMS Key

Figure 4.32. AWS Console – AMI Creation with default key (Author)

That one change to the configuration allowed for the Auto Scaling group to be healthy and launching working instances again:

Instances (1/10) Info							
<input type="text" value="Find Instance by attribute or tag (case-sensitive)"/>							
<input type="checkbox"/>	Name ✎	Instance ID	Instance state ▼	Instance type ▼	Status check		
<input type="checkbox"/>		i-0697c8cc279078461	Terminated 🔍 🔍	t2.micro	–		
<input type="checkbox"/>		i-0ce168024551c802f	Terminated 🔍 🔍	t2.micro	–		
<input type="checkbox"/>		i-08fa19b5c185595ae	Terminated 🔍 🔍	t2.micro	–		
<input type="checkbox"/>		i-0c0b6cb1acc4f4e9d	Running 🔍 🔍	t2.micro	Initializing 🔍		
<input type="checkbox"/>		i-08500cbeb694c98ac	Terminated 🔍 🔍	m1.small	–		
<input type="checkbox"/>		i-0a56155b477cfa566	Terminated 🔍 🔍	t2.micro	–		
<input type="checkbox"/>		i-0073fbf53dce7313a	Terminated 🔍 🔍	t2.micro	–		
<input type="checkbox"/>		i-04d9a74c52716c890	Running 🔍 🔍	t2.micro	Initializing 🔍		
<input type="checkbox"/>		i-080a1ed373ca099d7	Terminated 🔍 🔍	t2.micro	–		
<input checked="" type="checkbox"/>		i-0ebe38336a911adfe	Terminated 🔍 🔍	t2.micro	–		

Figure 4.33. AWS Console – EC2 Successful listing (Author)

Finally, the Hibernated Warm Pool was created with the following configuration:

Create warm pool ✕

Warm pool instance state

The state that the warm pool instances will be kept in when they are idle and not in service.

Hibernated

Keep a warm pool of hibernated instances to improve scale-out speed for applications that take ...

Minimum warm pool size

The minimum number of instances that should always be in the warm pool.

1

instance

Max prepared capacity

The maximum number of instances that can be in the warm pool and Auto Scaling group at the same time.

☒ Equal to the Auto Scaling group's maximum capacity

Any updates to the group's maximum capacity will automatically apply to the warm pool's max prepared capacity.

☐ Define a set number of instances

Enter a set number of instances to use for the warm pool's max prepared capacity.

Cancel

Create

Figure 4.34. AWS Hibernated Warm Pool Window (Author)

4.7 Conclusions

In conclusion, the development process embarked upon in this research has been a critical phase in realizing the objectives set forth in our exploration of chaos engineering

70

techniques within a self-healing cloud-native microservice architecture. The systematic approach employed throughout the development lifecycle aimed at creating a robust, scalable, and resilient system capable of withstanding the rigors of chaos engineered fault injections.

By leveraging state-of-the-art technologies and methodologies tailored to the unique demands of cloud-native microservices, we successfully translated theoretical concepts into a tangible and functional solution.

The incorporation of chaos engineering techniques into the cloud-based microservice architecture played a pivotal role in assessing the system's ability to withstand unforeseen challenges. Noteworthy findings indicate that while auto-scaling warm pools, a commonly advocated approach, may not be the silver bullet on aiding the architecture's recovery from chaos engineered fault injections. This insight prompts a reevaluation of existing assumptions and highlights the need for a more nuanced understanding of resilience in the context of cloud-native microservices.

AWS Well Architected Framework played a crucial role in the decisions taken when creating this self-healing architecture, ensuring that the resulting solution aligns closely with industry best practices.

In subsequent chapters, the focus will shift towards the evaluation and analysis of the developed solution. Rigorous testing and experimentation will be employed to validate the effectiveness of the chaos engineering techniques and to measure the system's performance under diverse conditions. The insights garnered from this development process not only contribute to the academic discourse surrounding self-healing microservice architectures but also offer practical implications for industry professionals seeking to enhance the resilience of their cloud-native applications.

5 RESULTS AND EVALUATION

“Research is formalized curiosity. It is poking and prying with a purpose.”

– Zora Neale Hurston.

5.1 Introduction

The culmination of the exploration into chaos engineering techniques in a self-healing cloud-native microservice architecture brings this analysis to the pivotal stage of results, evaluation, and discussion. This section represents the synthesis of theoretical insights, practical implementations, and empirical observations, providing a comprehensive understanding of the implications and outcomes of this research endeavor.

Throughout the preceding chapters, a discussion of the intricacies of designing and implementing a resilient microservices architecture capable of withstanding chaos engineered fault injections was presented. The development process, as detailed in Chapter 4, laid the foundation for the investigation, leading to a tangible manifestation of the theoretical principles governing self-healing systems in cloud-native environments.

In this section, the outcome of the research is presented, employing a multifaceted approach that encompasses quantitative metrics, qualitative assessments, and a thorough exploration of the implications derived from the chaos engineering experiments based on the various types of warm pooling configurations. The evaluation of the self-healing microservices architecture will be underpinned by rigorous testing scenarios, allowing for a scrutiny of its responsiveness, fault tolerance, and adaptability in the face of orchestrated disruptions.

As the results are presented, the broader implications of the findings will be discussed in the context of contemporary cloud-native application development. The discussion will go beyond the immediate scope of the experiments, weaving in theoretical perspectives, industry best practices, and the evolving landscape of cloud technologies.

Moreover, this section serves as a platform for critical reflection, offering insights into the limitations of the approach used in this research, potential areas for further research, and the applicability of the findings in real-world scenarios. Through a balanced and

comprehensive analysis, the aim is to contribute not only to the academic discourse on chaos engineering and self-healing architectures but also to provide practical guidance for professionals seeking to enhance the resilience and reliability of their cloud-native microservices.

This chapter will evaluate the results of this exploration, present a thorough evaluation of the developed solution, and engage in a nuanced discussion that contextualizes the findings within the broader spectrum of cloud-native application development and chaos engineering practices.

5.2 Calibration - Results and Evaluation

Throughout the experiment, the utilized broadband speeds were verified by testing it eight times to ensure it was consistent and would not interfere with the results.

No.	Download (MB)	Upload (MB)	Ping Latency (secs)	Download Latency (secs)	Upload Latency (secs)
1	405	50	7	51	235
2	450	51	9	48	272
3	448	50	8	46	62
4	428	50	8	38	60
5	410	51	9	46	128
6	424	51	8	39	90
7	423	48	8	56	152
8	455	51	7	83	246
Avg.	430	50	8	51	156

Table 5.1. Broadband Details (Author)

The results above were recorded and the full details of these results can be found at:

<https://github.com/brunorfranco/masterThesis/tree/main/experiments-results/BroadbandSpeed>

Timing each experiment step for the four different warm pooling variations was also consistent, where a timer was utilized to initialize the JMeter test plan, and then the FIS experiment with 30 seconds apart from each other. It was noted that across all the 32 executions of the experiment, it took an average of 13.68 seconds for the Fault Injector to disable the application from responding to the JMeter requests. In the tables below “TtU” is “Time to Unresponsiveness” in seconds.

Experiments 1-16:

	14	14	15	9	15	9	16	16	17	14	14	15	15	9	16	14

Experiments 17-32:

	14	10	13	14	15	9	16	12	15	15	14	15	15	9	15	15

Average Time to Unresponsiveness:

Over 32 experiments: 13.68 seconds

Table 5.2. Time to Unresponsiveness metrics (Author)

The numbers above were extracted from the experiments executions, by subtracting thirty seconds from the last successful row's 'Sample #' column in the JMeter View Results in Table after the Fault Injection.

The results above were recorded and can be found at: [masterThesis/experiments-results-screens at main · brunorfranco/masterThesis · GitHub](#)

The next four upcoming sections will present the results of the executed experiments per warm pooling configuration.

5.3 No Warm pooling Experiment

During the execution of the experiment while the Auto Scaling group had no warm pooling setup, the results were as follows:

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	16:29:28.433	16:32:07.260	02:38.827
2	16:37:21.088	16:39:59.916	02:38.828
3	16:44:46.607	16:46:14.237	01:27.630
4	16:51:41.303	16:54:01.630	02:20.327
5	16:58:59.813	17:00:30.580	01:30.767
6	17:05:48.506	17:08:27.393	02:38.887
7	17:13:14.345	17:16:11.756	02:57.411
8	17:20:24.474	17:22:16.195	01:51.721
Average			02:15.549

Table 5.3. No Warm Pooling Details (Author)

Below is a bar graph to facilitate the visualization of the results:

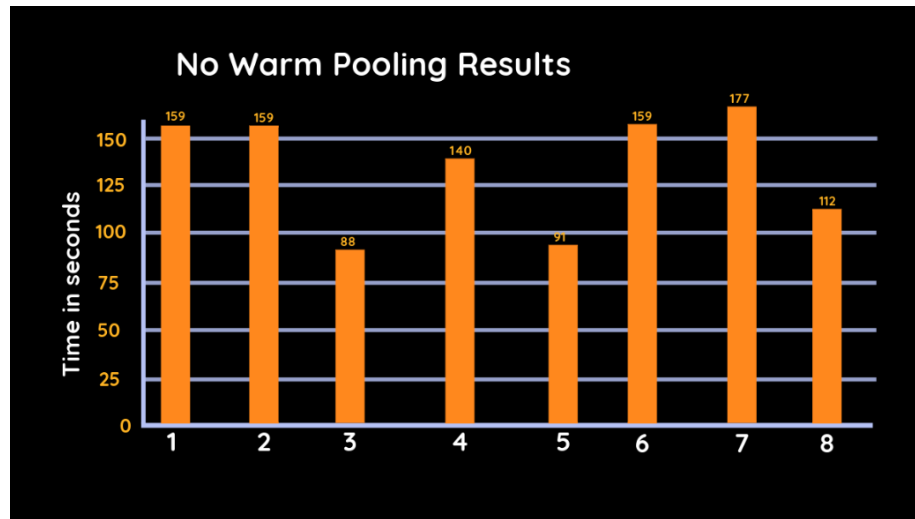


Figure 5.1. No Warm Pooling Results Graph (Author)

The Delta between the slowest and fastest experiments was 89 seconds. It is worth noting that after the first healthy instance started serving successful responses back to JMeter, AWS took an extra minute to spin up a second instance (this is an expected mechanism implemented by AWS to avoid creating unnecessary instances). It was noteworthy that during the addition of the second instance to the Elastic Load Balancer, some of the requests returned failures:

View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file:

Filename: ☐ Errors ☐ Successes

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
222	17:17:29.462	Thread Group 1-1	MastersTestAPI	106	Success	651	162	106	0
223	17:17:30.571	Thread Group 1-1	MastersTestAPI	103	Success	651	162	103	0
224	17:17:31.679	Thread Group 1-1	MastersTestAPI	102	Success	651	162	102	0
225	17:17:32.789	Thread Group 1-1	MastersTestAPI	101	Success	651	162	101	0
226	17:17:33.896	Thread Group 1-1	MastersTestAPI	99	Success	651	162	99	0
227	17:17:35.003	Thread Group 1-1	MastersTestAPI	102	Success	651	162	102	0
228	17:17:36.112	Thread Group 1-1	MastersTestAPI	101	Success	651	162	101	0
229	17:17:37.219	Thread Group 1-1	MastersTestAPI	103	Success	651	162	103	0
230	17:17:38.324	Thread Group 1-1	MastersTestAPI	103	Success	651	162	103	0
231	17:17:39.432	Thread Group 1-1	MastersTestAPI	105	Success	651	162	105	0
232	17:17:40.542	Thread Group 1-1	MastersTestAPI	99	Success	651	162	99	0
233	17:17:41.647	Thread Group 1-1	MastersTestAPI	102	Success	651	162	102	0
234	17:17:42.753	Thread Group 1-1	MastersTestAPI	104	Success	651	162	104	0
235	17:17:43.866	Thread Group 1-1	MastersTestAPI	102	Success	651	162	102	0
236	17:17:44.973	Thread Group 1-1	MastersTestAPI	102	Success	651	162	102	0
237	17:17:46.078	Thread Group 1-1	MastersTestAPI	101	Success	651	162	101	0
238	17:17:47.186	Thread Group 1-1	MastersTestAPI	99	Success	651	162	99	0
239	17:17:48.296	Thread Group 1-1	MastersTestAPI	104	Success	651	162	104	0
240	17:17:49.408	Thread Group 1-1	MastersTestAPI	119	Success	651	162	119	0
241	17:17:50.533	Thread Group 1-1	MastersTestAPI	99	Success	651	162	99	0
242	17:17:51.639	Thread Group 1-1	MastersTestAPI	102	Success	651	162	102	0
243	17:17:52.743	Thread Group 1-1	MastersTestAPI	99	Success	651	162	99	0
244	17:17:53.852	Thread Group 1-1	MastersTestAPI	99	Success	651	162	99	0
245	17:17:54.952	Thread Group 1-1	MastersTestAPI	101	Success	651	162	101	0
246	17:17:56.068	Thread Group 1-1	MastersTestAPI	103	Success	651	162	103	0
247	17:17:57.174	Thread Group 1-1	MastersTestAPI	101	Success	651	162	101	0
248	17:17:58.281	Thread Group 1-1	MastersTestAPI	90	Failure	277	162	90	0
249	17:17:59.384	Thread Group 1-1	MastersTestAPI	99	Failure	651	162	99	0
250	17:18:00.486	Thread Group 1-1	MastersTestAPI	90	Failure	277	162	90	0
251	17:18:01.579	Thread Group 1-1	MastersTestAPI	102	Failure	651	162	102	0
252	17:18:02.681	Thread Group 1-1	MastersTestAPI	91	Failure	277	162	91	0
253	17:18:03.774	Thread Group 1-1	MastersTestAPI	104	Failure	651	162	104	0
254	17:18:04.893	Thread Group 1-1	MastersTestAPI	90	Failure	277	162	90	0
255	17:18:05.995	Thread Group 1-1	MastersTestAPI	101	Failure	651	162	101	0
256	17:18:07.110	Thread Group 1-1	MastersTestAPI	91	Failure	277	162	91	0

☒ Scroll automatically! ☐ Child samples? No of Samples: 256 Latest Sample: 91 Requests: 256 Failures: 124

Figure 5.2. JMeter Intermittent Failures (Author)

This behavior would only present itself for a short time (~10 seconds), and the second instance would soon serve successful responses. Given that the detailed algorithm for adding instances into AWS Elastic Load Balancers is not publicly available, it has been theorized that the second instance would pass the health check validation once the computing instance is available, but the microservice application is not yet up and running. Based on this theory, the first instance added to the Load Balancer is also susceptible to this behavior, but unnoticeable, given that all requests were failing up to the point where the first microservice application is up and running.

5.4 Stopped Warm Pooling Experiment

During the execution of the experiment while the Auto Scaling group had a stopped warm pooling setup, the results were as follows:

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	12:21:14.040	12:23:53.779	02:39.739
2	14:18:39.652	14:20:16.130	01:36.47
3	14:27:41.048	14:30:02.658	02:21.610
4	14:36:24.950	14:38:00.258	01:35.308
5	14:44:04.283	14:46:00.484	01:56.201
6	14:51:21.225	14:54:21.301	03:00.076
7	15:01:16.763	15:04:03.782	02:47.019
8	15:09:03.961	15:12:19.420	03:15.459
Average			02:23.986

Table 5.4. Stopped Warm Pooling Details (Author)

Below is a bar graph to facilitate the visualization of the results:

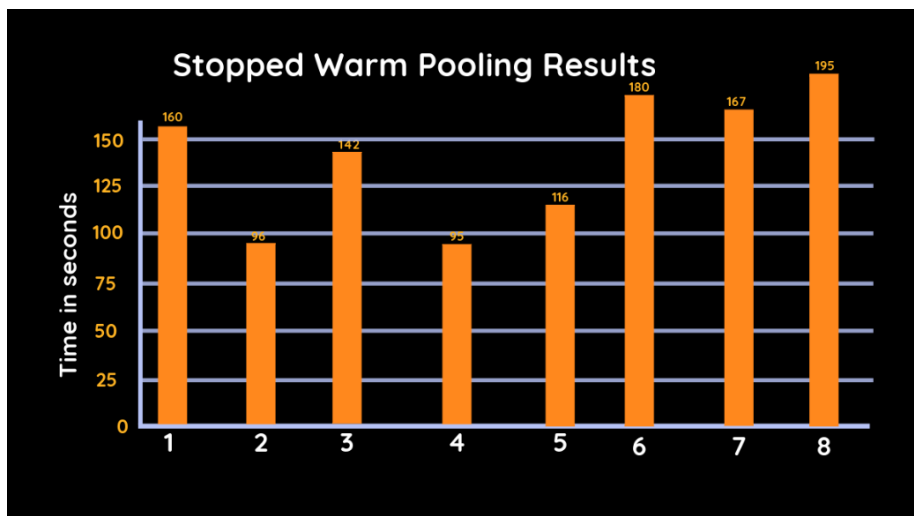


Figure 5.3. Stopped Warm Pooling Results Graph (Author)

The Delta between the slowest and fastest experiments was 100 seconds. Regarding the issue discussed in section 5.3 where the second computing instance added to the load balancer fails for around 8 seconds, this behavior can also be noticed when using stopped warm pools.

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
236	15:13:37.237	Thread Group 1-1	MastersTestAPI	104	Success	651	182	104	0
237	15:13:38.343	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0
238	15:13:39.464	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0
239	15:13:40.585	Thread Group 1-1	MastersTestAPI	102	Success	651	182	102	0
240	15:13:41.694	Thread Group 1-1	MastersTestAPI	105	Success	651	182	105	0
241	15:13:42.803	Thread Group 1-1	MastersTestAPI	105	Success	651	182	102	0
242	15:13:43.924	Thread Group 1-1	MastersTestAPI	105	Success	651	182	105	0
243	15:13:45.038	Thread Group 1-1	MastersTestAPI	108	Success	651	182	108	0
244	15:13:46.153	Thread Group 1-1	MastersTestAPI	106	Success	651	182	106	0
245	15:13:47.262	Thread Group 1-1	MastersTestAPI	100	Success	651	182	100	0
246	15:13:48.367	Thread Group 1-1	MastersTestAPI	105	Success	651	182	105	0
247	15:13:49.474	Thread Group 1-1	MastersTestAPI	103	Success	651	182	103	0
248	15:13:50.580	Thread Group 1-1	MastersTestAPI	112	Success	651	182	112	0
249	15:13:51.694	Thread Group 1-1	MastersTestAPI	94	Failed	277	182	94	0
250	15:13:52.801	Thread Group 1-1	MastersTestAPI	107	Failed	651	182	107	0
251	15:13:53.921	Thread Group 1-1	MastersTestAPI	91	Failed	277	182	91	0
252	15:13:55.013	Thread Group 1-1	MastersTestAPI	107	Failed	651	182	107	0
253	15:13:56.135	Thread Group 1-1	MastersTestAPI	90	Failed	277	182	90	0
254	15:13:57.241	Thread Group 1-1	MastersTestAPI	104	Failed	651	182	104	0
255	15:13:58.349	Thread Group 1-1	MastersTestAPI	91	Failed	277	182	91	0
256	15:13:59.454	Thread Group 1-1	MastersTestAPI	108	Failed	651	182	108	0
257	15:14:00.563	Thread Group 1-1	MastersTestAPI	92	Failed	277	182	92	0
258	15:14:01.669	Thread Group 1-1	MastersTestAPI	111	Failed	651	182	111	0
259	15:14:02.787	Thread Group 1-1	MastersTestAPI	92	Failed	277	182	92	0
260	15:14:03.885	Thread Group 1-1	MastersTestAPI	123	Failed	651	182	123	0
261	15:14:05.009	Thread Group 1-1	MastersTestAPI	91	Failed	277	182	91	0
262	15:14:06.101	Thread Group 1-1	MastersTestAPI	116	Failed	651	182	116	0
263	15:14:07.226	Thread Group 1-1	MastersTestAPI	92	Failed	277	182	92	0
264	15:14:08.333	Thread Group 1-1	MastersTestAPI	218	Failed	651	182	218	106
265	15:14:09.552	Thread Group 1-1	MastersTestAPI	91	Failed	277	182	91	0
266	15:14:10.645	Thread Group 1-1	MastersTestAPI	116	Failed	651	182	116	0
267	15:14:11.768	Thread Group 1-1	MastersTestAPI	811	Failed	651	182	808	0
268	15:14:13.594	Thread Group 1-1	MastersTestAPI	110	Success	651	182	110	0
269	15:14:14.717	Thread Group 1-1	MastersTestAPI	106	Success	651	182	106	0
270	15:14:15.352	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0

Figure 5.4. Stopped Warm Pooling JMeter failures (Author)

5.5 Hibernated Warm Pooling Experiment

During the execution of the experiment while the Auto Scaling group had a hibernated warm pooling setup, the results were as follows:

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	18:08:25.344	18:09:48.567	01:23.223
2	18:15:08.020	18:17:42.913	02:34.893
3	18:22:12.935	18:23:47.238	01:34.303
4	18:29:12.944	18:31:43.360	02:30.416
5	18:39:44.713	18:41:51.029	02:06.316
6	18:46:36.257	18:48:14.913	01:38.656
7	18:54:54.811	18:55:57.026	01:02.215
8	19:01:58.829	19:03:51.875	01:53.046
Average			01:50.383

Table 5.5. Hibernated Warm Pooling Details (Author)

Below is a bar graph to facilitate the visualization of the results:

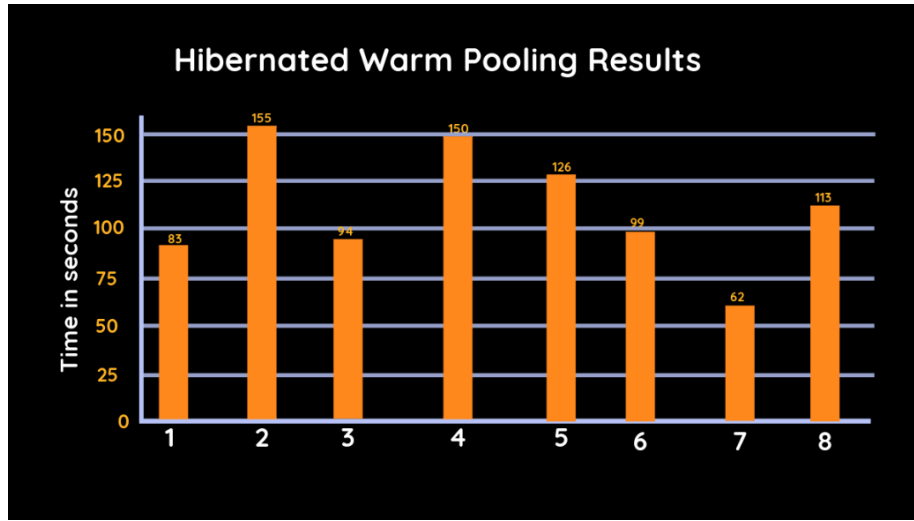


Figure 5.5. Hibernated Warm Pooling Results Graph (Author)

The Delta between the slowest and fastest experiments was 93 seconds. Hibernated warm pools would not present failed responses when adding instances into the load balancer. This is likely due to the nature of hibernated instances. When the instance is started again, the root volume is restored to its previous state and the RAM contents are reloaded, therefore the application will be up and running as soon as the instance is pulled from the pool.

5.6 Running Warm Pooling Experiment

During the execution of the experiment while the Auto Scaling group had a running warm pooling setup, the results were as follows:

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	17:37:44.065	17:39:35.936	01:51.871
2	17:45:18.922	17:47:34.990	02:16.068
3	17:52:46.518	17:53:48.744	01:02.226
4	18:02:41.585	18:03:43.793	01:02.208
5	18:10:01.389	18:11:29.012	01:27.62
6	18:16:43.552	18:17:45.765	01:02.213
7	18:30:05.661	18:31:34.408	01:28.747
8	18:37:05.916	18:38:08.136	01:02.220
Average			01:24.147

Table 5.6. Running Warm Pooling Details (Author)

Below is a bar graph to facilitate the visualization of the results:

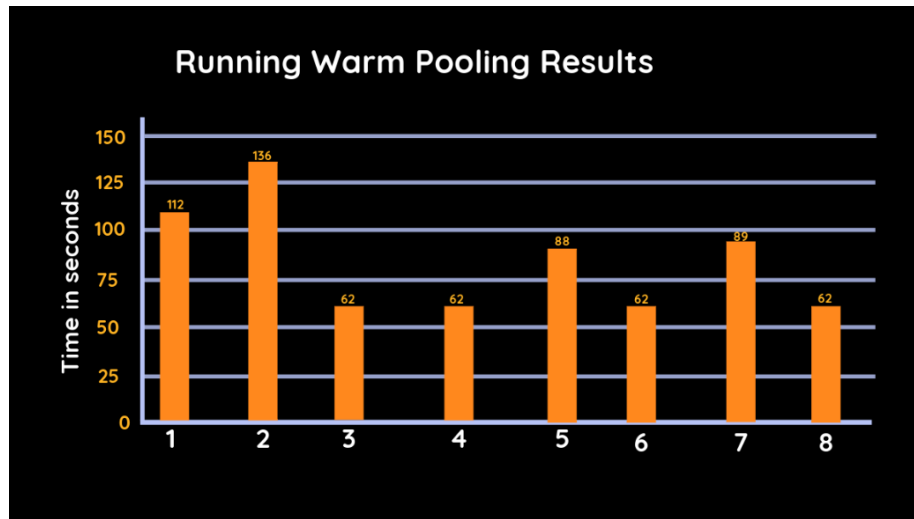


Figure 5.6. Running Warm Pooling Results Graph (Author)

The Delta between the slowest and fastest experiments was 74 seconds. As expected, running warm pools would also not present failed responses when adding instances into the load balancer, given that the instance is already running, therefore the application is already up.

