POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

AI-Driven Unit Test Generation

Supervisors
Prof. Riccardo Coppola
Massimo Pacileo

Candidate
Francesco Pio Cellamare

Academic Year 2023-2024

Abstract

In the context of cloud-based software development on Amazon Web Services (AWS), adopting DevOps practices through services like AWS CodePipeline is essential for ensuring Continuous Integration (CI) and Continuous Delivery (CD) while maintaining high software quality standards. However, integrating automated testing within these pipelines can be complex, particularly when aiming to generate comprehensive, high-quality tests with good overall coverage.

To address this challenge, this study proposes leveraging AI-driven test generation techniques within DevOps pipelines on AWS CodePipeline, followed by a manual approval phase for the generated tests.

The primary objective of this research is to explore the use of artificial intelligence (AI) for automated test generation to optimize DevOps practices, enhance software quality, and accelerate release cycles. The study focuses on designing a pipeline within AWS CodePipeline, incorporating all key Software Development Life Cycle (SDLC) stages (source, build, test, and deploy), while identifying automated test generation as a crucial area for AI integration. The required infrastructure will be provisioned using AWS CloudFormation, an Infrastructure as Code (IaC) service.

The experimentation will primarily target microservices architectures (Java, Spring Boot), which will serve as the foundational source for this thesis. The proposed approach involves integrating AI-powered tools such as OpenAI or custom solutions into the pipeline architecture to generate automated tests. These generated tests will then undergo an approval phase to assess their quality, relevance, and adequacy, ensuring a qualitative control over the generated test suites.

The results of this research aim to demonstrate the potential of AI-driven test generation techniques in improving software quality, development productivity, and the release process of new components, while fully embracing DevOps best practices.

Contents

Chapter 1

Introduction

Software testing is fundamental to ensure high-quality product to the end user. Through testing, developers strive to achieve