During the execution of this part of the experiment, one outlier was observed (screenshots can be found at [masterThesis/experiments-results-screensies/RunningWarmPool/7-outlier at main · brunorfranco/masterThesis · GitHub](https://github.com/brunorfranco/masterThesis/tree/master/experiments-results-screensies/RunningWarmPool/7-outlier)).

Even though the overall time to recover was 1 minute and 35 seconds (slightly above the average but still within range of acceptance), an unusual behavior was observed after the 3 minutes mark into the experiment. The first successful request after the fault injection was followed by a failure that held the thread for an average of 10 seconds. This ‘success-into-10-seconds-failure’ pattern repeated itself three times in total, and after that, all requests started succeeding again:

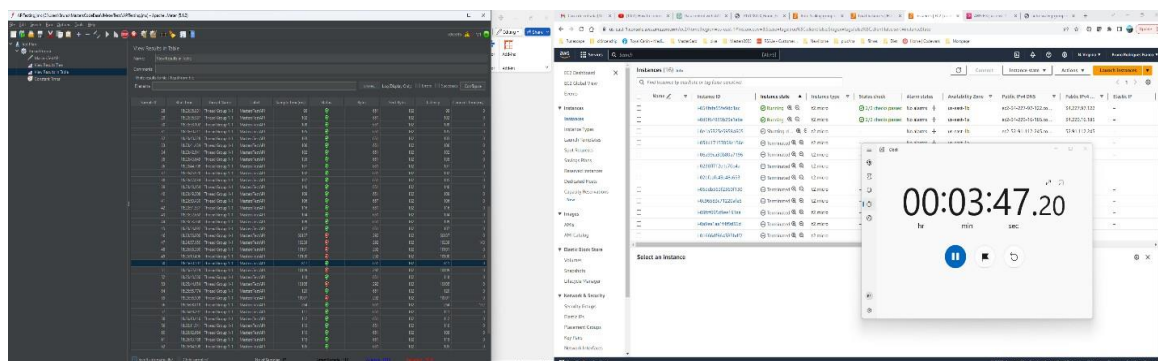


Figure 5.7. JMeter Running Warm Pool Outlier (Author)

It appears that the issue occurred when adding the second instance into the load balancer (as observed for the experiments with no warm pooling and stopped warm pooling). However, this should not have happened for running warm pooling, given that the instances are supposed to be already running when pulling from the warm pool, and what makes it more interesting is that this is the first and only time in all the 32 executions of the experiment that the response took ten seconds before returning with a failure.

The explanation for such an event can be vague, given the error logs are limited at best. It could have been an issue with the network connection for that specific EC2 instance, or possibly the load balancer recruited the instance at the same time that the instance was commissioned and prepared, and was sent to the warm pool itself causing a recruiting conflict between the warm pool and the load balancer, or it could simply be that the instance misbehaved due to unexpected issues in hardware/software but was healed in around 30 seconds.

5.7 Conclusions

This chapter presented the outcomes of the research, using a multifaceted approach that encompasses quantitative metrics, qualitative assessments, and a thorough exploration of the implications derived from the chaos engineering experiments based on the various types of warm pooling configurations. The evaluation of the self-healing microservices architecture is underpinned by rigorous testing scenarios, allowing for a scrutiny of its responsiveness, fault tolerance, and adaptability in the face of orchestrated disruptions. All the data presented in chapter 5 can be verified from the screenshots presented at: <https://github.com/brunorfranco/masterThesis/tree/main/experiments-results-screensies>.

Four separate experiments were undertaken (eight times each), and the average time for each type of experiment was:

#	Experiment Type	Average Time
1	No warm pooling recovery time	02:15.549
2	Stopped warm pooling recovery time	02:23.986
3	Hibernated warm pooling recovery time	01:50.383
4	Running warm pooling recovery time	01:24.147

Given that the average metrics were collected in the ‘mm:ss.mmm’ format, they were converted to seconds to facilitate comparisons.

No.	No Pool	Stopped	Hibernated	Running
1	159	160	83	112
2	159	96	155	136
3	88	142	94	62
4	140	95	150	62
5	91	116	126	88
6	159	180	99	62
7	177	167	62	89
8	112	195	113	62
Avg.	136	144	110	84

Table 5.7. Result Conversion to Seconds (Author)

The following vertical bar graph compares the recovery time in seconds per warm polling configuration:

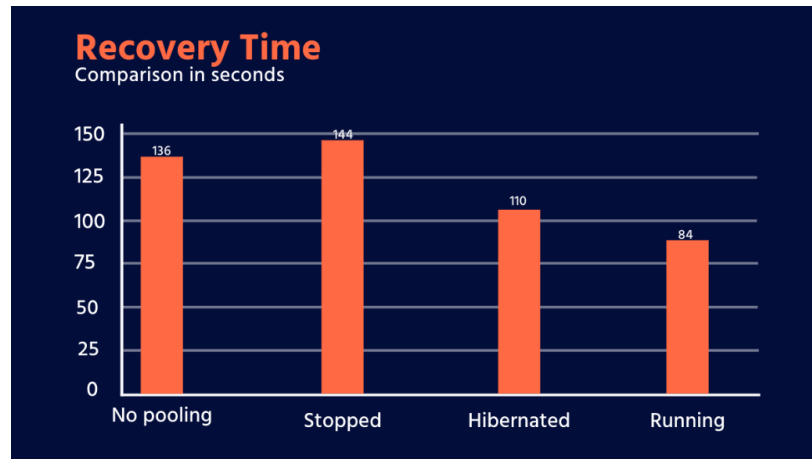


Figure 5.8. Recovery Time Comparison Graph (Author)

When visualizing the data through a pie chart, this is the outcome:

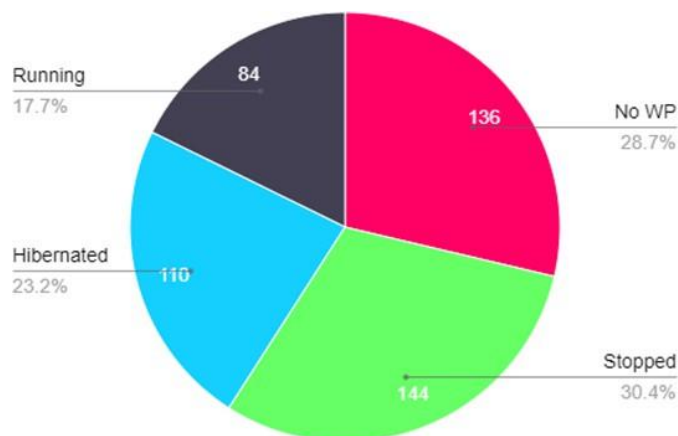


Figure 5.9. Recovery Time Comparison Chart (Author)

The following table presents the warm pooling recovery time comparison between one another:

	<i>No WP</i>	<i>Stopped</i>	<i>Hibernated</i>	<i>Running</i>
<i>No WP</i>		+5.88%	-23.64%	-61.9%
<i>Stopped</i>			-30.91%	-71.43%
<i>Hibernated</i>				-30.95%
<i>Running</i>				

Table 5.8. Warm Pooling Performance Comparison (Author)

Unexpectedly, the results show that having no warm pools was 5.88% faster than setting up Stopped Warm pools, contrary to popular belief. This could have been caused by the effects of the overhead created in the Auto Scaling group when the computing instances are small sized machines (t2.micro – one of the smallest machine offered by AWS at the time of this writing) and the start-up time of the application is also minimal (averaging 3.47 seconds).

The remaining results are in accordance with what was expected by this paper and the AWS documentation¹², where running warm pools should outperform all of the other's options, and hibernated warm pools should outperform stopped warm pools and having no warm pools.

As mentioned in section 4.6, it is important to note that only the hibernated and running warm pooling configurations did not present failed responses when adding the second instance into the Load Balancer. That can be explained by the fact that hibernated instances pre-initialize the entire EC2 instance state, not just the disk state, therefore when they are requested from the pool, they already have the Java application in running state, and running warm pool, as the name suggests, already has the instances fully running, so they are ready to serve incoming requests. That concludes that when prioritizing availability over performance, it is best to use hibernated or running warm pools to avoid intermittent failed responses when requesting instances from the warm pool.

¹² <https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-warm-pools.html>

The delta between fastest and slowest recovery times for the four experiments also conclude that Stopped Warm Pooling is the most irregular of the four, with 100 seconds of delta, while Running Warm Pooling only has 74 seconds, therefore it has a more stable behavior.

When comparing the experiment findings with Frincu, *et al.* (2011), the AWS architecture self-healed slower than Frincu's multi agent system for inter-provider task scheduling enhanced with self-healing capabilities.

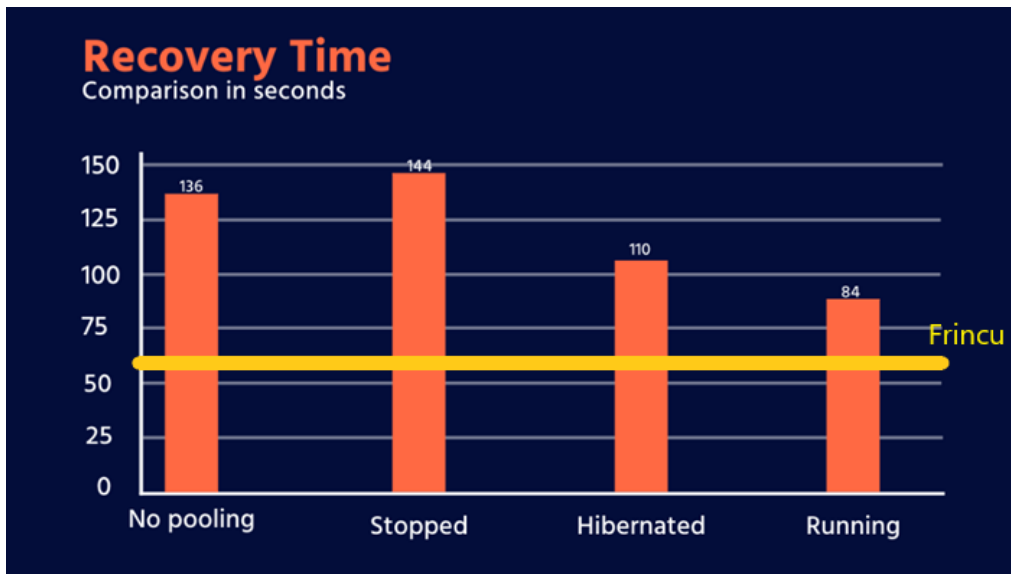


Figure 5.10. AWS Recovery Time compared to Frincu and friends (Author)

It is worth mentioning that each pooling configuration has different financial costs, and requires different levels of expertise to configure:

Configuration Type	Financial Costs	Configuration Complexity
No WP	None	None
Stopped	Low – EBS Volume	Low (simple UI wizard)
Hibernated	Medium – EBS Volume (including RAM)	High (requires encrypted AMI)
Running	High – EBS and compute time	Low (simple UI wizard)

Table 5.9. Warm Pooling Cost/Complexity Comparison (Author)

Given the nature of each warm pooling option, some conclusions can be drawn. Warm pooling in general is known to help decrease latency for application with long boot times, based on the AWS Warm Pooling documentation. Given that the current experiment had an application with an average time of 3.47 seconds, a separate experiment with a longer boot time application will likely yield different results.

It is also important to note that the running warm pool option is not financially advised. If one is already willing to pay for a compute instance and its corresponding block storage, it would be best to simply add such instance to the load balancer and use it, rather than keep it in warm pool running and incurring costs.

When looking back into the research question, the results presented in this paper confirm that AWS could not self-heal its microservice architecture within the defined timeframe of sixty seconds. By utilizing auto scaling group with running warm pools, the AWS platform came close to achieving it, but it missed the mark by extra 24 seconds, totaling 84 seconds to get back to a healthy state.

5.8 *Expert Interviews Conclusions*

Three interviews were undertaken with industry experts specializing in Cloud-based Software Engineering. The forthcoming subsections will comprehensively outline and expound upon the feedback provided by these experts.

5.8.1. First Interview

I1 has suggested for the experiment to be conducted again with the Elastic Load Balancer reconfigured with a shorter time-out (which is defaulted to 60 seconds). They concluded that the default time-out could cause performance issues by overloading instances with throttled requests while the Auto Scaling group works to spin up more instances.

I1 has wondered how AWS would cope with a scenario where the deployed application holds the requests for 5 seconds but JMeter continues to fire requests every second, so that there will always be a growing queue of unattended requests.

I1 agreed with the paper when concluding that if the experiment had used larger compute

instances, the warm pooling results would have been vastly different, likely much better than no warm pooling.

I1 has also stated that in many industries segments, the 2 minutes, and 16 seconds necessary for AWS to self-heal itself without any warm pooling is acceptable and the return in investment for setting up and maintaining warm pooling might not be justified. Overall, I3 was satisfied with how the experiment was conducted and the tools/platforms chosen.

5.8.2. Second Interview

I2 have challenged the paper's choice of having 60 seconds SLO as part of the Research Question, given that Frincu, *et al.* (2011) experiment has many different aspects from this paper (i.e., optimal number of agents and number of idle clones).

They also suggested that instead of utilizing the FIS tool, a different approach where the Java application itself would cause the error and bring the instance down with it.

During the experiment, when the Auto Scaling group is spinning up the first and second instances after the fault injection, I2 has queried what is the average time between the instantiation of the first and second instances, which unfortunately was a metric that was not collected during the experiment execution.

I2 suggested for the experiment to be executed more times with variations on the start-up time of the Java application, to correlate the start-up time with the warm pooling results and infer if the relation between the two is linear or exponential.

I2 agreed that the findings of the experiment where no warm pooling outperformed stopped warm pooling are interesting and valuable. They have stated that it is likely that many companies have implemented Stopped Warm Pooling without knowing that this is worsening their resilience rather than helping them.

I2 has approved the design the of experiment and the chosen tools and platforms, however, they have said they would ensure JMeter is accurately recording the findings, by having a second tool (for example Postman or Karate API testing) to compare it with.

I2 was satisfied with how the metrics were calculated, but they have stated that given the human interactions when clicking buttons to start JMeter and the FIS tool, as well as the JMeter one second delay per request, there will be a margin of error in the results.

The interviewee would have liked to have the experiment executed with more variables, for example more instances in the auto scaling group, or more instances in the warm pool, to understand better if the resilience would grow linearly or exponentially with more instances.

They have also suggested that the paper could present the delta between the slowest and fastest recovery times for each warm pooling configuration. This suggestion was heard and implemented.

I2 has concluded that they would recommend the use of Hibernated warm pools for most cases, given that it has substantial performance gains over no warm pools, and it is budget friendly (cheaper than running warm pool).

Overall, I2 was satisfied with how the experiment was conducted and the tools/platforms chosen.

5.8.3. Third Interview

I3 has reviewed the experiment design and suggested that more than one AWS region should be used to achieve higher resilience.

I3 expected AWS to recover in less time than the experiment has shown. They have also agreed that if the Auto scaling group had more frequent health checks, the recovery time would decrease significantly.

I3 also agrees that if the application start-up time was longer or the compute instances were larger, warm pools would have performed better than no warm pools.

Overall, I3 was satisfied with how the experiment was conducted and the tools/platforms chosen.

6 CONCLUSIONS AND FUTURE WORK

“If we knew what we were doing, it would not be called research, would it?”

– Albert Einstein.

6.1 Introduction

Within the ever-evolving landscape of cloud-native microservice architectures, this MSc thesis has embarked on a pioneering expedition, delving deep into the realm of chaos engineering techniques and their application in fostering resilience within self-healing systems. As this comprehensive exploration nears its denouement, this conclusion serves as a compass, directing attention towards the cardinal discoveries, implications, and future trajectories illuminated throughout this odyssey.

Throughout the preceding chapters, a meticulous examination of chaos engineering techniques has unfolded, strategically applied to the intricate web of a self-healing cloud-native microservice architecture. This pursuit has not only unraveled the intricate interplay between chaos and resilience but has also underscored the significance of proactive measures in fortifying systems against unforeseen adversities.

In the crucible of this investigation, the synthesis of empirical data, the evaluation of resilience patterns, and the validation of chaos engineering methodologies have all converged to unveil a tapestry of insights. Moreover, this conclusion serves as the nexus where the synergistic amalgamation of theoretical frameworks and practical applications within the realm of self-healing architectures is encapsulated.

As the thesis approaches its crescendo, the far-reaching implications of this exploration within the broader context of cloud-native ecosystems are elucidated. The ramifications extend beyond theoretical frameworks, transcending into the realm of real-world implementation, where the principles elucidated herein hold the potential to redefine practices, guide strategic decision-making, and reshape the paradigms of system reliability and robustness.

This conclusion, however, does not signify an endpoint but rather a pivotal juncture. It

beckons a contemplation of the expedition thus far—a testament to the rigor, innovation, and resilience inherent in the pursuit of advancing technological frontiers. Furthermore, it propels the discourse forward, inviting continued investigation, refinement, and application of chaos engineering techniques within the ever-evolving tapestry of cloud-native microservice architectures.

6.2 Conclusions

6.2.1. Experiment Design

The experimental design conducted in this exploration of Chaos Engineering techniques within a self-healing Cloud Native Microservice Architecture has yielded profound insights into the resilience and adaptability of complex systems.

Through a meticulously structured methodology, deliberate faults were injected into the system to understand its response under stress. This experimentation revealed invaluable information about the system's behavior, vulnerabilities, and the efficacy of its self-healing mechanisms. The findings not only validated the importance of Chaos Engineering but also highlighted its pivotal role in fortifying system robustness against unforeseen disruptions.

The fault injections provided unique perspectives on system resilience, aiding in the identification of weaknesses and the enhancement of recovery strategies. This granular understanding forms a cornerstone for refining existing practices and forging innovative approaches to bolster Cloud Native Microservice Architectures.

Moreover, the experiments underscored the dynamic nature of system resilience. The adaptive nature of self-healing mechanisms was observed, showcasing their ability to learn from and adapt to disruptions, further strengthening the architecture's overall robustness.

The insights gleaned from this experiment design offer a roadmap for future endeavors in Chaos Engineering. They emphasize the need for continual exploration and evolution, advocating for a proactive stance towards system reliability within modern architectures. As technology progresses, the lessons learned here provide a solid foundation for

advancing the field and ensuring the resilience and dependability of Cloud Native Microservice Architectures in an ever-changing landscape.

In conclusion, the experiment design undertaken in this exploration has not only validated the importance of Chaos Engineering but has also illuminated its potential in fortifying the resilience of Cloud Native Microservice Architectures. The findings serve as a catalyst for future research, shaping a more robust and adaptive technological landscape through the application of proactive Chaos Engineering techniques in Self-healing architectures.

6.2.2. Tools and Platforms

This paper discussed in detail the options for conducting the experiment, from the cloud provider to the application, the architectural design, the API testing tool, and the fault injection tool. The chosen tools and platforms were proven to be fit for purpose, delivering on their promises, and facilitating the realization of the experiment.

The key findings for chapter 3 were:

- The JMeter API testing tool proved itself to be highly configurable, facilitating the setup of HTTP requests waiting times, delay per request, number of active threads, and multiple options for visualizing collected data.
- The AWS FIS service offers a variety of options to inject faults into multiple AWS services. For this experiment, only the EC2 termination action was used, but other options could have been leveraged.
- The choice of AWS as the cloud provider allowed for low level configuration on compute instances, auto scaling groups, warm pools, and load balancers.

6.2.3. Experiment Execution

The choices made for the experiment execution had either strong reasonings or were inspired by other papers in the field. The number of executions (inspired by Ali Naqvi, *et al.* (2022)) summed up to 32 in total, which gives high confidence in the results. They were conducted in a controlled environment, started each time from a stable state, timed accordingly and recorded in details ([masterThesis/experiments-results-screens at main](#)

[. brunorfranco/masterThesis · GitHub](#)).

The key findings were:

- The timing system applied to the experiment helped infer more data than initially envisioned (i.e., time to unresponsiveness from the start of the fault injection)
- Broadband variation was negligible, given that the fault injection occurred within the AWS network.
- The choice to terminate instances via instance ID in AWS FIS proved to be time consuming and manual, given that the FIS template needed to be updated with new IDs for each experiment execution. The author regrets not having used ‘tags’.
- The number of executions was satisfactory for high confidence in the results.

6.2.4. Results Evaluation

The results presented in this paper confirm that AWS could not self-heal its microservice architecture within the defined timeframe of sixty seconds. Some specific configurations in the auto scaling groups performed better than others, but none reached the pre-defined time window SLO.

AWS presented its best results by utilizing running warm pools, missing the SLO mark by 24 seconds, totaling 84 seconds to self-heal.

Also, the results show that having no warm pools was 5.88% faster than setting up Stopped Warm pools, contrary to expectations.

It is worth noting that for this experiment, JMeter was configured to submit one request per second, therefore there is a small margin of error for each experiment (i.e., AWS could had healed itself just after a request went off and waited close to 1000 milliseconds for the subsequent request to arrive and return successfully).

When evaluating the delta between fastest and slowest recovery times for the four experiments, it can be concluded that Stopped Warm Pooling is the most unpredictable of the four, with 100 seconds of delta, while Running Warm Pooling is the most predictable, with a variation of 74 seconds.

As mentioned in section 4.6, it is important to note that only the hibernated and running warm pooling configurations did not present failed responses when adding the second instance into the Load Balancer. Therefore, when prioritizing availability, it is best to use hibernated or running warm pools to avoid intermittent failed responses.

It is worth reiterating that when comparing the experiment findings with Frincu, *et al.* (2011), the AWS's best results self-healed 40% slower than Frincu's multi agent system for inter-provider task scheduling enhanced with self-healing capabilities, which averaged 60 seconds.

Warm pooling in general is known to help decrease latency for application with long start-up times. Given that the current experiment had an application with an average start-up time of 3.47 seconds, a separate experiment with a longer start-up time application will likely yield different results.

It is also important to note that it is not financially advised to configure running warm pools. If one is already willing to pay for a compute instance and its corresponding block storage, it would be best to simply add such instance to the load balancer and use it, rather than keep it in warm pool running and incurring costs.

6.2.4. Experts Interviews Summary

The validity of the experiment and the accuracy of the selected tools and platforms were affirmed by the three interviewees. However, I2 highlighted certain inherent fluctuations in the collected metrics, indicating a variance of a few seconds. This was attributed to JMeter's triggering of requests every second, potential request throttling due to timeouts, and observed fluctuations in broadband performance on different days during the experiments.

All interviewees expressed a desire for the experiment to incorporate more variations, enabling a broader range of conclusions. For instance, I1 proposed varying the Load Balancer timeout configuration, I2 suggested altering the cluster size and warm pool dimensions, while I3 recommended varying the number of AWS regions. Regrettably, implementing these suggestions would significantly expand the paper's scope and require substantial time and effort to set up the proposed experiments.

A consensus was reached among the interviewees regarding a bottleneck in AWS's time to recover, which is the time window in which the Auto Scaling group runs health-checks against instances. Both I1 and I3 acknowledged the unalterable nature of this configuration at the time of writing. However, I2 expressed uncertainty about the feasibility of changing the health-check time window and opted to refrain from providing a definitive response.

Despite the wealth of suggestions and insights offered during the interviews, the experts expressed satisfaction with the experiment's execution and the credibility of the obtained results.

6.3 Contributions and Impact

The research project incorporates contributions that augment the existing body of work, encompassing:

- The research explored the various options provided by the AWS platform when it comes to auto scaling groups warm pooling configuration.
- The research compared how warm pooling options compare with one another when under stress.
- The research investigated various tools and technologies to implement a successful chaos engineered experiment against a cloud-based architecture.
- The study highlights both the advantages and limitations of utilizing auto scaling groups and elastic load balancers.
- The research described in detail how to create a highly available architecture in AWS by following the Well Architected Framework.
- The research examined how resilient and available the AWS platform can be

when under chaos engineered fault injections.

6.4 Future Work

Upon concluding this research project, numerous prospects for future research emerged.

6.4.1 Long Start-up time applications

Further work on executing a similar experiment, with an application that requires longer start-up time than the 3.47 seconds used in this research, could be carried out.

Based on the AWS documentation, warm pools would be more beneficial in such scenarios. Therefore, the results should vary substantially from the results in this paper. If a similar Java/Spring application were used in this scenario, then the longer start-up time could be accomplished by adding a `Thread.sleep()` in the main method, or in a `CommandLineRunner` 'run' method.

Here is an example that could be used, with a 'three minutes delay':

```
3*import org.springframework.boot.CommandLineRunner;
6
7 @SpringBootApplication
8 public class BackendService implements CommandLineRunner{
9
10     public static void main(String[] args) {
11         SpringApplication.run(BackendService.class, args);
12     }
13
14     @Override
15     public void run(String... args) throws Exception {
16         Thread.sleep(180000);
17     }
18 }
19 }
```

Figure 6.1. Java Thread Sleep Example (Author)

It would be beneficial to verify if there is a linear or exponential correlation between start-up time and self-healing capabilities when using warm pools.

6.4.2 AWS Tag-Based Resources

When creating compute instances in AWS, the use of 'tags' could have been leveraged.

That way, the AWS FIS tool would be able to randomize the termination of instances by tag, allowing for a wider variation of experiments, instead of the chosen path in this paper, where the FIS experiment had to be updated with each instance ID for every execution.

Tags can be added when creating resources in AWS, including EC2 instances:

The screenshot shows the 'Launch an instance' page in the AWS EC2 console. The breadcrumb navigation at the top reads 'EC2 > Instances > Launch an instance'. The main heading is 'Launch an instance' with an 'Info' link. Below the heading is a descriptive paragraph: 'Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.' The 'Name and tags' section is expanded, showing a table with two columns: 'Key' and 'Value'. The 'Key' column has a search icon and the text 'Name'. The 'Value' column has a search icon and the text 'test'. To the right of the 'Value' column is a 'Resource types' dropdown menu with the text 'Select resource ty...' and a downward arrow. Below the dropdown menu is a button labeled 'Instances'. To the right of the 'Resource types' dropdown menu is a 'Remove' button. Below the table is an 'Add new tag' button. At the bottom of the section is a note: 'You can add up to 49 more tags.'

Figure 6.2. AWS EC2 Console – Name and tags (Author)

During the experiment creation in AWS FIS, such tags can be referenced to define targets in the fault injection:

Edit target

Specify the target resources on which to run your selected actions. [Learn more](#)

Name

Instances-Target-1

The name must have 1 to 64 characters.

Resource type

aws:ec2:instance

Actions

KillInstance

Target method

☐ Resource IDs

☒ Resource tags, filters and parameters

► Service Quotas

Resource tags

Key

name

Value - optional

test

Remove

Figure 6.3. AWS FIS Console – Tagging targets (Author)

6.4.3 Compute-Size Instance Variation

Further work could also be carried out by executing a variation of this experiment, by changing the compute size of the instance from a t2.micro to a larger instance (i.e., t2.2xlarge or higher).

That can be accomplished by choosing the desired instance size during the EC2 Launch setup:

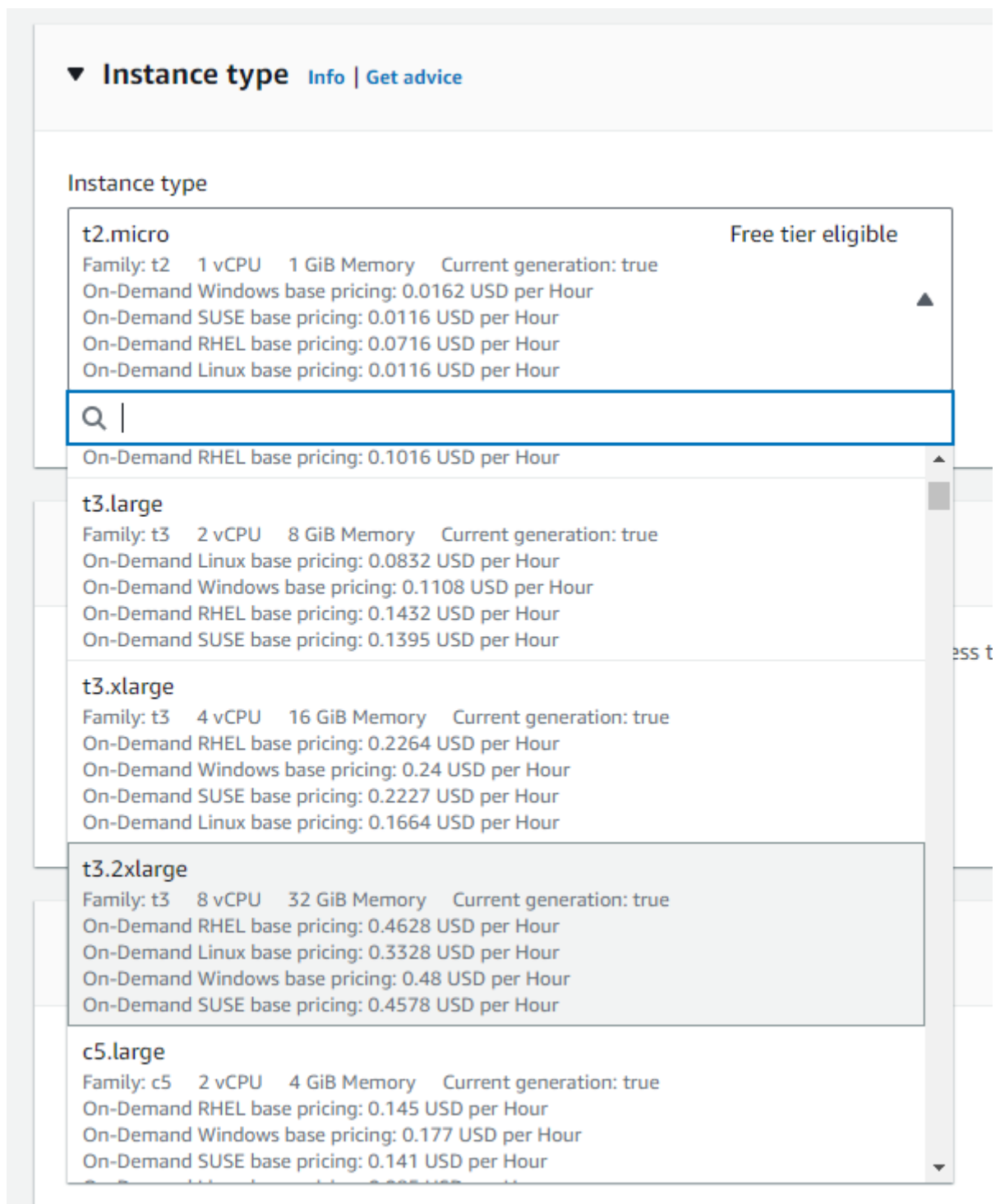


Figure 6.4. AWS EC2 Creation – Compute options (Author)

It can be theorized that larger instances will benefit more from warm pools, as they take longer to be commissioned. Therefore, the results will differ from this paper.

6.4.4 Cloud Provider Variation

Future work could be done by executing similar experiments within other cloud providers (i.e., Microsoft Azure and Google Cloud) to compare the results and challenge the providers claims.

The scope of the research would be much broader, given that an initial mapping of corresponding services between cloud providers would need to be carried out, to ensure the comparisons are reasonable and fair.

The research would be more technically challenging, as it would require in-dept knowledge in various cloud providers to successfully set up the experiments.

6.4.5 Multiple Microservices Variation

The current experiment had one microservice application running when injecting faults. Future work where multiple services interact with each other when faults are injected could be carried on. That would elucidate how to implement lowly coupled services to promote high availability when under stress.

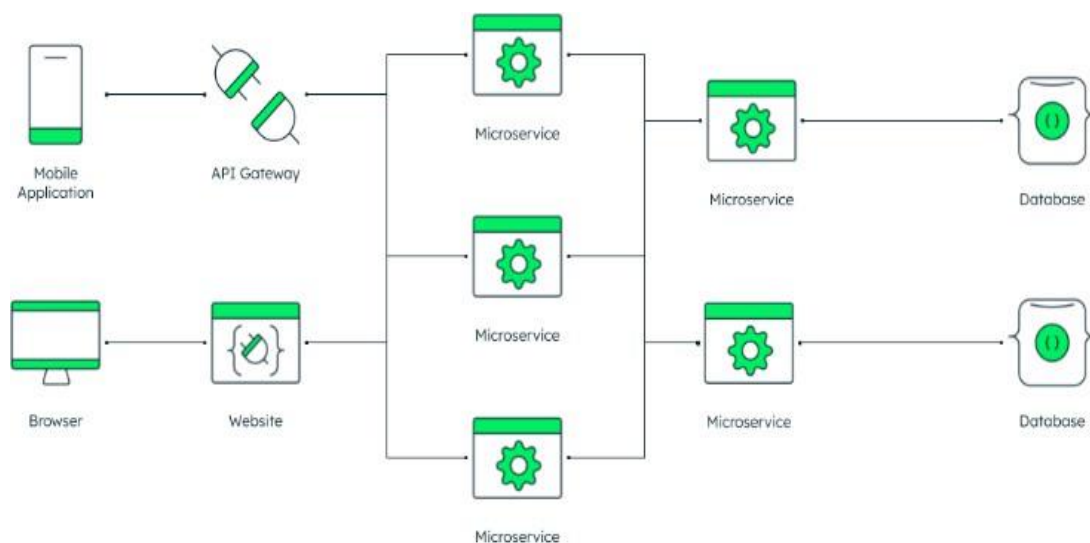


Figure 6.5. Microservice Architecture example (MongoDB website¹³)

6.4.6 Performance-based Research

Performance was not covered in this paper, therefore future work could explore how cloud-based infrastructure would behave when under stress load (i.e., heavy number of

¹³ <https://www.mongodb.com/databases/what-are-microservices>

users that exceed the system's breaking point).

For instance, similar research could be carried out where the AWS Auto Scaling group is created with an 'Average CPU Utilization' Target tracking scaling policy:

The screenshot displays the AWS Auto Scaling console configuration page. At the top, the 'Scaling' section includes an 'Info' icon and a note: 'You can resize your Auto Scaling group manually or automatically to meet changes in demand.' Below this, the 'Scaling limits' section allows setting 'Min desired capacity' to 1 and 'Max desired capacity' to 4. The 'Automatic scaling - optional' section shows two radio buttons: 'No scaling policies' (unselected) and 'Target tracking scaling policy' (selected). The 'Target tracking scaling policy' section is active, showing a 'Scaling policy name' of 'Target Tracking Policy', a 'Metric type' of 'Average CPU utilization', and a 'Target value' of 50. Informational text explains that the target tracking policy will choose a CloudWatch metric and target value to adjust capacity proportionally.

Scaling [Info](#)
You can resize your Auto Scaling group manually or automatically to meet changes in demand.

Scaling limits
Set limits on how much your desired capacity can be increased or decreased.

Min desired capacity
1
Equal or less than desired capacity

Max desired capacity
4
Equal or greater than desired capacity

Automatic scaling - optional
Choose whether to use a target tracking policy [Info](#)
You can set up other metric-based scaling policies and scheduled scaling after creating your Auto Scaling group.

☐ No scaling policies
Your Auto Scaling group will remain at its initial size and will not dynamically resize to meet demand.

☒ Target tracking scaling policy
Choose a CloudWatch metric and target value and let the scaling policy adjust the desired capacity in proportion to the metric's value.

Scaling policy name
Target Tracking Policy

Metric type [Info](#)
Monitored metric that determines if resource utilization is too low or high. If using EC2 metrics, consider enabling detailed monitoring for better scaling performance.

Average CPU utilization ▼

Target value
50

Figure 6.6. Auto Scaling – CPU utilization (Author)

Then the JMeter Test Plan would perform load testing with hundreds of simultaneous threads and verify how fast AWS can scale up.

6.4.7 Application Internal Fault Injection

Further work on executing a similar experiment, where the difference lies on how the Chaos Engineered fault is injected, could be conducted. Rather than using AWS FIS, the fault would come from the deployed application itself. For example, the following

Java/Spring code could be used to bring down a compute instance after a pre-defined number of minutes after the application was deployed:

```
26 @Bean
27 public CommandLineRunner CommandLineRunnerBean() throws IOException {
28     String shutdownCommand;
29     String operatingSystem = System.getProperty("os.name");
30
31     if ("Linux".equals(operatingSystem) || "Mac OS X".equals(operatingSystem)) {
32         shutdownCommand = "shutdown -h now";
33     }
34     // This will work on any version of windows including version 11
35     else if (operatingSystem.contains("Windows")) {
36         shutdownCommand = "shutdown.exe -s -t 0";
37     }
38     else {
39         throw new RuntimeException("Unsupported operating system.");
40     }
41
42     Runtime.getRuntime().exec(shutdownCommand);
43     System.exit(0);
44     return null;
45 }
```

Figure 6.7. Java/Spring Example – Shutting down instances (Author)

6.4.8 Future Work Conclusions

In conclusion, while this dissertation has made significant contributions to the understanding of self-healing systems, there exist several unexplored paths and opportunities for future researchers to continue building upon this work, advancing the fields of chaos engineering, microservice paradigm, cloud providers and self-healing architecture.

BIBLIOGRAPHY

- Stubbs, J., Moreira, W., & Dooley, R. (2015). Distributed Systems of Microservices Using Docker and Serfnode. *7th International Workshop on Science Gateways*, 34-39.
<https://doi.org/10.1109/IWSG.2015.16>
- Naqvi, M. A., Malik, S., Astekin, M., & Moonen, L. (2022). On Evaluating Self-Adaptive and Self-Healing Systems using Chaos Engineering. *3rd IEEE International Conference on Autonomic Computing and Self-Organizing Systems*, 1-10. <https://doi.org/10.1109/ACSOS55765.2022.00018>
- Wang, Y. (2019). Towards service discovery and autonomic version management in self-healing microservices architecture. *Proceedings of the 13th European Conference on Software Architecture*, 2(13), 63-66. <https://doi.org/10.1145/3344948.3344952>
- Mendonca, N. C., Jamshidi, P., Garlan, D., & Pahl, C. (2019). Developing Self-Adaptive Microservice Systems: Challenges and Directions. *IEEE Software*, 38(2), 70-79.
<https://doi.org/10.48550/arXiv.1910.07660>
- Filho, M., Pimentel, E., Pereira, W., Maia, P. H. M., & Cortes, M. I. (2021). Self-Adaptive Microservice-based Systems - Landscape and Research Opportunities. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 167-178.
<https://doi.ieeecomputersociety.org/10.1109/SEAMS51251.2021.00030>
- Garriga, M. (2018). Towards a Taxonomy of Microservices Architectures. In A. Cerone & M. Roveri (Eds.), *Software Engineering and Formal Methods. SEFM 2017. Lecture Notes in Computer Science*, vol 10729. Springer. https://doi.org/10.1007/978-3-319-74781-1_15
- Petrenko, S. A. (2021). Self-Healing Cloud Computing. *Voprosy kiberbezopasnosti*, 80-89.
<https://doi.org/10.21681/2311-3456-2021-1-80-89>
- Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, 23(3), 24-35. <https://doi.org/10.1109/MS.2018.2141039>
- Basiri, A., Behnam, N., Rooij, R. C., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos Engineering. *IEEE Software*, 33(3), 35-41.
<https://doi.ieeecomputersociety.org/10.1109/MS.2016.60>
- Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., ... Lang, M. (2016). Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 179-182. <https://doi.ieeecomputersociety.org/10.1109/MS.2016.60>
- Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., & Deardeuff, M. (2015). How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4), 66-73.
<https://doi.org/10.1145/2699417>
- Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., ... Bao, X. (2017). Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. *Proceedings of the 2017 ACM International Conference on Management of Data*, 1041-1052.
<https://doi.org/10.1145/3035918.3056101>
- Kotas, C. W., Naughton III, T. J., & Imam, N. (2018). A comparison of Amazon Web Services and Microsoft Azure cloud platforms for high performance computing. *IEEE Cloud Summit*.

<https://doi.org/10.1109/ICCE.2018.8326349>

Sen, A., & Skrobot, I. (2021). Implementation of DevOps paradigm to deployment and provisioning of microservices. *Issues in Information Systems*, 22(1), 136-148. https://doi.org/10.48009/1_iis_2021_136-148

Borge, S., & Poonia, N. (2020). Review on Amazon Web Services, Google Cloud Provider and Microsoft Windows Azure. *International Journal of Advance and Innovative Research*, 7(3), 49-54.

Zhang, H., Li, S., Jia, Z., Zhong, C., & Zhang, C. (2019). Microservice Architecture in Reality: An Industrial Inquiry. *IEEE International Conference on Software Architecture (ICSA)*, 51-60. <https://doi.org/10.1109/ICSA.2019.00014>

Blinowski, G., Ojdowska, A., & Przybylek, A. (2022). Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*, 10, 20357-20374. <https://doi.org/10.1109/ACCESS.2022.3152803>

Yussupov, V., Breitenbucher, U., Krieger, C., Leymann, F., Soldani, J., & Wurster, M. (2020). Pattern-based Modelling, Integration, and Deployment of Microservice Architectures. *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, 40-50. <https://doi.org/10.1109/EDOC49727.2020.00015>

Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., ... Patterson, D. A. (2021). What serverless computing is and should become: the next phase of cloud computing. *Communications of the ACM*, 64(5), 76-84. <https://doi.org/10.1145/3406011>

Jindal, A., Podolskiy, V., & Gerndt, M. (2019). Performance Modelling for Cloud Microservice Applications. *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, 25-32. <https://doi.org/10.1145/3297663.3310309>

Dashofy, E. M., Hoek, A. V. D., & Taylor, R. N. (2002). Towards architecture-based self-healing systems. *Proceedings of the first workshop on Self-healing systems (WOSS '02)*, 21-26. <https://doi.org/10.1145/582128.582133>

Frincu, M. E., Villegas, N. M., Pectu, D., Muller, H. A., & Rouvoy, R. (2011). Self-Healing Distributed Scheduling Platform. *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 225-234. <https://doi.org/10.1109/CCGrid.2011.23>

Rios, E., Iturbe, E., & Palacios, M. C. (2017). Self-healing Multi-Cloud Application Modelling. *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES '17)*, Article 93, 1-9. <https://doi.org/10.1145/3098954.3104059>

Seeger, J., Broring, A., & Carle, G. (2020). Optimally Self-Healing IoT Choreographies. *ACM Transactions on Internet Technology*, 20(3), 1-20. <https://doi.org/10.1145/3386361>

Dias, J. P., Sousa, T. B., Restivo, A., & Ferreira, H. S. (2020). A Pattern-Language for Self-Healing Internet-of-Things Systems. *Proceedings of the European Conference on Pattern Languages of Programs*, Article 25, 1-17. <https://doi.org/10.1145/3424771.3424804>

Colombo, V., Tundo, A., Ciavotta, M., & Mariani, L. (2022). Towards self-adaptive peer-to-peer monitoring for fog environments. *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 156-166. <https://doi.org/10.1145/3524844.3528055>

Eismann, S., Shang, W., Bezemer, C. P., & Okanovic, D. (2020). Microservices: A Performance Tester's

Dream or Nightmare?. *International Conference on Performance Engineering*, 1-13.
<http://dx.doi.org/10.1145/3358960.3379124>

Mendonca, N. C., Garlan, D., Schmerl, B., & Camara, J. (2018). Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices. *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, Article 18, 1-6.
<https://doi.org/10.1145/3241403.3241423>

Migirditch, S., Asplund, J., & Curran, W. (2022). Chaos engineering: stress-testing algorithms to facilitate resilient strategic military planning. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2160-2167. <https://doi.org/10.1145/3520304.3533962>

Basiri, A., Hochstein, L., Jones, N., & Tucker, H. (2019). Automating Chaos Experiments in Production. *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 31-40. <https://doi.org/10.1109/ICSE-SEIP.2019.00012>

Velepucha, V. and Flores, P., 2023. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*.

Ghosh, D., Sharman, R., Rao, H.R., & Upadhyaya, S. (2007). Self-healing Systems - Survey and Synthesis. *Decision Support Systems*, 42(4), 2164-2185.

Jernberg, H., Runeson, P., & Engström, E. (2020). Getting Started with Chaos Engineering-design of an implementation framework in practice. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1-10.

Qian, L., Luo, Z., Du, Y., & Guo, L. (2009). Cloud Computing: An Overview. In *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*, 626-631. Springer Berlin Heidelberg.

Rosenthal, C., & Jones, N. (2020). *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media.

Zhang, M. (2023, January 1). Top 10 Cloud Service Providers Globally in 2023. DgtlInfra.
<https://dgtlinfra.com/top-10-cloud-service-providers-2022/>

Amazon Web Services. (2023, April 10). AWS Well-Architected Framework. AWS.
<https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html>

Estes, A. (2023). *AWS Certified Solutions Architect - Associate (SAA-C03)* [MOOC]. Pluralsight. [AWS Certified Solutions Architect - Associate \(SAA-C03\) | Pluralsight](#)

Ellis, F. (2023). *Hands-On Chaos Engineering with AWS Fault Injection Simulator* [MOOC]. Pluralsight. [Hands-On Chaos Engineering with AWS Fault Injection Simulator \(pluralsight.com\)](#)

APPENDIX A: MY JOURNEY

I would like to take the time to describe my personal journey, and what led me to the decision of writing this dissertation.

I graduated in Software Engineering in 2014, but my career as a programmer started in 2011, when got my first job as an intern programmer at a small IT company developing software for advocacy agencies.

Since then, I've come a long way, certifying myself in many different programming languages, cloud platforms, agile methodologies, IT services standardizations, as well as getting exposure to different companies, different business domains, and different organizational cultures. I'm currently a Lead Software Engineer at a major FinTech company.

I've started this master's course back in 2017 and finished all the credits except for the dissertation by 2019, however, due to personal issues I was not able to submit the dissertation's proposal in 2019 and had to take a study break.

Once my personal life got back on track, in late February of 2023 I've made the decision to finish the masters.

At the time, I was very interested in cloud architecture and microservices and would love to have my thesis in this field, so I have decided to recertify myself as an AWS Architect, to make sure I was up to date with the latest services, options, and configurations offered by AWS.

I already had previously certified myself in 2017 as an AWS Associate Solutions Architect, as well as an AWS Associate Developer, however these certifications expire every 3 years, so I saw it as the perfect opportunity to get recertified.

I have covered the 'A Cloud Guru – AWS Certified Solutions Architect – Associate (SAA-CO3)' course, read through the recommended AWS whitepapers, covered multiple mock exams from 'Tutorials Dojo, sat the exam and passed:



AWS Certified Solutions Architect - Associate

Notice of Exam Results

Candidate: Bruno Franco	Exam Date: Jun 14, 2023
Candidate ID: AWS03443638	Registration Number: 444024411
Candidate Score: 776	Pass/Fail: PASS

Congratulations! You have successfully completed the AWS Certified Solutions Architect - Associate and you are now AWS Certified.

With that out of the way, I've started my literature review in June 2023 to look for inspiration on what type of pioneering research I could conduct in the cloud space.

That's when I came across the academic concepts of self-adaptive and self-healing, as well as Chaos Engineering, and I thought that together, these concepts had a lot of potential for research.

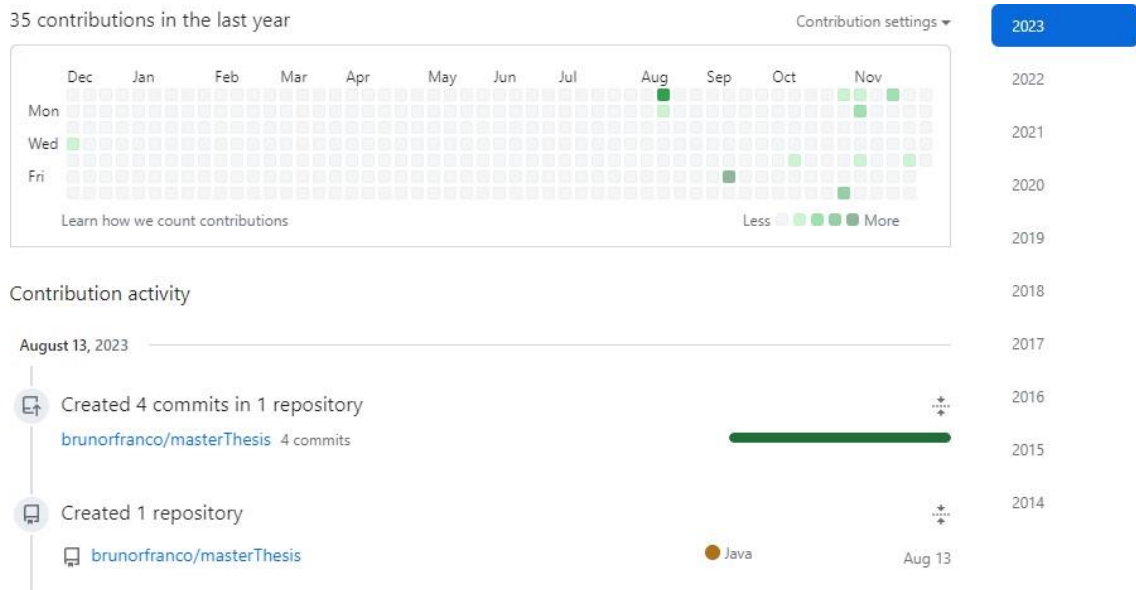
It was challenging to find a Chaos engineering tool freely available, especially one that could interact with AWS resources. That's when I came across the AWS FIS service. I've enrolled myself on the course 'Hands-On Chaos engineering with AWS Fault Injection Simulator' by A Cloud Guru, and by the end of the course I had a clear idea on how to use this tool to help me with the research.

Given my previous years of experiences as a Software Engineer, I naturally selected JMeter for the task of independently health-check the Elastic Load Balancer. Therefore, my review of JMeter was less structured. I've found a few good tutorials on YouTube, did a couple of exploratory tests with it, ensured it would be fit for purpose and settled for it. In hindsight, I could have used two separate tools to ensure the metrics collected in the experiment were validated by another independent tool (this idea was also brought up by an expert in the field during the interviews).

I've also written all the necessary Java code for the experiment in August 2023, in preparation for the dissertation, so I could hit the ground running when the time came for the experiment execution and the dissertation writing.

That can be confirmed by the dates shown in the contribution activities in my GitHub,

where I stored all the material for this dissertation:



As can be seen, the initial code was pushed on the 13th of August.

Once the dissertation proposal was approved in September, I already had a clear path to follow. I knew that the experiment design was feasible, and the chosen tools were fit for purpose, so I only had minor challenges when implementing the experiment and collecting data.

APPENDIX B: EXPERIMENT USER GUIDE

Please note that this user guide was revised, reedited, and reviewed when a complete redevelopment of the system infrastructure was undertaken to assess the effectiveness of these instructions.

1. Java Micro-service Implementation

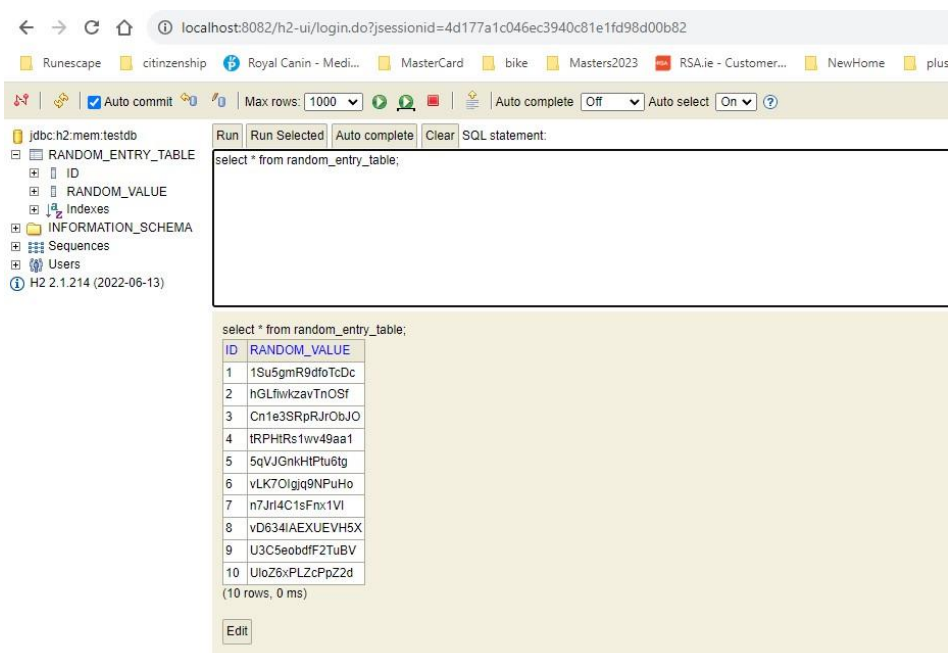
The ‘backend-service’ Java application was developed and compiled in JavaSE-17, in the Eclipse IDE version 2023-09 (4.29.0). It was built with maven, Spring Boot 3.1.2, Spring Boot Starter Data JPA, Spring boot Starter Web and H2 database.

```

6< <parent>
7  <groupId>org.springframework.boot</groupId>
8  <artifactId>spring-boot-starter-parent</artifactId>
9  <version>3.1.2</version>
10 <relativePath /> <!-- lookup parent from repository -->
11 </parent>
12 <groupId>com.tu.bruno</groupId>
13 <artifactId>backend-service</artifactId>
14 <version>0.0.1-SNAPSHOT</version>
15 <name>backend-service</name>
16 <description>Backend Service for Bruno's Master thesis</description>
17< <properties>
18   <java.version>17</java.version>
19 </properties>
20< <dependencies>
21<   <dependency>
22     <groupId>org.springframework.boot</groupId>
23     <artifactId>spring-boot-starter-data-jpa</artifactId>
24   </dependency>
25<   <dependency>
26     <groupId>org.springframework.boot</groupId>
27     <artifactId>spring-boot-starter-web</artifactId>
28   </dependency>
29
30<   <dependency>
31     <groupId>com.h2database</groupId>
32     <artifactId>h2</artifactId>
33     <scope>runtime</scope>
34   </dependency>
35<   <dependency>
36     <groupId>org.springframework.boot</groupId>
37     <artifactId>spring-boot-starter-test</artifactId>
38     <scope>test</scope>
39   </dependency>
40
41<   <dependency>
42     <groupId>org.apache.commons</groupId>
43     <artifactId>commons-lang3</artifactId>
44   </dependency>
45 </dependencies>

```

It contains a single model called RandomEntry, which maps to a database table named 'RandomEntryTable', with two columns, 'id' and 'randomValue'. The application also exposes a Rest API under port 8082, URL '/api/entries' which returns all the rows of the RandomEntryTable in JSON format. When the application starts up, it executes a command to insert ten random rows into the in-memory database. Once the application is running, the database can be interacted with from the '/h2-ui' URL:



localhost:8082/h2-ui/login.do?jsessionid=4d177a1c046ec3940c81e1fd98d00b82

Run | Run Selected | Auto commit | Max rows: 1000 | Auto complete | Off | Auto select | On

jdbc:h2:mem:testdb

RANDOM_ENTRY_TABLE

ID | RANDOM_VALUE

1 | 1Su5gmR9dfoTcDc

2 | hGLfwkzavTnOSf

3 | Cn1e3SRpRrObJO

4 | tRPHIRs1wv49aa1

5 | 5qVJGnkHtPu6tg

6 | vLK7Olgig9NPuHo

7 | n7JrI4C1sFnx1VI

8 | vD634lAEXUEVH5X

9 | U3C5eobdFF2TuBV

10 | UloZ6xPLZcPpZ2d

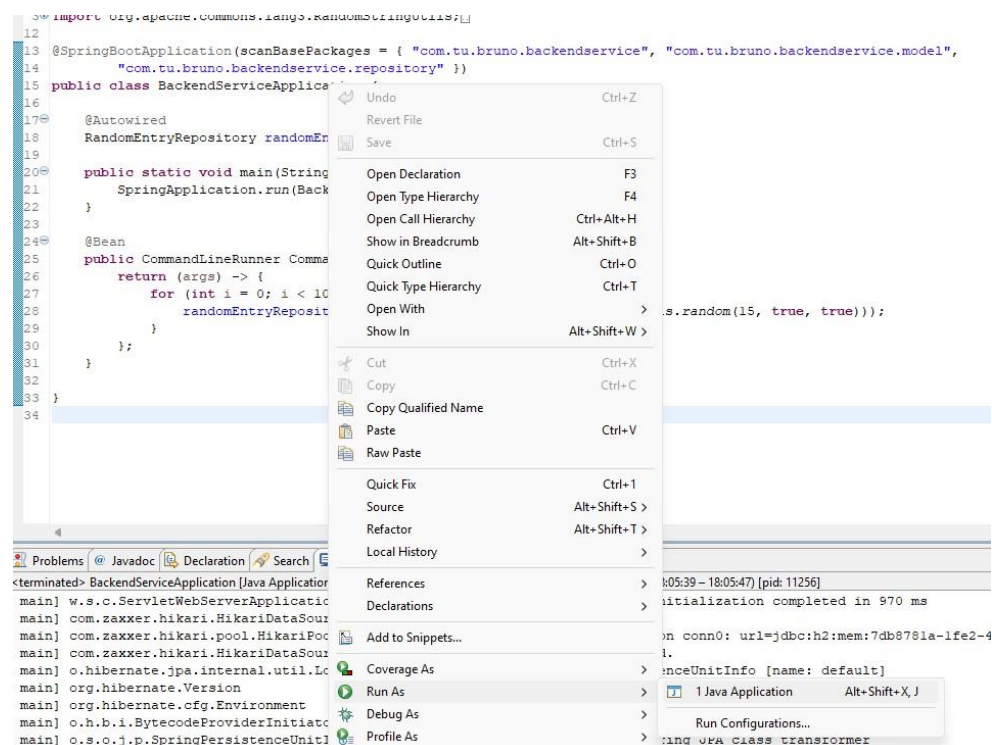
(10 rows, 0 ms)

Edit

Login details can be found at the 'application.properties' file:

```
1 server.port=8082
2
3 spring.datasource.url=jdbc:h2:mem:testdb
4 spring.datasource.driverClassName=org.h2.Driver
5 spring.datasource.username=sa
6 spring.datasource.password=
7
8 spring.jpa.show-sql=true
9 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
10 spring.jpa.hibernate.ddl-auto= update
11 spring.jpa.defer-datasource-initialization=true
12
13 spring.sql.init.mode=always
14
15 spring.h2.console.enabled=true
16 # default path: h2-console
17 spring.h2.console.path=/h2-ui
```

The application can be executed by running the 'BackendServiceApplication.java' class:



The application's code can be found in its entirety at

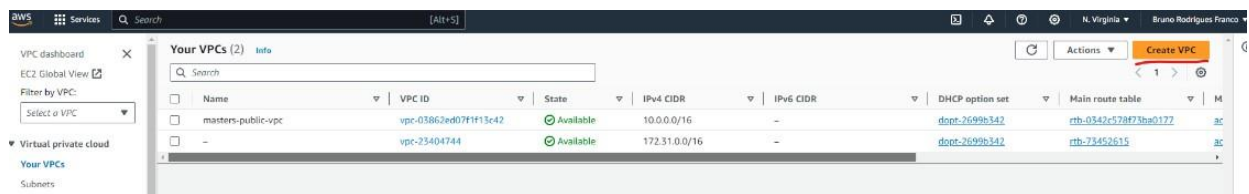
<https://github.com/brunorfranco/masterThesis/tree/main/backend-service>.

2. AWS Cloud Architecture implementation

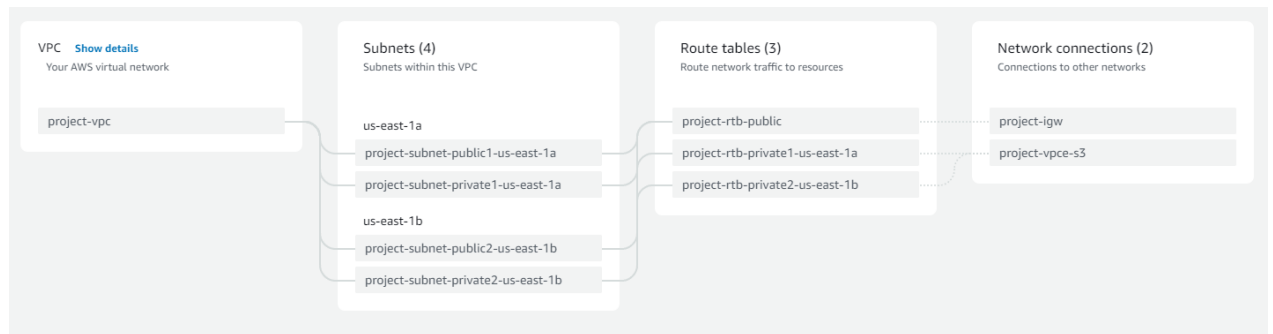
2.1 VPC Creation

The first step towards creating the cloud architecture is setting up a virtual private cloud.

Once logged into the AWS console, open the VPC section, click on ‘Create VPC’:



When doing it through the UI wizard, select ‘VPC and more’, by default that will create four subnets (one public and one private for two separate availability zones). It will also create three route tables (one public and two private), as well as two network connections:



Also provide the IPv4 CIDR, enter '10.0.0.0/24'. The default values for the other fields are sufficient to proceed with the setup:

Create VPC workflow

✓ Success

▼ Details

- ✓ Create VPC: [vpc-05da7623501febc74](#)
- ✓ Enable DNS hostnames
- ✓ Enable DNS resolution
- ✓ Verifying VPC creation: [vpc-05da7623501febc74](#)
- ✓ Create S3 endpoint: [vpce-0f25a70f9cc64b94a](#)
- ✓ Create subnet: [subnet-048704e79a2b50a2d](#)
- ✓ Create subnet: [subnet-0633309e122061e27](#)
- ✓ Create subnet: [subnet-08b5e3b2ec3176d6c](#)
- ✓ Create subnet: [subnet-09d999aea2753b975](#)
- ✓ Create internet gateway: [igw-0da427073f9ebae80](#)
- ✓ Attach internet gateway to the VPC
- ✓ Create route table: [rtb-0c1666f09c1e3c502](#)
- ✓ Create route
- ✓ Associate route table
- ✓ Associate route table
- ✓ Create route table: [rtb-007e8b16a66e7cb28](#)
- ✓ Associate route table
- ✓ Create route table: [rtb-0880c0ae14f9992ec](#)
- ✓ Associate route table
- ✓ Verifying route table creation
- ✓ Associate S3 endpoint with private subnet route tables: [vpce-0f25a70f9cc64b94a](#)

[View VPC](#)

2.2 Key Pair Creation

Once logged into the AWS console, navigate to the ‘Key Pair’ section, click ‘Create key pair’.

Key pairs (1) Info					
<input type="text" value="Search"/>					
<input type="checkbox"/>	Name	Type	Created	Fingerprint	ID
<input type="checkbox"/>	master-thesis	rsa	2023/10/12 15:42 GMT+1	14:a0:2ec1:cd:66:e1:e3:db:55:e0:bb:d4:...	key-034f7f239085a78bd

Once logged into the AWS console, navigate to the ‘Key Pair’ section, click ‘Create key pair’. Type a unique name, leave the default options for RSA and .ppk format, and click on ‘Create Key pair’:

Create key pair [Info](#)

Key pair

A key pair, consisting of a private key and a public key, is a set of security credentials that you use to prove your identity when connecting to an instance.

Name

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type [Info](#)

☒ RSA

☐ ED25519

Private key file format

☐ .pem
For use with OpenSSH

☒ .ppk
For use with PuTTY

Tags - *optional*

No tags associated with the resource.

Add new tag

You can add up to 50 more tags.

Cancel

Create key pair

That will automatically trigger a download, save this file in a secure location as this cannot be found anywhere else.

2.3 (Windows users) Install Putty

To prepare for the next steps, the download and installation of Putty to be able to SSH into remote machines is necessary. If users have downloaded in .pem format file, then use Puttygen to convert it to .ppk.

2.4 EC2 Creation

Navigate to the EC2 dashboard, click on 'Launch Instance', enter a name, quick select the Amazon Linux AWS option under 'Amazon Machine Image':

Launch an instance [Info](#)

Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags [Info](#)

Name

[Add additional tags](#)

▼ Application and OS Images (Amazon Machine Image) [Info](#)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

[Recents](#)

[My AMIs](#)

[Quick Start](#)



[Browse more AMIs](#)

Including AMIs from AWS, Marketplace and the Community

This will spin up an Amazon Linux 2023 x86_64 HVM kernel-6.1.

Select the key pair created on step 2 under the 'Key pair (login)' section:

▼ **Key pair (login)** [Info](#)

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - *required*

▲

Proceed without a key pair (Not recommended) Default value

master-thesis ✓
Type: rsa

[Create new key pair](#)

[Edit](#)

Under ‘VPC – required’, select the VPC created in step 2.1 and select one of the subnets created with it:

▼ **Network settings** [Info](#)

VPC - *required* [Info](#)

▲

vpc-0bf4b5d40a6786ebe (interview-experiment-vpc) ✓
10.0.0.0/16

vpc-23404744 (default)
172.31.0.0/16

[Create](#)

Auto-assign public IP [Info](#)

▼

Under the ‘Network settings’, select ‘Create security group’, this will simplify the setup steps as it will be created with SSH traffic already allowed. Also tick the box to ‘Allow HTTP traffic from the internet’, as a Rest API endpoint will be exposed by the Java application:

▼ Network settings Info

Edit

Network Info

vpc-23404744

Subnet Info

No preference (Default subnet in any availability zone)

Auto-assign public IP Info

Enable

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

☒ Create security group

☐ Select existing security group

We'll create a new security group called 'launch-wizard-1' with the following rules:

☒ Allow SSH traffic from

Helps you connect to your instance

Anywhere
0.0.0.0/0

☐ Allow HTTPS traffic from the internet

To set up an endpoint, for example when creating a web server

☒ Allow HTTP traffic from the internet

To set up an endpoint, for example when creating a web server

⚠ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

×

Make sure that the 'Auto-assign public IP' option is enabled so that is assigned to the primary network interface of the instance.

Launch Instance:

▼ Summary

Number of instances [Info](#)

1

Software Image (AMI)

Amazon Linux 2023 AMI 2023.2.2...[read more](#)
ami-0dbc3d7bc646e8516

Virtual server type (instance type)

t2.micro

Firewall (security group)

New security group

Storage (volumes)

1 volume(s) - 8 GiB

❏

Free tier: In your first year includes 750 hours of t2.micro (or t3.micro in the Regions in which t2.micro is unavailable) instance usage on free tier AMIs per month, 30 GiB of EBS storage, 2 million IOs, 1 GB of snapshots, and 100 GB of bandwidth to the internet.

×

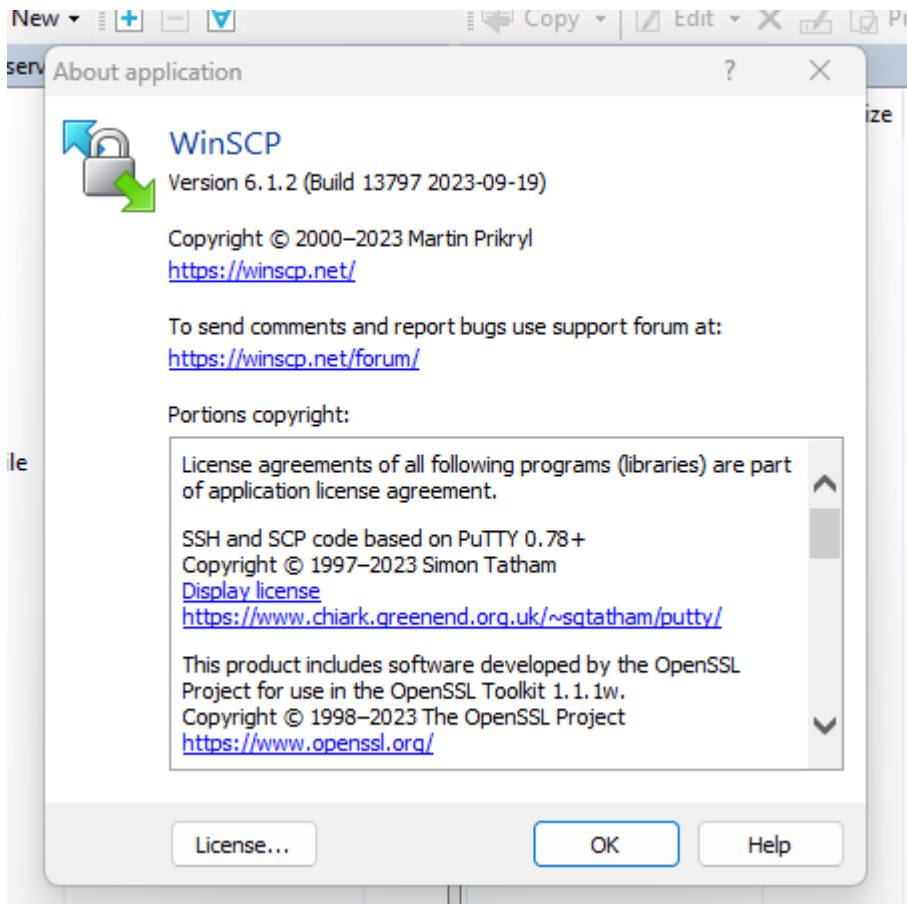
Cancel

Launch instance

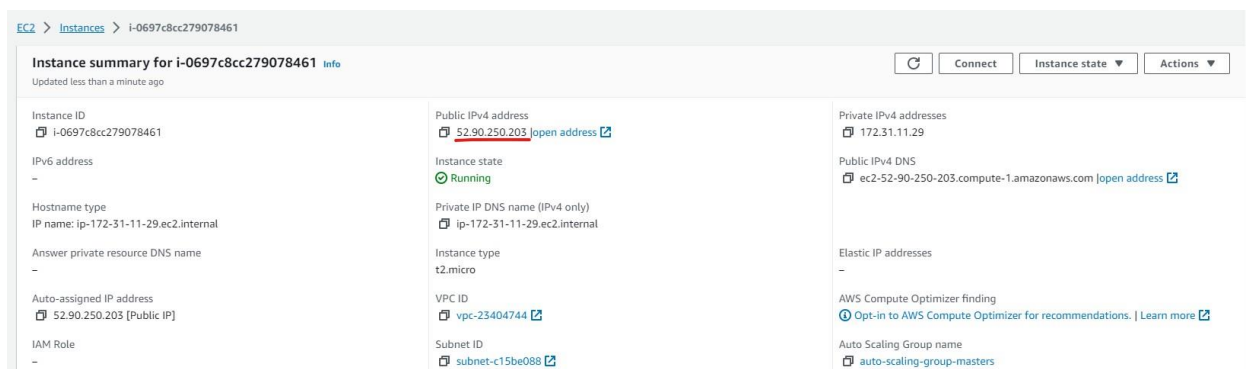
[Review commands](#)

2.5 Adding Java Application file into the EC2 instance

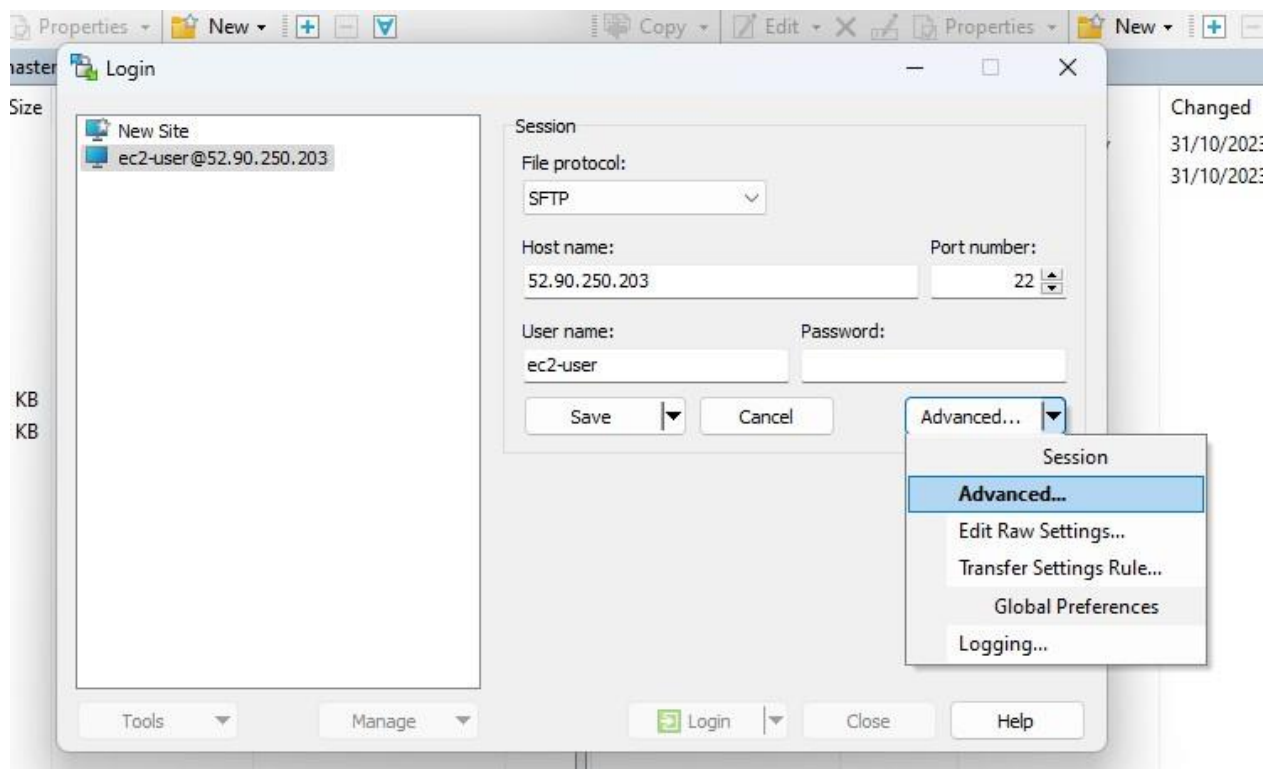
To add the .jar file into the EC2 instance, use the WinSCP tool, version 6.1.2, build 13797 2023-09-19:



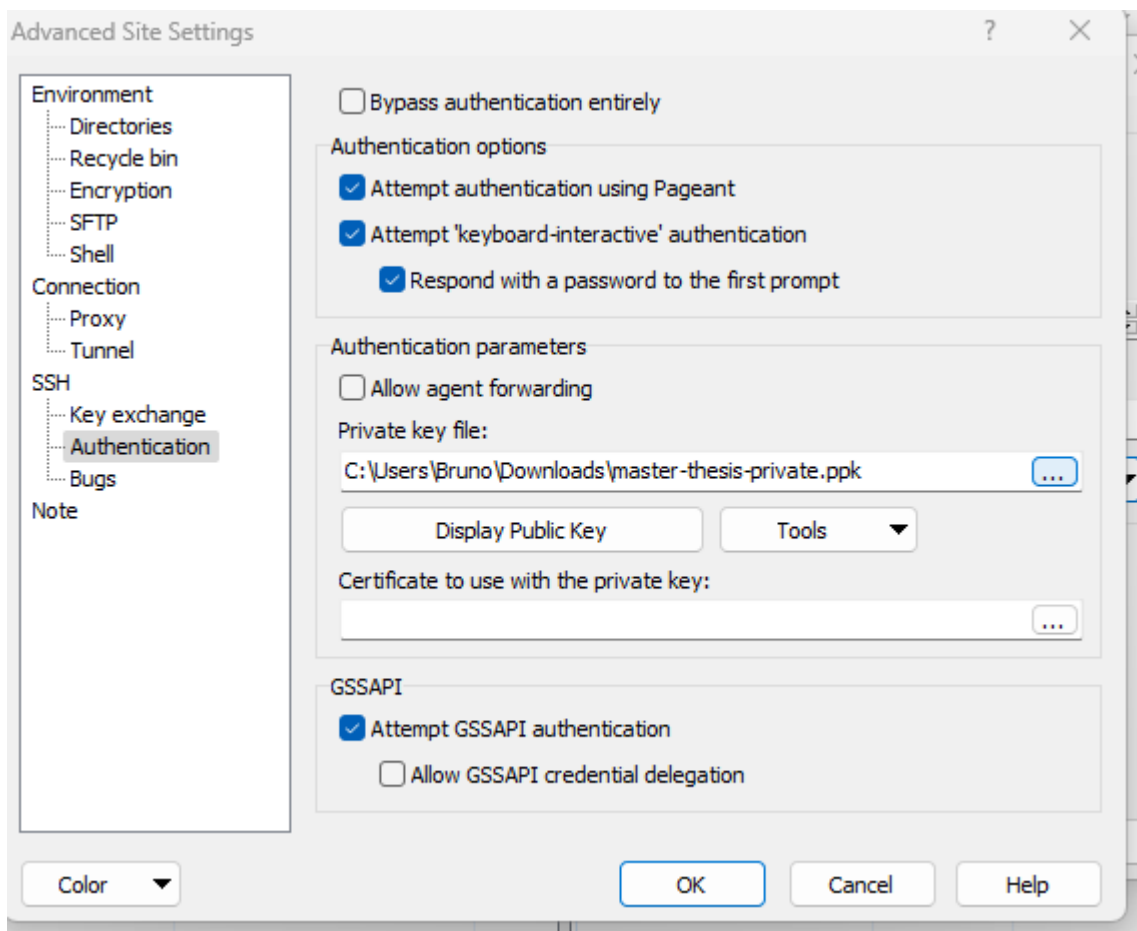
Once in WinSCP, click 'New Tab' to start a new connection, then provide the EC2 IPv4 address (found in the EC2 details as per image below) into the Host name field, port 22. The default username is 'ec2-user'.



Before trying to connect, the .ppk key file generated on step 2 needs to be referenced, so go to 'Advanced...', then 'Advanced...' again:

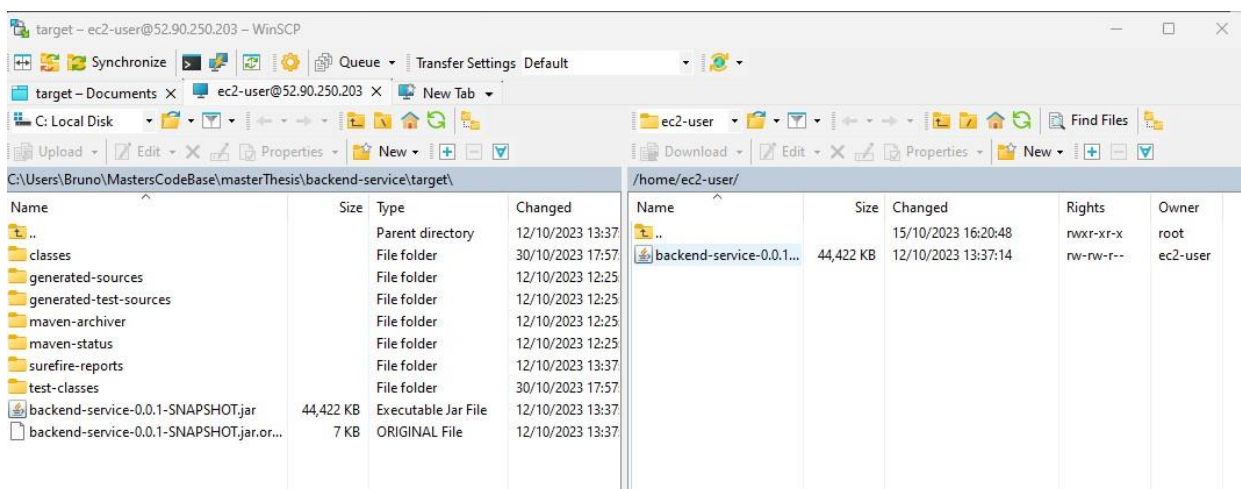


Under ‘SSH’, ‘Authentication’, provide the location of the private key under the field ‘Private key file’:



Once this is in place, continue to click ‘Login’ and the connection will be established.

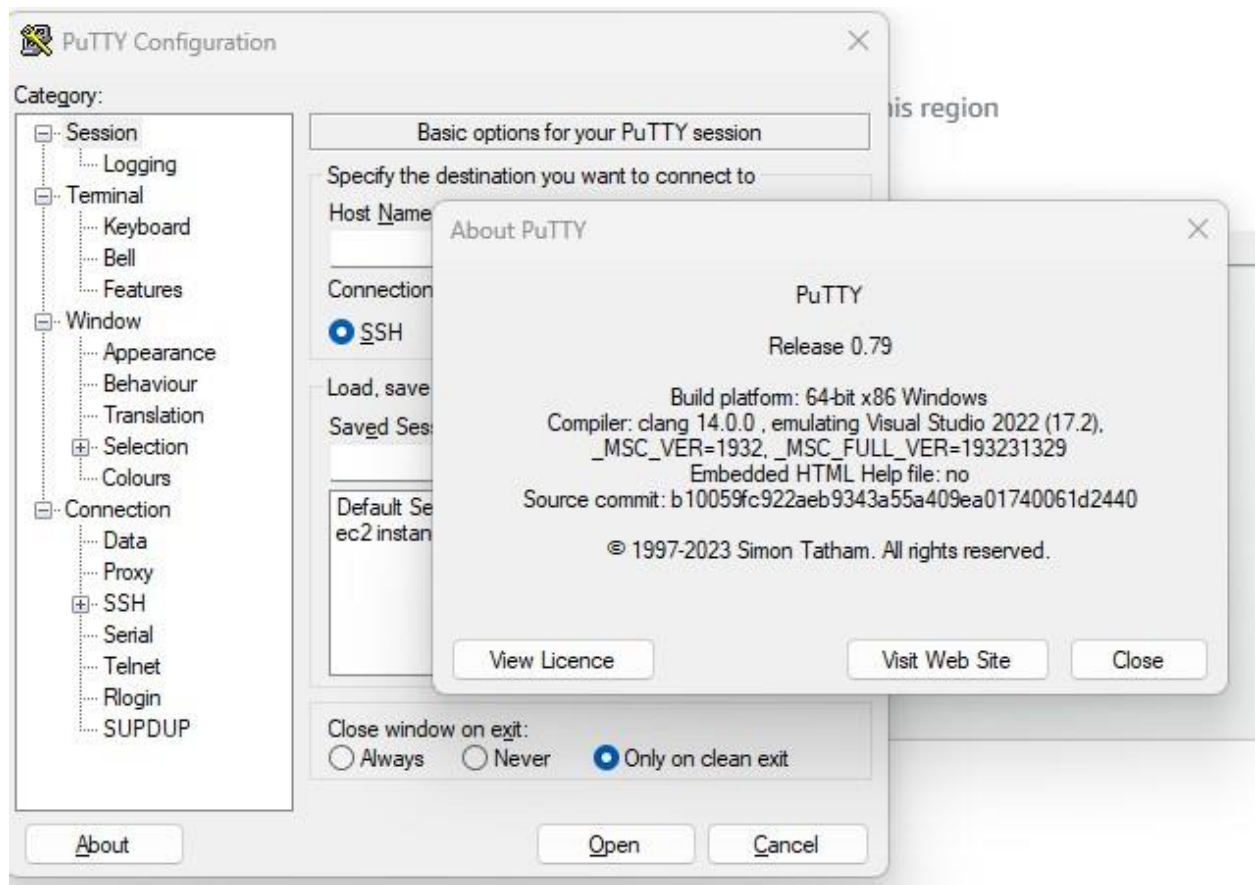
Copy the backend-service-0.0.1-SNAPSHOT.jar file from the Java application ‘target’ folder, and paste it inside the EC2 instance, under the ‘/home/ec2-user/’ folder:



2.6 Connecting into EC2 through Putty and running the application

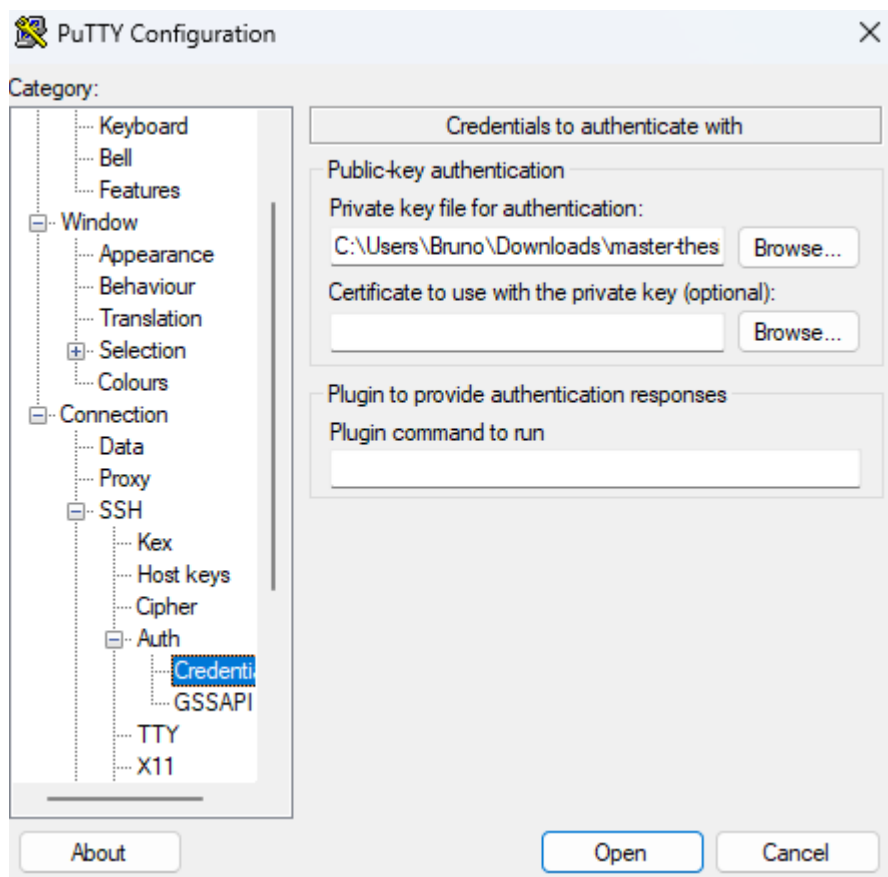
Once the .jar file is loaded into the EC2 instance, then it is time to SSH into the instance through the Putty command line and run it.

To achieve that, use Putty version 0.79:



To open a new session from Putty, go back to the AWS console and get the Public IPv4 address from the instance, then insert it into the Putty Host Name and use port 22.

Under 'Connection', open 'SSH', then 'Auth', then 'Credentials', then browse and select the .ppk key file that was generated on step 2.



Go back to the 'Session' section, select a unique name and save the session, that way this configuration can be reused in the future. Click 'Open' and the SSH connection will be established:

EC2 > Instances > i-0697c8cc279078461

Instance summary for i-0697c8cc279078461 [Info](#)

Updated less than a minute ago

Instance ID i-0697c8cc279078461	Public IPv4 address 52.90.250.203 open address
IPv6 address -	Instance state ✔ Running
Hostname type IP name: ip-172-31-11-29.ec2.internal	Private IP DNS name (IPv4 only) ip-172-31-11-29.ec2.internal
Answer private resource DNS name -	Instance type t2.micro
Auto-assigned IP address 52.90.250.203 [Public IP]	VPC ID vpc-23404744
IAM Role -	Subnet ID subnet-c15be088
IMDSv2 Required	

[Details](#) | [Security](#) | [Networking](#) | [Storage](#) | [Status checks](#) | [Monitoring](#) | [Tags](#)

▼ Security details

IAM Role -	Owner ID 096958155378
Security groups sg-05836dcfcd6c22680 (masters-security-group)	

Make sure there is an inbound rule for SSH, with Source 0.0.0.0/0 (Anywhere).

This is not a safe approach, as anyone holding the private key could connect to the instance, so the key should be kept in a safe place.

Once in, ensure that the .jar file is indeed there and accessible by executing an 'ls -l' command:

```

_/_m/'
Last login: Sun Oct 15 15:52:33 2023 from 145.224.69.87
[ec2-user@ip-172-31-11-29 ~]$ ls -l
total 44424
-rw-rw-r--. 1 ec2-user ec2-user 45487177 Oct 12 12:37 backend-service-0.0.1-SNAP
SHOT.jar
[ec2-user@ip-172-31-11-29 ~]$

```

These Linux instances do not come with Java pre-installed, therefore please execute the commands in this order:

1. `sudo su`
2. `yum update -y`
3. `sudo yum install java-17-amazon-corretto-headless`

```
root@ip-10-0-8-17:/home/ec2-user
PSHOT.jar
[ec2-user@ip-10-0-8-17 ~]$ sudo su
#[root@ip-10-0-8-17 ec2-user]# yum update -y
Last metadata expiration check: 0:06:02 ago on Tue Nov 28 19:11:56 2023.
Dependencies resolved.
Nothing to do.
Complete!
[root@ip-10-0-8-17 ec2-user]# sudo yum install java-17-amazon-corretto-headless
Last metadata expiration check: 0:07:03 ago on Tue Nov 28 19:11:56 2023.
Dependencies resolved.
=====
Package                                Arch    Version                                Repository    Size
=====
Installing:
java-17-amazon-corretto-headless      x86_64  1:17.0.9+8-1.amzn2023.1              amazonlinux   91 M
Installing dependencies:
alsa-lib                               x86_64  1.2.7.2-1.amzn2023.0.2              amazonlinux   504 k
cairo                                   x86_64  1.17.6-2.amzn2023.0.1              amazonlinux   684 k
dejavu-sans-fonts                      noarch  2.37-16.amzn2023.0.2              amazonlinux   1.3 M
dejavu-sans-mono-fonts                 noarch  2.37-16.amzn2023.0.2              amazonlinux   467 k
dejavu-serif-fonts                     noarch  2.37-16.amzn2023.0.2              amazonlinux   1.0 M
fontconfig                             x86_64  2.13.94-2.amzn2023.0.2             amazonlinux   273 k
fonts-filessystem                      noarch  1:2.0.5-12.amzn2023.0.2            amazonlinux   9.5 k
=====
```

After that, then the application can be started by executing “`java -jar /home/ec2-user/backend-service-0.0.1-SNAPSHOT.jar &`”:

EC2 > Security Groups > sg-0ae1681d9ec624056 - launch-wizard-1

sg-0ae1681d9ec624056 - launch-wizard-1 Actions ▾

Details

Security group name
launch-wizard-1

Security group ID
sg-0ae1681d9ec624056

Description
launch-wizard-1 created 2023-11-28T19:00:27.469Z

VPC ID
vpc-0bf4b5d40a6786ebe

Owner
096958155378

Inbound rules count
4 Permission entries

Outbound rules count
1 Permission entry

Inbound rules Outbound rules Tags

Inbound rules (4)
Manage tags
Edit inbound rules

<input type="checkbox"/>	Name ▾	Security group rule... ▾	IP version ▾	Type ▾	Protocol ▾	Port range ▾	Source ▾	Description
<input type="checkbox"/>	-	sgr-052777261bcd9b...	IPv4	HTTPS	TCP	443	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-048b5623b83a58c...	IPv4	Custom TCP	TCP	8082	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-00f0ae84ed0580a94	IPv4	SSH	TCP	22	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-06eb0a7e1a6c9abc9	IPv4	HTTP	TCP	80	0.0.0.0/0	-

2.7 AMI Extraction

With a healthy EC2 instance fully setup, then it is time to create an Amazon Machine Image (AMI), so that other instances can be spun up from the same setup.

From the ‘EC2 Dashboard’ in the AWS console, select the healthy EC2 instance, then select ‘Actions’, then ‘Image and templates’, then ‘Create image’:

Instances (1/2) Info

Find Instance by attribute or tag (case-sensitive)

Instance state = running X Clear filters

<input type="checkbox"/>	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
<input checked="" type="checkbox"/>		i-0697c8cc279078461	Running	t2.micro	2/2 checks passed	No alarms	us-east-1a	ec2-52-90-250-203.co...
<input type="checkbox"/>		i-0a56155b477cfa566	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	ec2-34-229-211-121.co...

Create image
Create template from instance
Launch more like this

Connect
View details
Manage instance state
Instance settings
Networking
Security
Image and templates
Monitor and troubleshoot

A name for the image needs to be selected, then click the ‘Create Image’ button:

imageTest

Maximum 127 characters. Can't be modified after creation.

Image description - optional

Image description

Maximum 255 characters

No reboot

☐ Enable

Instance volumes

Storage type	Device	Snapshot	Size	Volume type	IOPS	Throughput	Delete on termination	Encrypted
EBS	/dev/...	Create new snapshot fr...	8	EBS General Purpose S...	3000		<input checked="" type="checkbox"/> Enable	<input type="checkbox"/> Enable

Add volume

ⓘ

During the image creation process, Amazon EC2 creates a snapshot of each of the above volumes.

Tags - optional

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

☒ Tag image and snapshots together
 Tag the image and the snapshots with the same tag.

☐ Tag image and snapshots separately
 Tag the image and the snapshots with different tags.

No tags associated with the resource.

Add new tag

You can add up to 50 more tags.

Cancel

Create image

2.8 Launch template:

To create a Launch template, go into the EC2 dashboard, then 'Auto Scaling', then 'Auto Scaling Groups'.

aws Services Search [Alt+S]

- Launch Templates
- Spot Requests
- Savings Plans
- Reserved Instances
- Dedicated Hosts
- Capacity Reservations
- New
- ▼ Images
 - AMIs
 - AMI Catalog
- ▼ Elastic Block Store
 - Volumes
 - Snapshots
 - Lifecycle Manager
- ▼ Network & Security
 - Security Groups
 - Elastic IPs
 - Placement Groups
 - Key Pairs
 - Network Interfaces
- ▼ Load Balancing
 - Load Balancers
 - Target Groups
- ▼ Auto Scaling
 - Auto Scaling Groups

Auto Scaling group updated successfully

EC2 > Auto Scaling groups

Auto Scaling groups (1) Info

Search your Auto Scaling groups

<input type="checkbox"/>	Name	Launch template/configuration	Instances
<input type="checkbox"/>	auto-scaling-group-masters	masters-launch-template Version Latest	2

0 Auto Scaling groups selected

From there, then click on 'Launch Templates':

EC2 > Auto Scaling groups

Auto Scaling groups (1) Info

Search your Auto Scaling groups

Launch configurations Launch templates Launch templates Actions Create Auto Scaling group

Opens in a new tab

<input type="checkbox"/>	Name	Launch template/configuration	Instances	Status	Desired capacity	Min	Max	Availability Zones
<input type="checkbox"/>	auto-scaling-group-masters	masters-launch-template Version Latest	2	-	2	2	4	us-east-1a, us-east-1b

Then 'Create Launch Templates':

Launch Templates (1) Info

Search

Actions Create launch template

Launch Template ID	Launch Template Name	Default Version	Latest Version	Create Time	Created By
lt-0419f717f231b919b	masters-launch-template	6	7	2023-10-15T17:24:09.000Z	arn:aws:iam::096958155378:root

A name must be specified, then the 'Auto Scaling guidance' checkbox must be selected:

Launch template name and description

Launch template name - *required*

Mytemplate

Must be unique to this account. Max 128 chars. No spaces or special characters like '&', '*', '@'.

Template version description

A prod webserver for MyApp

Max 255 chars

Auto Scaling guidance [Info](#)

Select this if you intend to use this template with EC2 Auto Scaling

☒ Provide guidance to help me set up a template that I can use with EC2 Auto Scaling

Under the 'Launch template contents', go into 'My AMIs', and select the AMI created on step 7.

Launch template contents

Specify the details of your launch template below. Leaving a field blank will result in the field not being included in the launch template.

▼ Application and OS Images (Amazon Machine Image) - required [Info](#)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Search our full catalog including 1000s of application and OS images

Recents

My AMIs

Quick Start

Specify a custom value...

masters-image-backend

ami-049d22fe9ddb99f09

2023-10-15T17:11:42.000Z

Virtualization: hvm

ENA enabled: true

Root device type: ebs



masters-image-backend-enc3

ami-0a07869af9ac87d5d

2023-11-05T17:26:58.000Z

Virtualization: hvm

ENA enabled: true

Root device type: ebs

masters-image-backend

ami-049d22fe9ddb99f09

2023-10-15T17:11:42.000Z

Virtualization: hvm

ENA enabled: true

Root device type: ebs



Description

masters-image-backend

Architecture

x86_64

AMI ID

ami-049d22fe9ddb99f09

Under 'Network Settings', select the security group created previously, then under 'Advanced network configuration', select 'Auto-assign public IP' as 'Enable':

Common security groups [Info](#)

Select security groups ▼

launch-wizard-1 sg-0ae1681d9ec624056 ✕
VPC: vpc-0bf4b5d40a6786ebe

[Compare security group rules](#)

Security groups that you add or remove here will be added to or removed from all your network interfaces.

▼ **Advanced network configuration**

Network interface 1 Remove

<p>Device index Info</p> <p>0</p>	<p>Network interface Info</p> <p>New interface ▼</p> <p>Existing network interfaces are not recommended when creating a template for auto-scaling.</p>	<p>Description Info</p> <p></p>
<p>Subnet Info</p> <p>Don't include in launch template</p> <p>Not applicable for EC2 Auto Scaling</p>	<p>Security groups Info</p> <p>Select security groups ▼</p> <p>Show all selected (1)</p>	<p>Auto-assign public IP Info</p> <p>Enable ▼</p>

All the other fields can be left with the default values, except for one. Under ‘Advanced details’, the ‘User data’ field must be provided so that the java executable runs every time a new instance is created, restarted or rebooted.

The required script to be added is as follows:

```

1 Content-Type: multipart/mixed; boundary="//"
2 MIME-Version: 1.0
3
4 --//
5 Content-Type: text/cloud-config; charset="us-ascii"
6 MIME-Version: 1.0
7 Content-Transfer-Encoding: 7bit
8 Content-Disposition: attachment; filename="cloud-config.txt"
9
10 #cloud-config
11 cloud_final_modules:
12 - [scripts-user, always]
13
14 --//
15 Content-Type: text/x-shellscript; charset="us-ascii"
16 MIME-Version: 1.0
17 Content-Transfer-Encoding: 7bit
18 Content-Disposition: attachment; filename="userdata.txt"
19
20 #!/bin/bash
21 java -jar /home/ec2-user/backend-service-0.0.1-SNAPSHOT.jar &

```

It can be also found at: <https://github.com/brunorfranco/masterThesis/blob/main/user->

[data%20file.txt](#)

Once this is in place, then click the ‘Create launch template’ button:

Create launch template

Creating a launch template allows you to create a saved instance configuration that can be reused, shared and launched at a later time. Templates can have multiple versions.

Launch template name and description

Launch template name - *required*

Mytemplate

Must be unique to this account. Max 128 chars. No spaces or special characters like '&', '*', '@'.

Template version description

A prod webserver for MyApp

Max 255 chars

Auto Scaling guidance [Info](#)

Select this if you intend to use this template with EC2 Auto Scaling

☒ Provide guidance to help me set up a template that I can use with EC2 Auto Scaling

► Template tags

► Source template

Software Image (AMI)

masters-image-backend
ami-049d22fe9ddb99f09

Virtual server type (instance type)

-

Firewall (security group)

-

Storage (volumes)

1 volume(s) - 8 GiB

Free tier: In your first year includes 750 hours of t2.micro (or t3.micro in the Regions in which t2.micro is unavailable) instance usage on free tier AMIs per month, 30 GiB of EBS storage, 2 million I/Os, 1 GB of snapshots, and 100 GB of bandwidth to the internet.

Cancel

Create launch template

2.9 Security Group configuration

Go to the Security Group tab under the EC2 dashboard and make sure you include inbound and outbound rules for the port 8080, as well as 8082 into the newly created security group:

Security Groups (1/4) [Info](#)

[Actions](#) [Export security groups to CSV](#) [Create security group](#)

	Name	Security group ID	Security group name	VPC ID	Description	Owner
<input type="checkbox"/>	-	sg-bdb9bac7	default	vpc-23404744	default VPC security group	096958155378
<input checked="" type="checkbox"/>	-	sg-0ae1681d9ec624056	launch-wizard-1	vpc-0bf4b5d40a6786ebe	launch-wizard-1 created 2023-11-28T...	096958155378
<input type="checkbox"/>	-	sg-07daca764717242b8	default	vpc-0bf4b5d40a6786ebe	default VPC security group	096958155378
<input type="checkbox"/>	-	sg-0d4e045d9de7b1f8f	launch-wizard-2	vpc-0bf4b5d40a6786ebe	launch-wizard-2 created 2023-11-28T...	096958155378

Inbound rules (5)

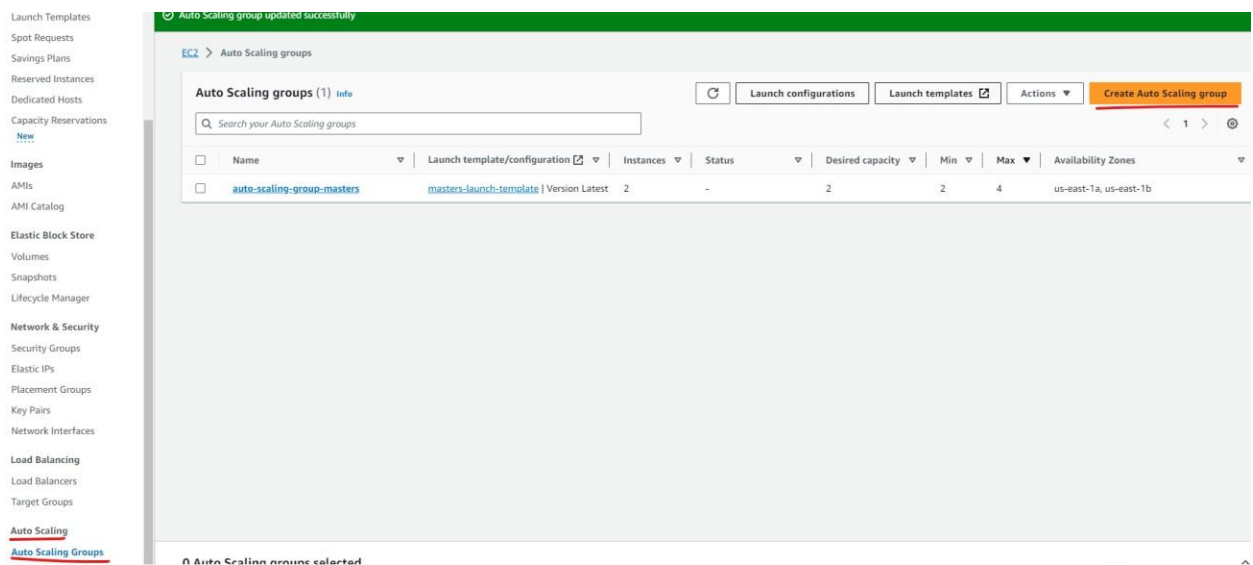
[Manage tags](#) [Edit inbound rules](#)

	Name	Security group rule...	IP version	Type	Protocol	Port range	Source	Description
<input type="checkbox"/>	-	sgr-052777261bcd9b...	IPv4	HTTPS	TCP	443	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-048b5623b83a58c...	IPv4	Custom TCP	TCP	8082	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-00f0ae84ed0580a94	IPv4	SSH	TCP	22	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-06eb0a7e1a6c9abc9	IPv4	HTTP	TCP	80	0.0.0.0/0	-
<input type="checkbox"/>	-	sgr-059e7ab9698e7fedc	IPv4	Custom TCP	TCP	8080	0.0.0.0/0	-

That is a very important step to allow for the Elastic Load balancer to be able to connect to the instances.

2.10 *Auto Scaling group and Elastic Load Balancer*

With the Launch template in place, then it was time to create the Auto Scaling group. That was achieved through the EC2 dashboard, 'Auto Scaling', 'Auto Scaling Groups', 'Create Auto Scaling group' button:



Once there, provide a name for the Auto scaling group, and select the Launch template defined on step 8:

Specify a launch template that contains settings common to all EC2 instances that are launched by this Auto Scaling group. If you currently use launch configurations, you might consider migrating to launch templates.

Name

Auto Scaling group name

Enter a name to identify the group.

Must be unique to this account in the current Region and no more than 255 characters.

Launch template [Info](#)

[Switch to launch configuration](#)

Launch template

Choose a launch template that contains the instance-level settings, such as the Amazon Machine Image (AMI), instance type, key pair, and security groups.



[Create a launch template](#)

Version



[Create a launch template version](#)

Description

master-launch-enc2

AMI ID

ami-0ce4bf6197472c257

Key pair name

master-thesis

Launch template

[masters-launch-template](#)
lt-0419f717f231b919b

Security groups

-

Security group IDs

[sg-05836dcfcd6c22680](#)

Instance type

t2.micro

Request Spot Instances

No

Tick the boxes for no minimum nor maximum vCPUs and Memory (GiB):

Required instance attributes

Enter your compute requirements in virtual CPUs (vCPUs) and memory.

vCPUs

Enter the minimum and maximum number of vCPUs per instance.

minimum maximum

☒ No minimum

☒ No maximum

Memory (GiB)

Enter the minimum and maximum GiBs of memory per instance.

minimum maximum

☒ No minimum

☒ No maximum

Click 'Next', and select two different Availability Zones and subnets, to ensure higher availability of the solution:

Instance type requirements [Info](#)

Override launch template

You can keep the same instance attributes or instance type from your launch template, or you can choose to override the launch template by specifying different instance attributes or manually adding instance types.

Launch template

[masters-launch-template](#) [lt-0419f717f231b919b](#)

Version

Default

Description

master-launch-enc2

Instance type

t2.micro

Network [Info](#)

☒ us-east-1a | subnet-c15be088
172.31.0.0/20 Default

☒ us-east-1b | subnet-5f3bf404
172.31.16.0/20 Default

☐ us-east-1c | subnet-d0c8a6b5
172.31.64.0/20 Default

☐ us-east-1d | subnet-b164ac9c
172.31.48.0/20 Default

☐ us-east-1e | subnet-53857d6f
172.31.32.0/20 Default

☐ us-east-1f | subnet-7a63ed76
172.31.80.0/20 Default

Select Availability Zones and subnets

▲

us-east-1a | subnet-c15be088 X
172.31.0.0/20 Default

us-east-1b | subnet-5f3bf404 X
172.31.16.0/20 Default

Instances and let EC2 Auto Scaling balance your instances across the group for getting started quickly.

↺

can use in the chosen VPC.

↺

After clicking ‘Next’, select ‘Attach a new load balancer’, then selected ‘Application Load Balancer (HTTP, HTTPS)’, selected a name for the load balancer, selected ‘Internet-facing:

134

Load balancing [Info](#)

Use the options below to attach your Auto Scaling group to an existing load balancer, or to a new load balancer that you define.

☐ No load balancer
Traffic to your Auto Scaling group will not be fronted by a load balancer.

☐ Attach to an existing load balancer
Choose from your existing load balancers.

☒ Attach to a new load balancer
Quickly create a basic load balancer to attach to your Auto Scaling group.

Attach to a new load balancer

Define a new load balancer to create for attachment to this Auto Scaling group.

Load balancer type

Choose from the load balancer types offered below. Type selection cannot be changed after the load balancer is created. If you need a different type of load balancer than those offered here, visit the [Load Balancing console](#).

☒ Application Load Balancer
HTTP, HTTPS

☐ Network Load Balancer
TCP, UDP, TLS

Load balancer name

Name cannot be changed after the load balancer is created.

Auto-scaling-group-name-1

Load balancer scheme

Scheme cannot be changed after the load balancer is created.

☐ Internal

☒ Internet-facing

Network mapping

Your new load balancer will be created using the same VPC and Availability Zone selections as your Auto Scaling group. You can select different subnets and add subnets from additional Availability Zones.

Under ‘Health checks’, select ‘Turn on Elastic Load Balancing health checks’ and left the health check grace period at the 300 seconds’ default.

Under ‘Listeners and routing, in the dropdown ‘Default routing (forward to)’, select ‘Create a target group:

Listeners and routing

If you require secure listeners, or multiple listeners, you can configure them from the [Load Balancing console](#) after your load balancer is created.

Protocol

Port

Default routing (forward to)

HTTP

80

Create a target group ▼

New target group name

An instance target group with default settings will be created.

interview-auto-scaling-group-1

135

Click 'Next' once again, then it is time to setup the Group sizing with two desired capacity, two minimum capacity and four maximum capacities:

Group size - optional [Info](#)

Specify the size of the Auto Scaling group by changing the desired capacity. You can also specify minimum and maximum capacity limits. Your desired capacity must be within the limit range.

Desired capacity

Minimum capacity

Maximum capacity

Review all the details provided and created the Auto Scaling group:

Scaling policy

No scaling policy

Instance scale-in protection

Instance scale-in protection
☐ Enable instance protection from scale in

Step 5: Add notifications

Edit

Notifications

No notifications

Step 6: Add tags

Edit

Tags (0)

Key	Value	Tag new instances
No tags		

Cancel

Previous

Create Auto Scaling group

With that in place, double check if the Auto scaling group and the load balancer were created appropriately and are healthy:

Load balancers (1)

Elastic Load Balancing scales your load balancer capacity automatically in response to changes in incoming traffic.

Filter load balancers

< 1 >

<input type="checkbox"/>	Name	DNS name	State	VPC ID	Availability Zones	Type	Date created
<input type="checkbox"/>	auto-scaling-group-m...	auto-scaling-group-master...	Active	vpc-23404744	2 Availability Zones	application	October 17, 2023, 19:01 (UTC+01:00)

Auto Scaling groups (1) Info

Launch configurations

Launch templates

Actions

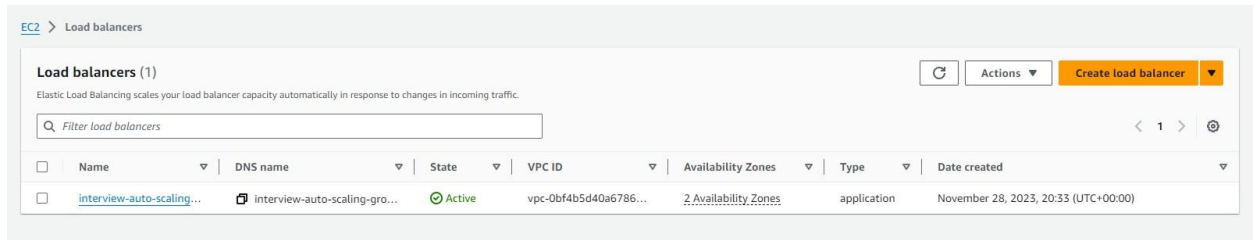
Create Auto Scaling group

Search your Auto Scaling groups

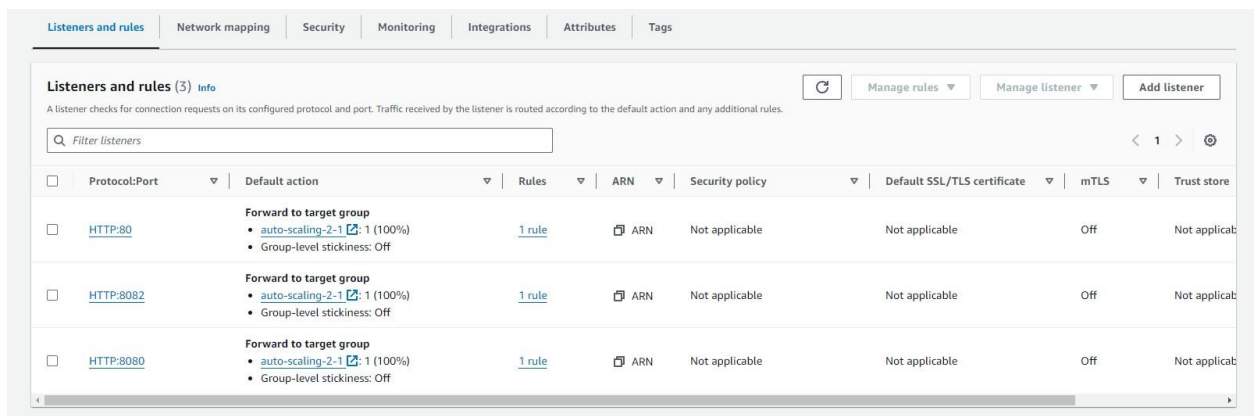
< 1 >

<input type="checkbox"/>	Name	Launch template/configuration	Instances	Status	Desired capacity	Min	Max	Availability Zones
<input type="checkbox"/>	auto-scaling-group-masters	masters-launch-template Version Latest	2	-	2	2	4	us-east-1a, us-east-1b

Once that is verified, get the Elastic Load Balancer DNS and access it by adding “/api/entries” and that should return a successful response:



Add Listeners and Rules to the newly created Elastic Load Balancer for port 80 and 8082 from the ‘Load Balancers’ tab:

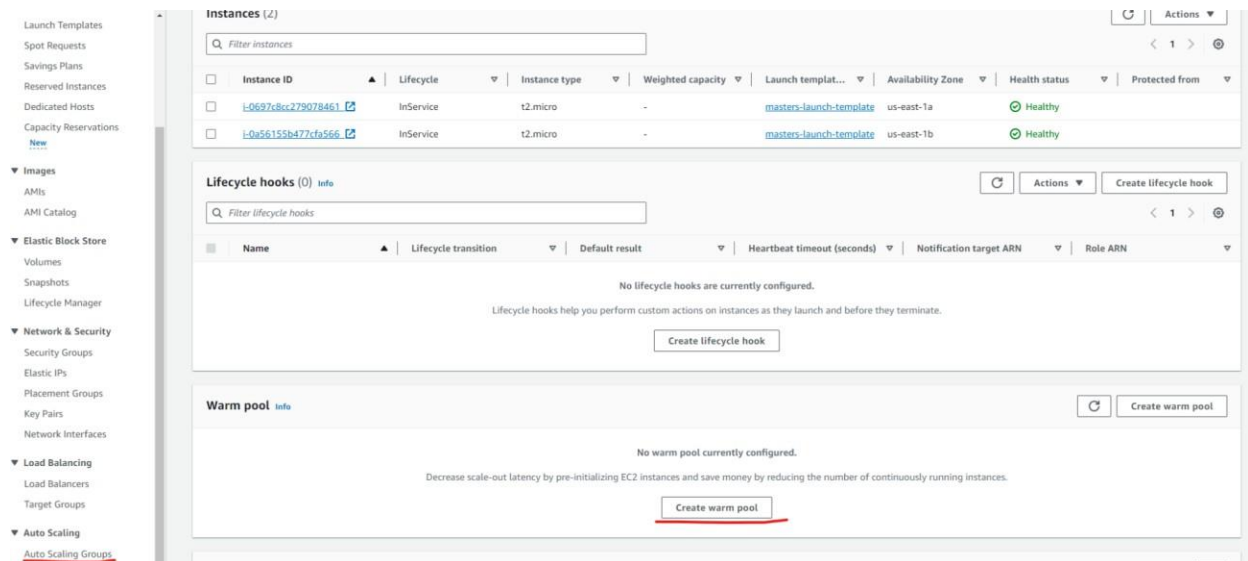


2.11 Warm Pooling

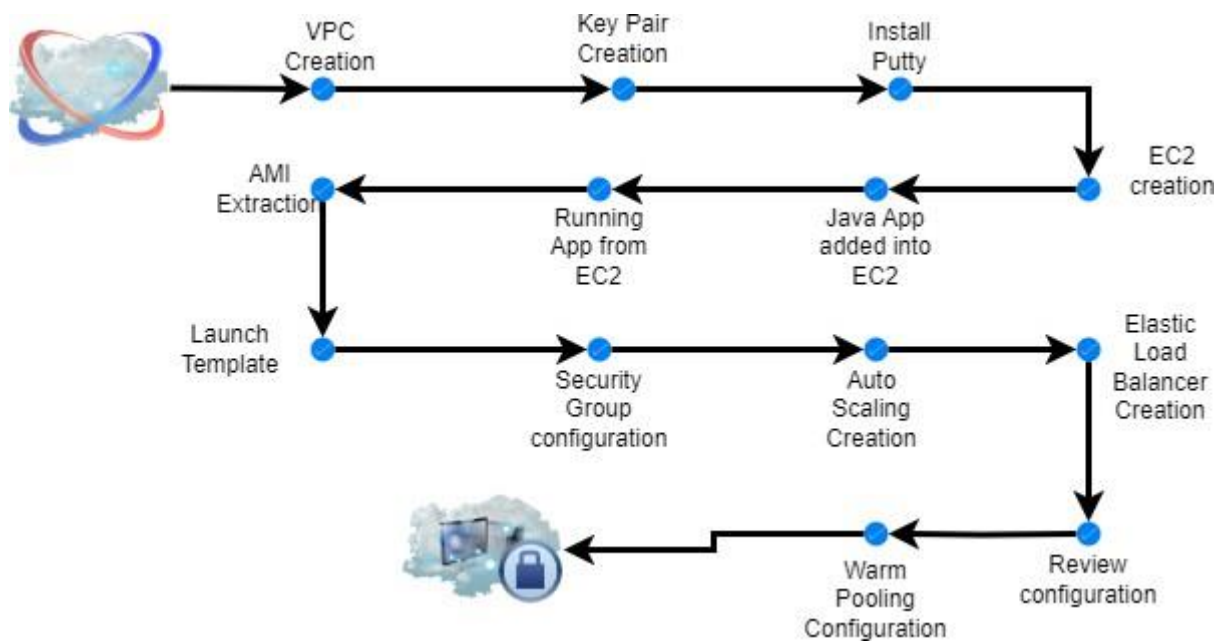
As mentioned previously, auto Scaling groups allow for different configuration variations, which have effects on recovery time of the service, so they will be explored as part of the experiment:

5. No warm pooling.
6. Warm pooling with one instance on ‘Stopped’ state.
7. Warm pooling with one instance on ‘Running’ state.
8. Warm pooling with one instance on ‘Hibernated’ state.

They can be configured under the ‘Auto Scaling Group’ section in the EC2 dashboard, under the ‘Instance management’ tab:

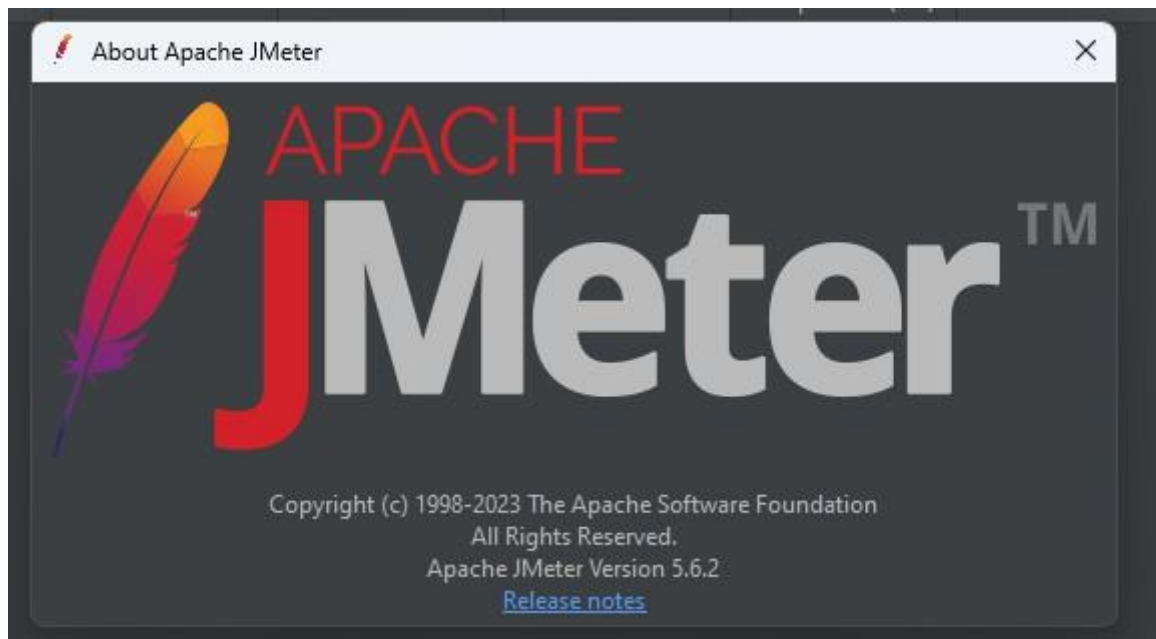


2.12 Setup Diagram



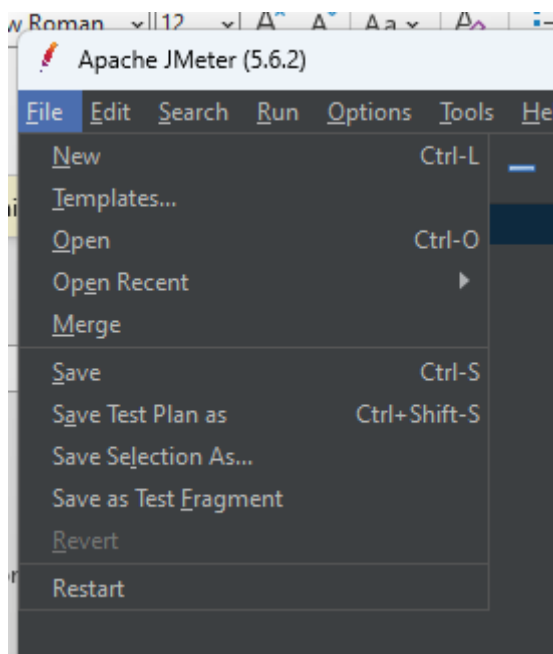
3. JMeter Test Plan Implementation

For this experiment, create a Test Plan in Apache JMeter version 5.6.2

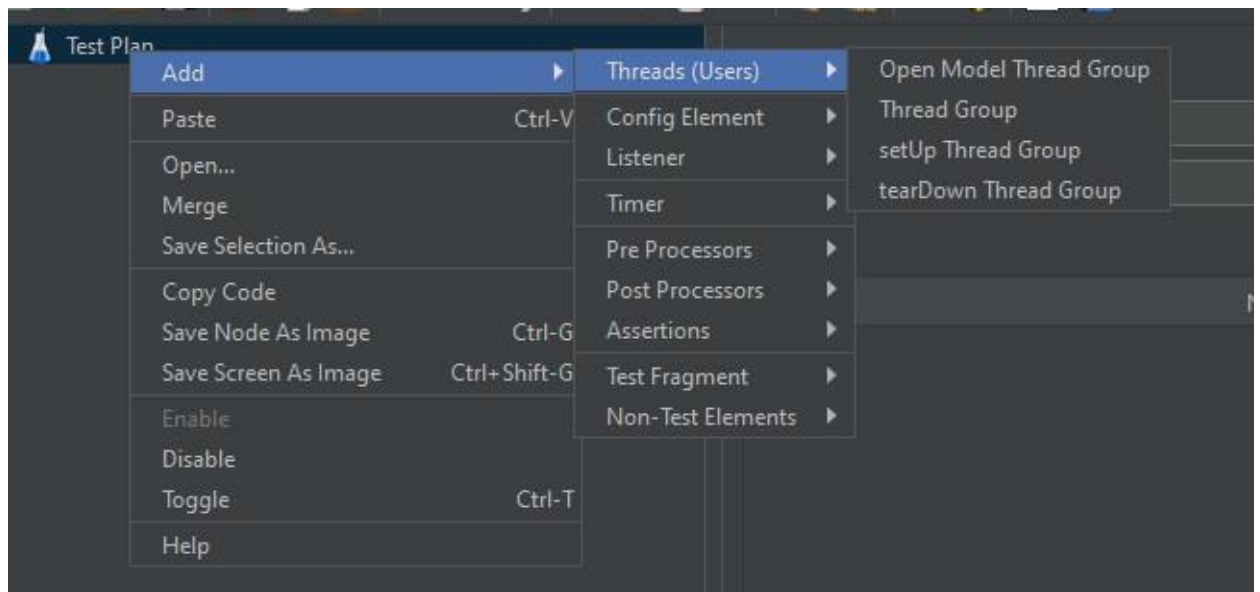


This tool is maintained by the Apache Software Foundation and is free and open source. At the time of this writing, the tool can be downloaded from https://jmeter.apache.org/download_jmeter.cgi

To create a test plan, click on 'File', then 'New':

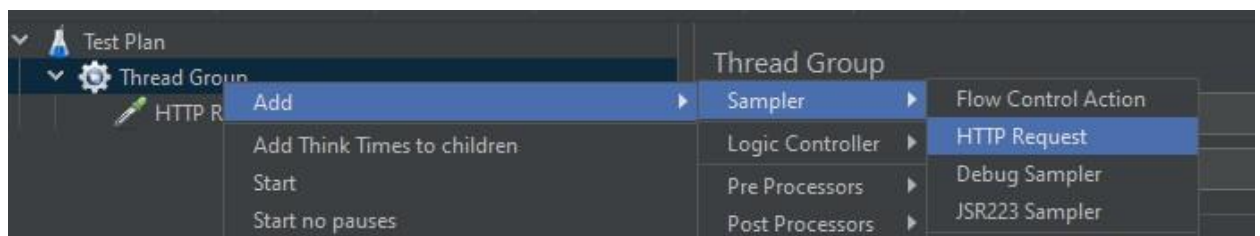


Then create a Thread Group by right-clicking the Test Plan, then going into 'Add', 'Threads (Users)', then selecting 'Thread Group':

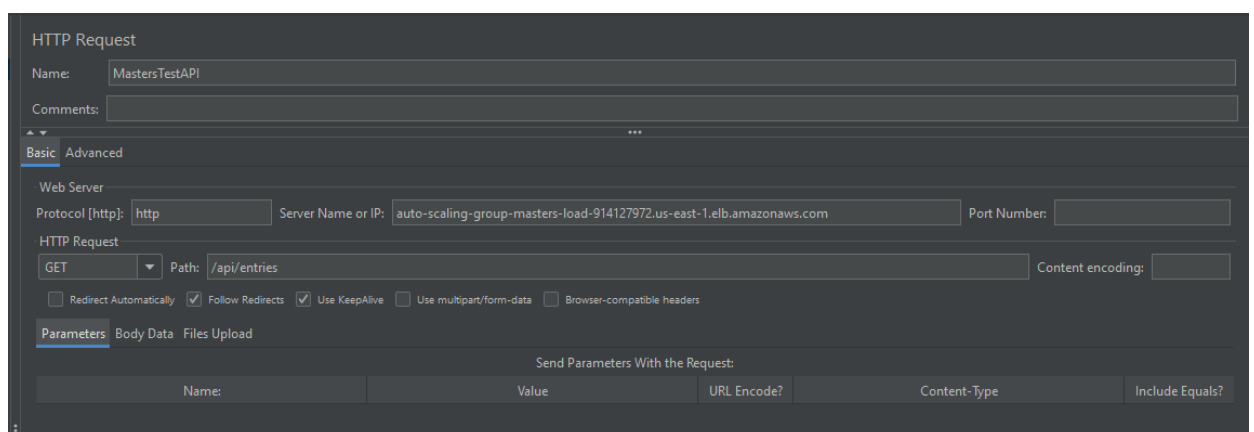


The default configuration for the Thread Group is enough for the purpose of this experiment, given that a single Thread will be used to check the health of the service running in AWS.

Add a HTTP Request into the Thread Group by right-clicking the Thread Group, then selecting 'Add', 'Sampler', 'HTTP Request':

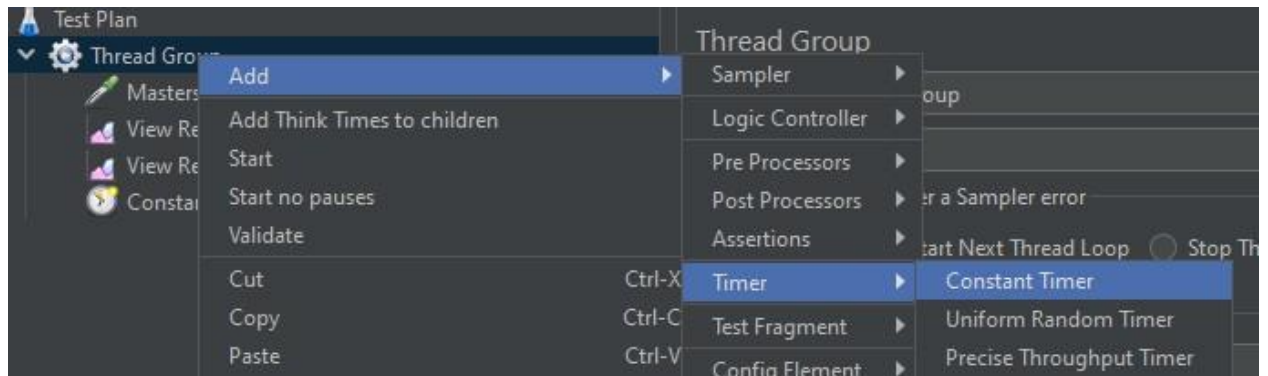


The HTTP Request needs to be configured to point to the Elastic Load Balancer address, therefore the Protocol, Server Name, HTTP Request and Path need to be configured as per the image below:

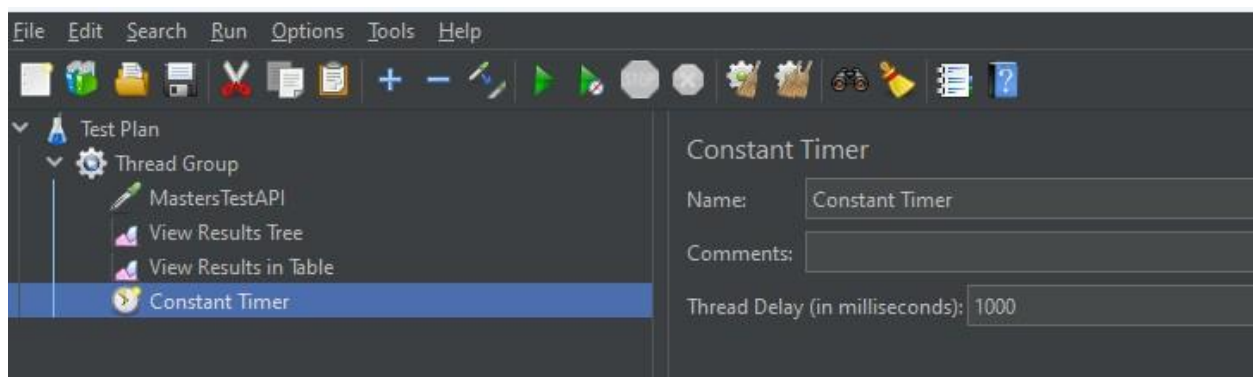


Given that by default JMeter's requests are fired every 300 milliseconds, that would

generate excessive entries in the result set, therefore configure a ‘Constant Timer’ so only a single request was fired per second, facilitating the result’s readings. That can be achieved by right-clicking on the Thread Group, then going into ‘Add’, ‘Timer’, ‘Constant Timer’:



The Constant Timer Thread Delay needs to be set to 1000 milliseconds:

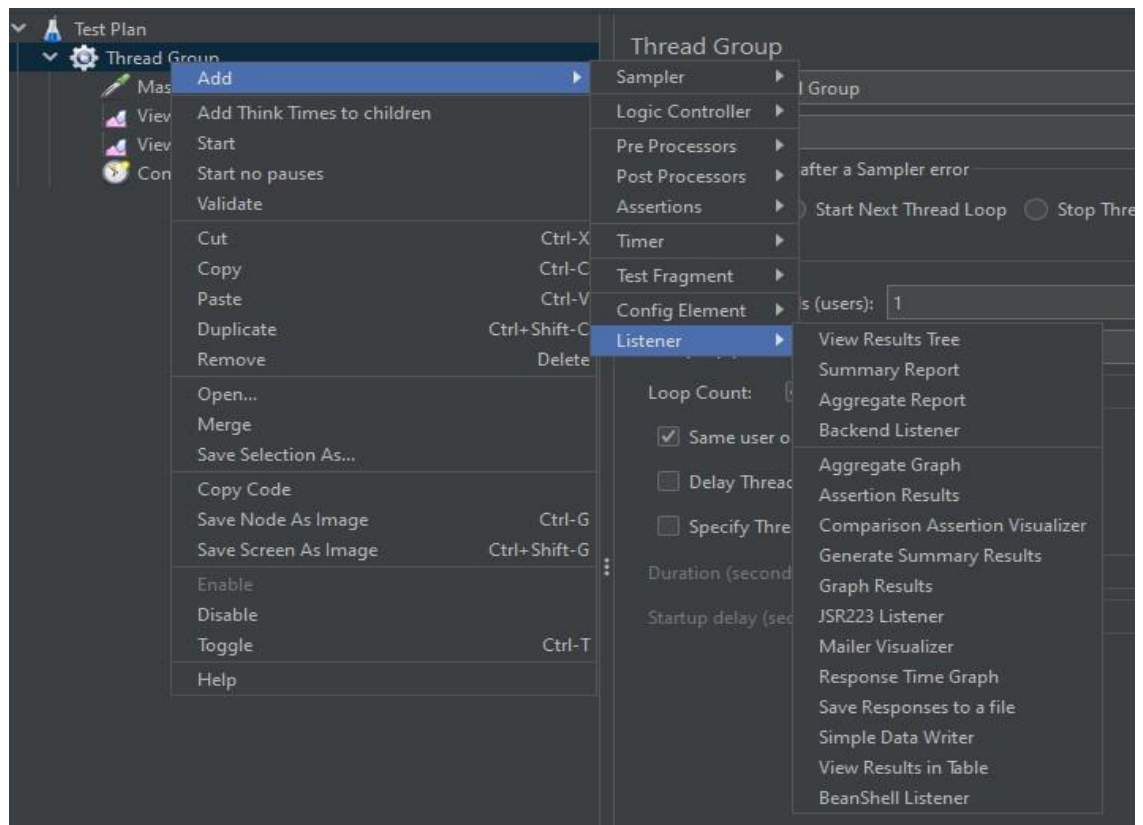


To view the results of the requests, two different listeners need to be created:

6.4.8.1 View Results Tree

6.4.8.2 View Results in Table

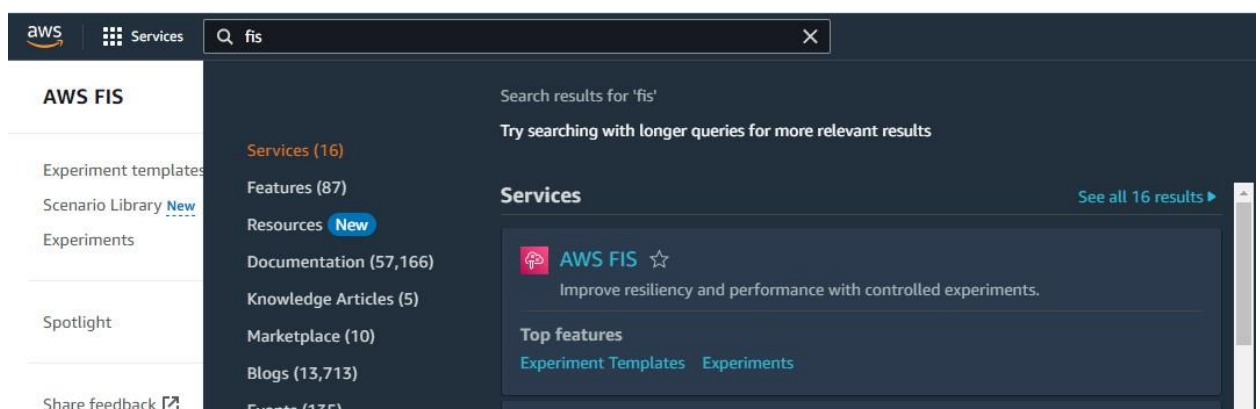
Listeners can be added by right-clicking ‘Thread Group’, then navigating to ‘Add’, then ‘Listeners’:



The full configuration can be found in .jmx format for ease of importing at <https://github.com/brunorfranco/masterThesis/blob/main/JMeterTests/APITesting.jmx>

4. AWS Fault Injection Simulator (FIS) Creation

The author has opted to utilize the AWS FIS as the chaos engineering tool of choice. The tool can be access through the AWS FIS console:



The first step is to create an experiment template:

Introducing Experiment Scheduler
We've added the ability to start FIS experiments at a preset time or on a recurrent schedule. [Learn more about Experiment Scheduler.](#)

AWS Fault Injection Simulator
Improve resiliency and performance with controlled experiments

Create experiment template
Choose your fault injection actions and the targets to run them on. Then start running your experiments.

[Create experiment template](#)

AWS Fault Injection Simulator is a fully managed service for running fault injection experiments on AWS that makes it easier to continuously improve an application's performance, observability, and resiliency.

Provide a Description, then add an action of 'Action type' of ALL, and 'aws:ec2:terminate-instances':

Actions (1) Info

▼ New action

Name
kill-instances
The name must have 1 to 64 characters.

Description - optional

The description must have 1 to 512 characters.

Action type [Info](#)
ALL ▼

Start after - optional [Info](#)
Select an action ▼

Add action

Search

CLOUDWATCH

- aws:cloudwatch:assert-alarm-state
Asserts that the CloudWatch alarms are in the expected states.

EBS

- aws:ebs:pause-volume-io
Pauses IO for a set of EBS volumes

EC2

- aws:ec2:reboot-instances
Reboot the specified EC2 instances.
- aws:ec2:send-spot-instance-interruptions
Interrupt the specified EC2 Spot instances.
- aws:ec2:stop-instances
Stop the specified EC2 instances.
- aws:ec2:terminate-instances**
Terminate the specified EC2 instances.

Remove

Targets also need to be setup, so click on 'Edit' under the 'Targets' section, keep the Resource type as 'aws:ec2:instance', also keep the Target method as 'Resource IDs', then select the Resource IDs of the EC2 instances currently running:

Edit target

×

Specify the target resources on which to run your selected actions. [Learn more](#)

Name

Instances-Target-1

The name must have 1 to 64 characters.

Resource type

aws:ec2:instance

Actions

kill-instances

Target method

☒ Resource IDs

☐ Resource tags, filters and parameters

Resource IDs

Select a resource ID

Q |

☐ i-0697c8cc279078461

☐ i-0a56155b477cfa566

Cancel

Save

The Resource IDs can be double checked by going back to the EC2 console, then accessing the Instances (running):

Instances (2) Info					
<input type="text" value="Find Instance by attribute or tag (case-sensitive)"/>					
<input type="text" value="Instance state = running"/> <input type="button" value="X"/>		<input type="button" value="Clear filters"/>			
<input type="checkbox"/>	Name ✎ ▼	Instance ID	Instance state ▼	Instance type ▼	Status check
<input type="checkbox"/>		i-0697c8cc279078461	<input checked="" type="checkbox"/> Running <input type="button" value="🔍"/> <input type="button" value="🔍"/>	t2.micro	<input checked="" type="checkbox"/> 2/2 checks passed
<input type="checkbox"/>		i-0a56155b477cfa566	<input checked="" type="checkbox"/> Running <input type="button" value="🔍"/> <input type="button" value="🔍"/>	t2.micro	<input checked="" type="checkbox"/> 2/2 checks passed

Select the ‘Create a new role for the experiment template’ option under ‘Service access’, so an IAM role can be automatically created with correct permissions to conduct the experiments:

Service access

FIS requires permission to conduct experiments on your behalf

☒ Create a new role for the experiment template
☐ Use an existing IAM role

Service role name

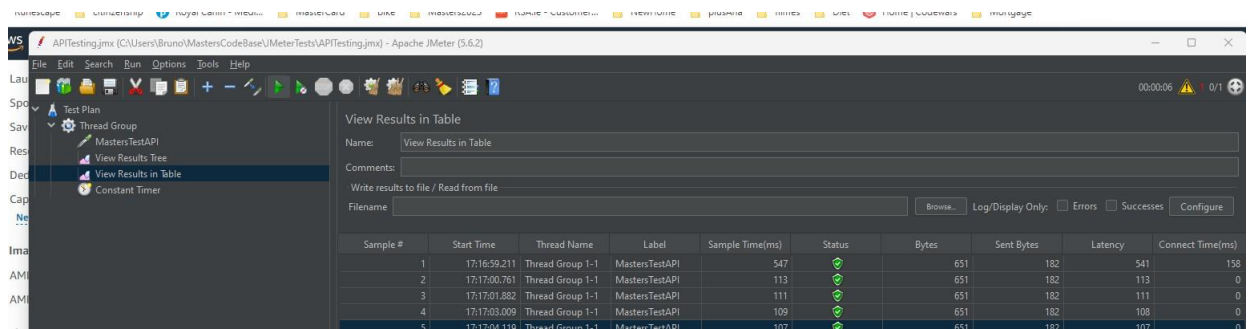
Click on ‘Create experiment template’.

The command line configuration can be found at the following link to facilitate the creation of the FIS experiment:
<https://github.com/brunorfranco/masterThesis/tree/main/FISExperimentTemplate>

5. Experiment Execution

Ensure that before any of the experiment executions there are two healthy EC2 instances running and serving requests through the Load Balancer.


Initiate a timer. As soon as the timer reaches thirty seconds, initiate the JMeter Test Plan to start collecting metrics:



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	17:16:59.211	Thread Group 1-1	MastersTestAPI	547	Success	651	182	541	158
2	17:17:00.761	Thread Group 1-1	MastersTestAPI	113	Success	651	182	113	0
3	17:17:01.882	Thread Group 1-1	MastersTestAPI	111	Success	651	182	111	0
4	17:17:03.009	Thread Group 1-1	MastersTestAPI	109	Success	651	182	108	0
5	17:17:04.119	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0

Once the timer reaches one minute, initiate the FIS experiment to shut-down running EC2 instances:

Start experiment

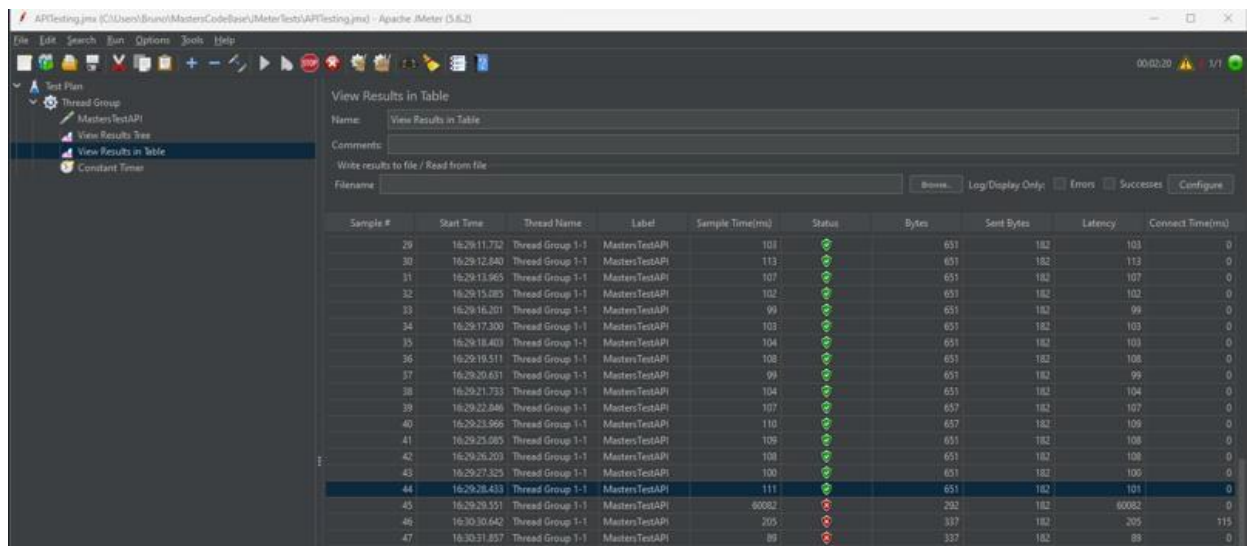


You are about to start your experiment, which might perform destructive actions on your AWS resources. Before you run fault injection experiments, review the best practices and planning guidelines. [Learn more](#)

To confirm that you want to start the experiment, enter *start* in the field:

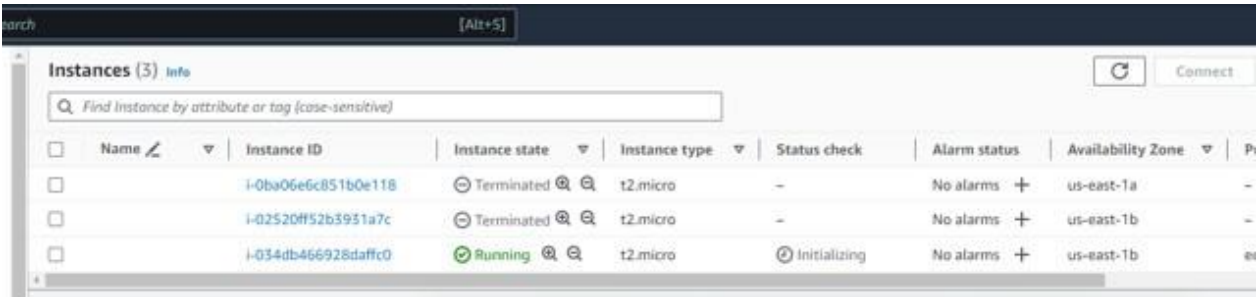
Cancel
Start experiment

The JMeter Test Plan would soon (~15 minutes after the FIS startup) indicate that the requests were no longer responding successfully, so take note up to the millisecond of the last time a request was successful before starting to fail, via the ‘View Result in Table’ in JMeter.



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
29	16:29:11.752	Thread Group 1-1	MastersTestAPI	103	Success	651	182	103	0
30	16:29:12.840	Thread Group 1-1	MastersTestAPI	113	Success	651	182	113	0
31	16:29:13.965	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0
32	16:29:15.085	Thread Group 1-1	MastersTestAPI	102	Success	651	182	102	0
33	16:29:16.301	Thread Group 1-1	MastersTestAPI	99	Success	651	182	99	0
34	16:29:17.300	Thread Group 1-1	MastersTestAPI	103	Success	651	182	103	0
35	16:29:18.403	Thread Group 1-1	MastersTestAPI	104	Success	651	182	103	0
36	16:29:19.511	Thread Group 1-1	MastersTestAPI	108	Success	651	182	108	0
37	16:29:20.631	Thread Group 1-1	MastersTestAPI	99	Success	651	182	99	0
38	16:29:21.733	Thread Group 1-1	MastersTestAPI	104	Success	651	182	104	0
39	16:29:22.846	Thread Group 1-1	MastersTestAPI	107	Success	651	182	107	0
40	16:29:23.965	Thread Group 1-1	MastersTestAPI	110	Success	651	182	109	0
41	16:29:25.085	Thread Group 1-1	MastersTestAPI	109	Success	651	182	108	0
42	16:29:26.203	Thread Group 1-1	MastersTestAPI	108	Success	651	182	108	0
43	16:29:27.325	Thread Group 1-1	MastersTestAPI	100	Success	651	182	106	0
44	16:29:28.433	Thread Group 1-1	MastersTestAPI	111	Success	651	182	101	0
45	16:29:29.551	Thread Group 1-1	MastersTestAPI	60082	Failure	292	182	60082	0
46	16:30:30.642	Thread Group 1-1	MastersTestAPI	205	Failure	337	182	205	115
47	16:30:31.857	Thread Group 1-1	MastersTestAPI	89	Failure	337	182	89	0

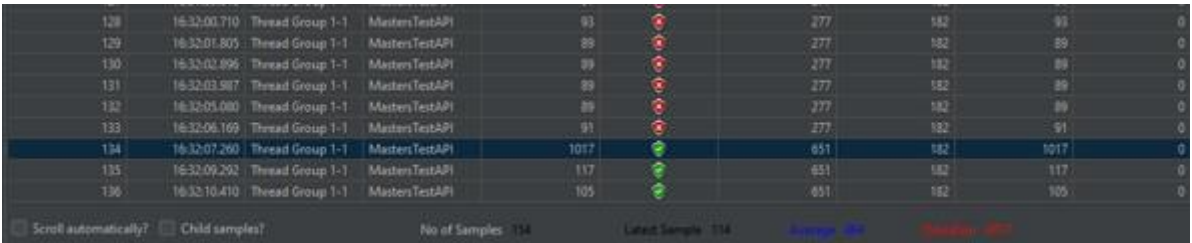
AWS auto scaling will realize that it does not have the minimum required number of instances as part of its group, so it will spin up a healthy instance, then subsequently a second one separately (as a mechanism to avoid spinning up extra unnecessary instances).



The screenshot shows the AWS Management Console 'Instances' page. It displays a list of three EC2 instances. The first two are in a 'Terminated' state, and the third is in a 'Running' state. The 'Running' instance is highlighted with a green checkmark in the status column.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
	i-0ba06e6c851b0e118	Terminated	t2.micro	-	No alarms	us-east-1a
	i-02520ff52b3931a7c	Terminated	t2.micro	-	No alarms	us-east-1b
	i-034db466928daffc0	Running	t2.micro	Initializing	No alarms	us-east-1b

As soon as the new instances are assigned to the Load Balancer and the JMeter Test Plan will stop failing and start receiving successful responses back again, then take note of the time up to milliseconds of the first successful response after the fault injection:



The screenshot shows a JMeter test results table. It displays a list of test samples with columns for ID, Time, Thread Group, Test Plan, Success, Latency, and Error. The table shows a transition from failed requests (red 'x' icons) to successful requests (green checkmark icons) around sample 134.

ID	Time	Thread Group	Test Plan	Success	Latency	Error
128	16:32:00.710	Thread Group 1-1	MastersTestAPI	93	277	182
129	16:32:01.805	Thread Group 1-1	MastersTestAPI	89	277	182
130	16:32:02.896	Thread Group 1-1	MastersTestAPI	89	277	182
131	16:32:03.987	Thread Group 1-1	MastersTestAPI	89	277	182
132	16:32:05.080	Thread Group 1-1	MastersTestAPI	89	277	182
133	16:32:06.169	Thread Group 1-1	MastersTestAPI	91	277	182
134	16:32:07.260	Thread Group 1-1	MastersTestAPI	1017	651	1017
135	16:32:09.292	Thread Group 1-1	MastersTestAPI	117	651	117
136	16:32:10.410	Thread Group 1-1	MastersTestAPI	105	651	105

With both entries noted, the simple following subtraction can be used to calculate the time it takes for AWS to self-heal its system:

Time of the first successful request after fault injection – time of the last successful request before fault injection

APPENDIX C: INTERVIEW PRESENTATION





AGENDA

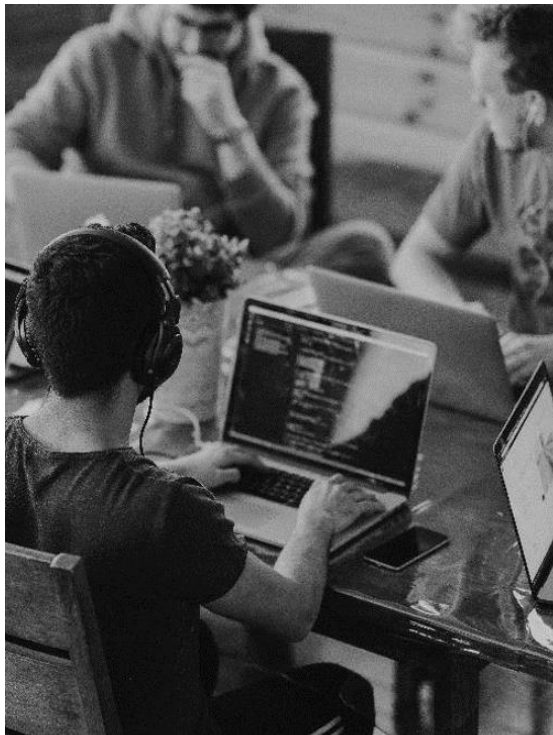
INTRODUCTION

RESEARCH QUESTION

EXPERIMENT

RESULTS

YOUR CONCLUSIONS!

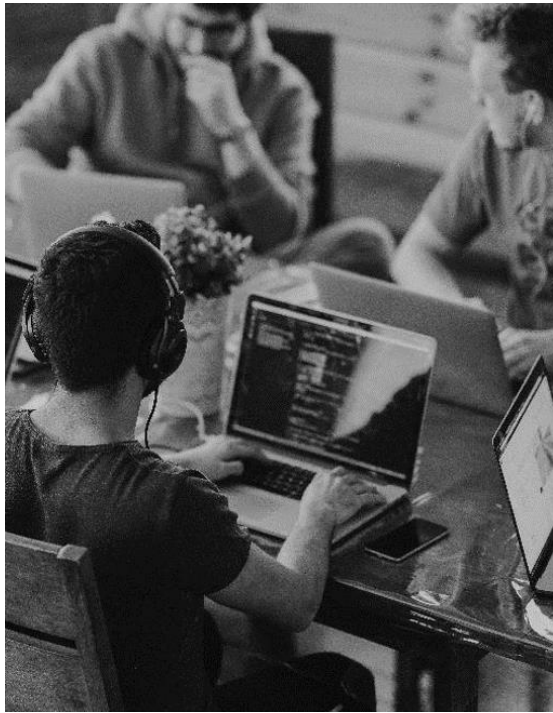


INTRODUCTION

MICROSERVICES

An architectural approach that organizes an application into a set of services characterized by the following attributes:

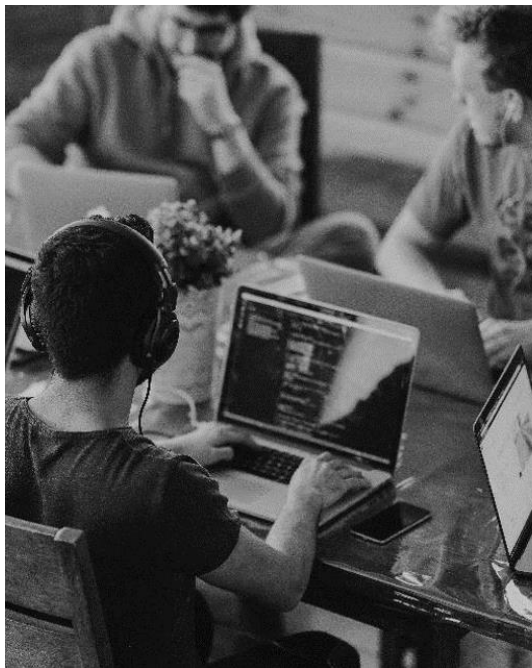
1. Capable of independent deployment.
2. Loosely interconnected.
3. Aligned with specific business functionalities.
4. Managed by small, dedicated teams.



INTRODUCTION

CLOUD INFRASTRUCTURE

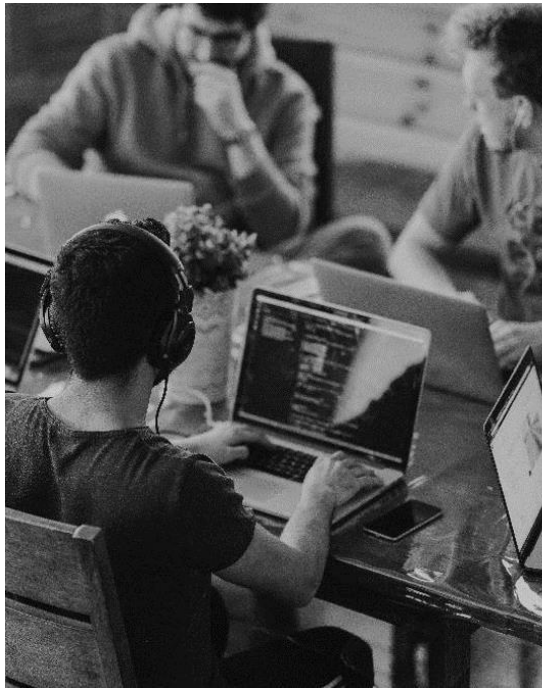
Cloud computing is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing.



INTRODUCTION

SELF-HEALING SYSTEMS

Self-healing systems perform periodic health assessments on various components and autonomously initiate corrective actions, such as redeployment, to restore them to their intended operational conditions (Ghosh, *et al.*, 2007).



INTRODUCTION

CHAOS ENGINEERING TECHNIQUES

Involves conducting systematic experiments on a system to instill confidence in its capacity to endure challenging operational conditions (Rosenthal and Jones, 2020). In software development, it is common to specify the need for a software system to gracefully handle failures while maintaining an acceptable level of service quality

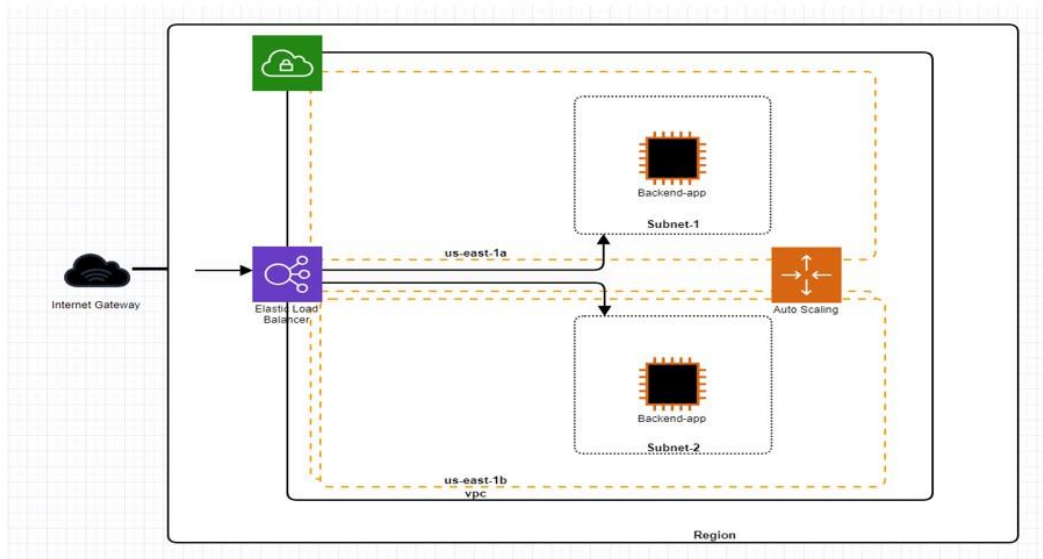


RESEARCH QUESTION

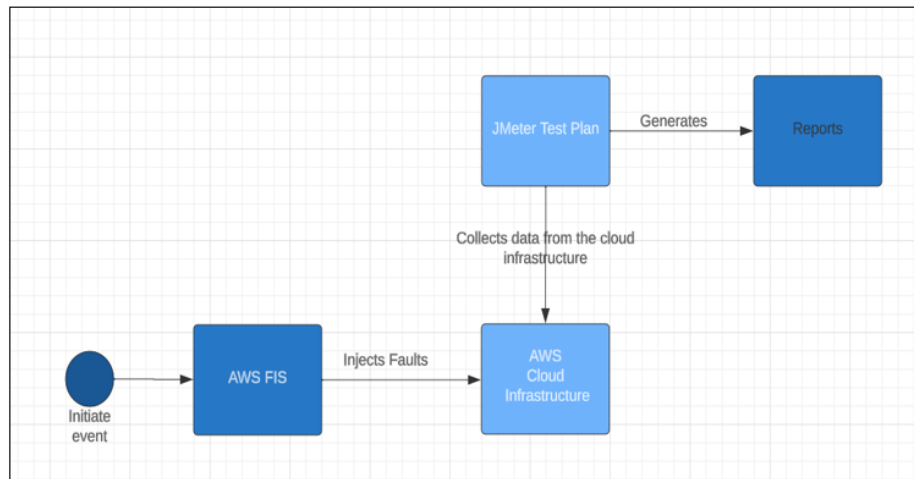
LET'S DIVE IN

Can Chaos Engineering techniques implemented via AWS Fault Injector Simulator (FIS) degrade an 'Industry standard Self-healing Cloud-based microservice architecture' beyond a one-minute response-time SLO?

AWS ARCHITECTURE

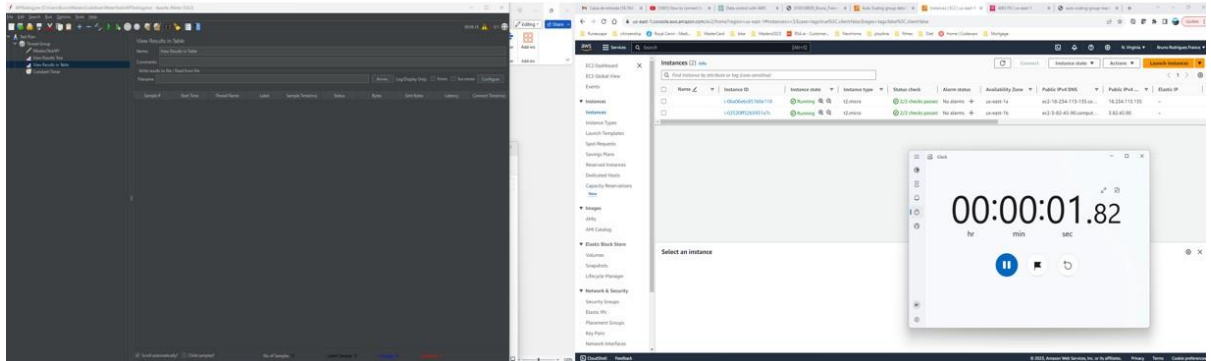


EXPERIMENT DESIGN



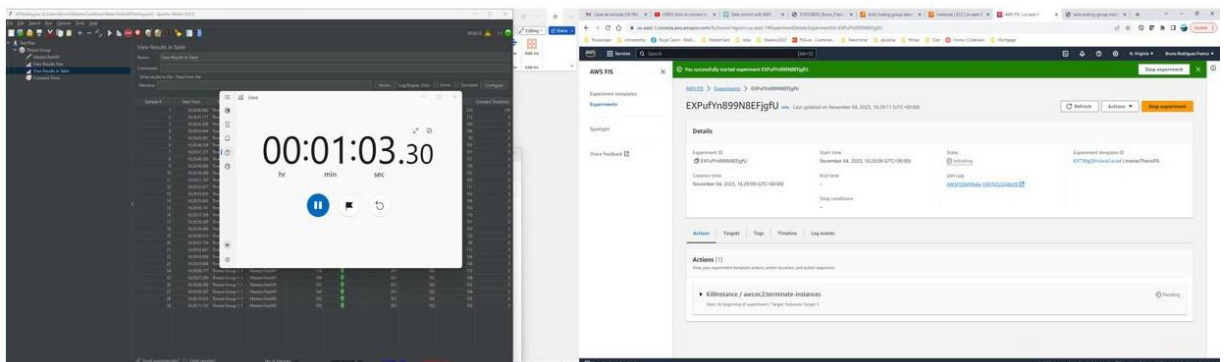
SAMPLE

Step 1



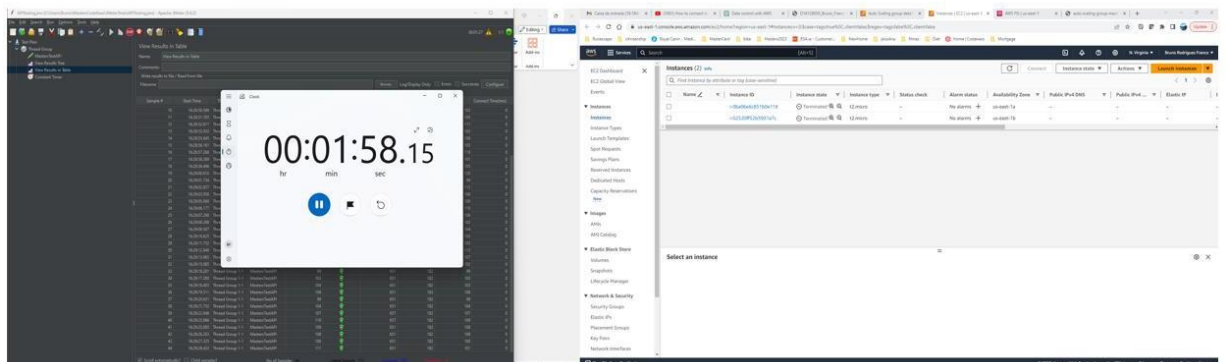
SAMPLE

Step 2



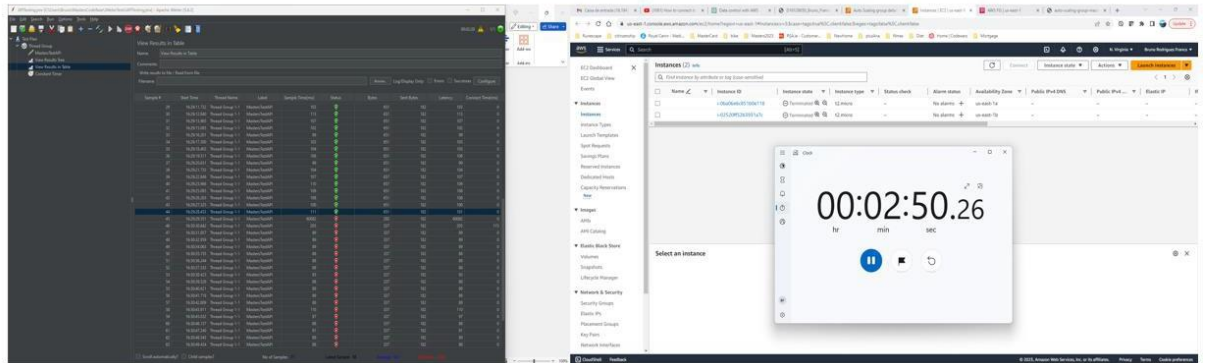
SAMPLE

Step 3



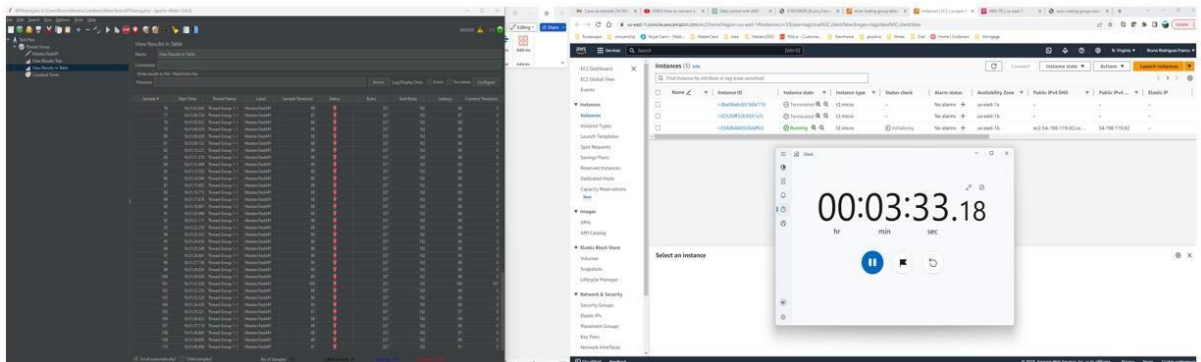
SAMPLE

Step 4



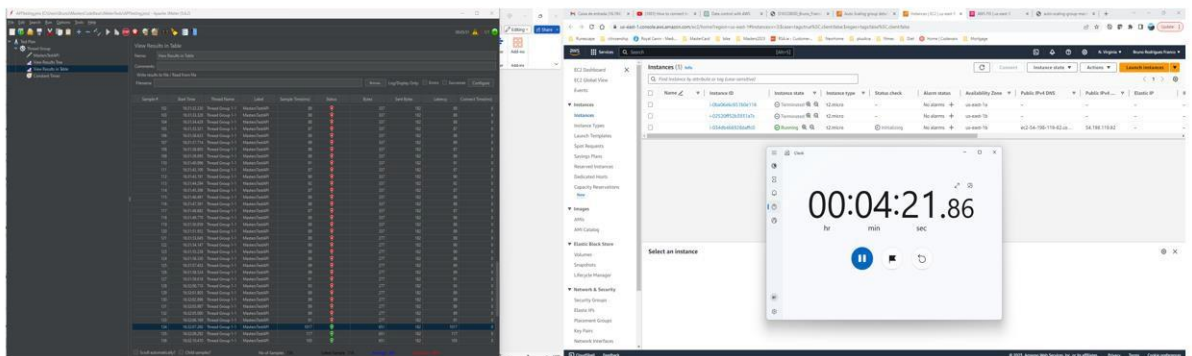
SAMPLE

Step 5



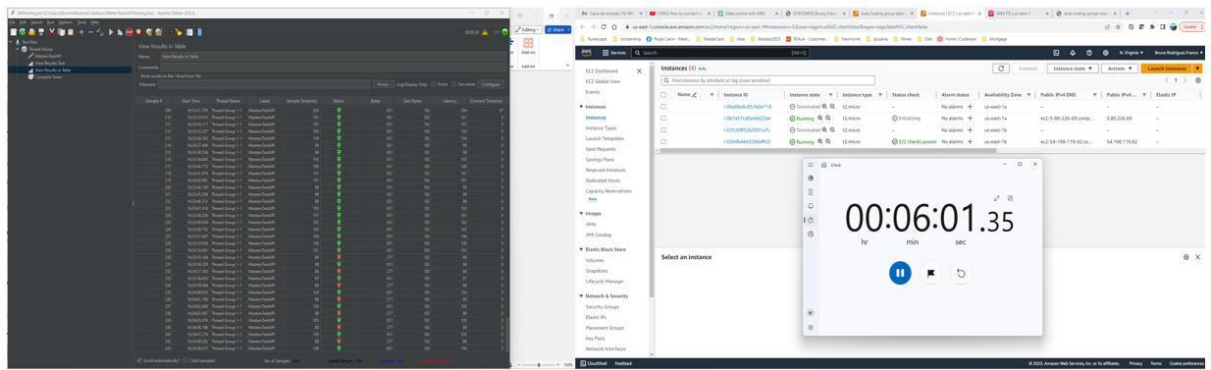
SAMPLE

Step 6



SAMPLE

Step 7

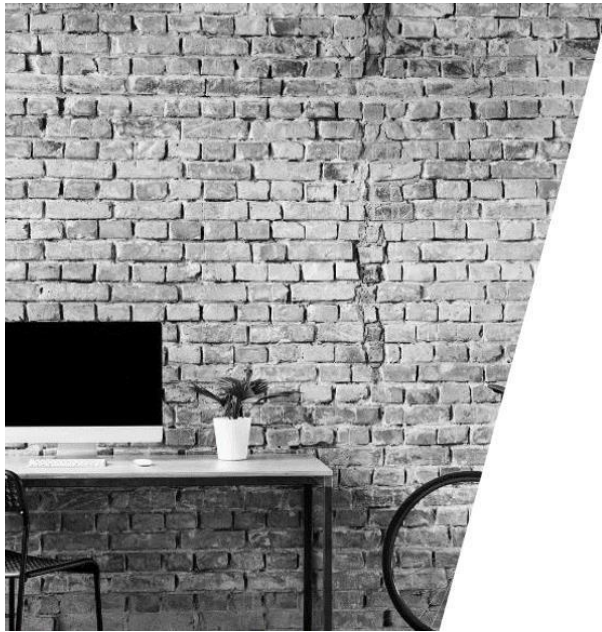


AWS CONFIGURATION VARIATION

WARM POOLING

The experiment was executed in four different phases (differentiated by auto-scaling configurations), eight times each as well:

1. No warm pooling.
2. Warm pooling with one instance on 'Stopped' state.
3. Warm pooling with one instance on 'Running' state.
4. Warm pooling with one instance on 'Hibernated' state.

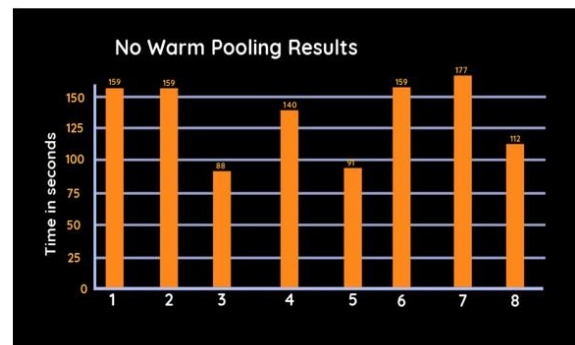


RESULTS

INCOMING DIAGRAMS!

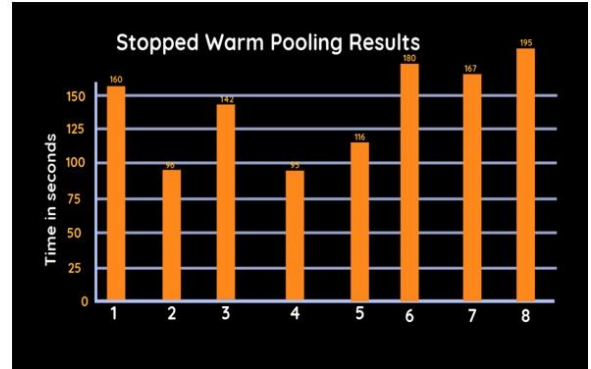
NO WARM POOLING

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	16:29:28.433	16:32:07.260	02:38.827
2	16:37:21.088	16:39:59.916	02:38.828
3	16:44:46.607	16:46:14.237	01:27.630
4	16:51:41.303	16:54:01.630	02:20.327
5	16:58:59.813	17:00:30.580	01:30.767
6	17:05:48.506	17:08:27.393	02:38.887
7	17:13:14.345	17:16:11.756	02:57.411
8	17:20:24.474	17:22:16.195	01:51.721
Average			02:15.549



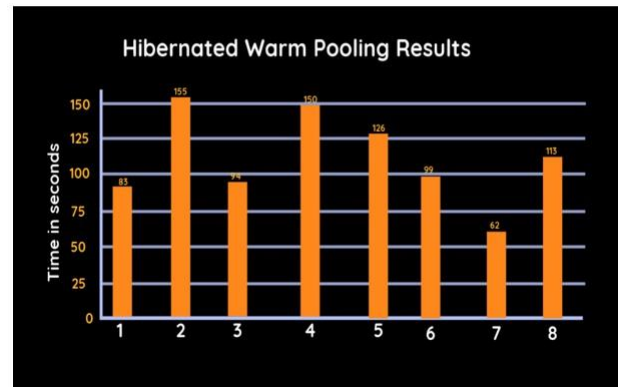
STOPPED WARM POOLING

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	12:21:14.040	12:23:53.779	02:39.739
2	14:18:39.652	14:20:16.130	01:36.47
3	14:27:41.048	14:30:02.658	02:21.610
4	14:36:24.950	14:38:00.258	01:35.308
5	14:44:04.283	14:46:00.484	01:56.201
6	14:51:21.225	14:54:21.301	03:00.076
7	15:01:16.763	15:04:03.782	02:47.019
8	15:09:03.961	15:12:19.420	03:15.459
Average			02:23.986



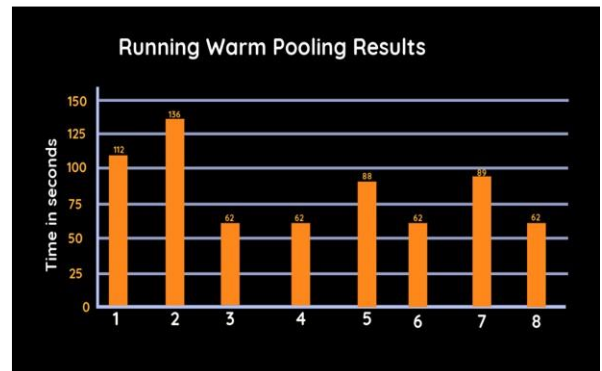
HIBERNATED WARM POOLING

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	18:08:25.344	18:09:48.567	01:23.223
2	18:15:08.020	18:17:42.913	02:34.893
3	18:22:12.935	18:23:47.238	01:34.303
4	18:29:12.944	18:31:43.360	02:30.416
5	18:39:44.713	18:41:51.029	02:06.316
6	18:46:36.257	18:48:14.913	01:38.656
7	18:54:54.811	18:55:57.026	01:02.215
8	19:01:58.829	19:03:51.875	01:53.046
Average			01:50.383



RUNNING WARM POOLING

No.	Last successful request after fault	First successful request after recovery	Elapsed time
1	17:37:44.065	17:39:35.936	01:51.871
2	17:45:18.922	17:47:34.990	02:16.068
3	17:52:46.518	17:53:48.744	01:02.226
4	18:02:41.585	18:03:43.793	01:02.208
5	18:10:01.389	18:11:29.012	01:27.62
6	18:16:43.552	18:17:45.765	01:02.213
7	18:30:05.661	18:31:34.408	01:28.747
8	18:37:05.916	18:38:08.136	01:02.220
Average			01:24.147



RECOVERY TIME PER POOL

1. No warm pooling recovery time: **02:15.549**
2. Stopped warm pooling recovery time: **02:23.986**
3. Hibernated warm pooling recovery time: **01:50.383**
4. Running warm pooling recovery time: **01:24.147**



RECOVERY TIME COMPARISON

	<i>No WP</i>	<i>Stopped</i>	<i>Hibernated</i>	<i>Running</i>
<i>No WP</i>		+5.88%	-23.64%	-61.9%
<i>Stopped</i>			-30.91%	-71.43%
<i>Hibernated</i>				-30.95%
<i>Running</i>				



CONCLUSIONS:

WARM-UP QUESTIONS:

How many years experience in Software Architecture
Business Domains
Cloud Domains

CHOICE OF TOOLS

Would you have chosen the tools/platforms differently?
AWS x FIS x JMeter

EXPERIMENT VALIDATION

Are you satisfied with how the experiment was conducted?
Would you have done it differently?

RESULTS

Are the results as you have expected?
Which warm pooling configuration would you recommend?
What can you conclude from the experiment results?



THANK YOU

I APPRECIATE YOUR TIME!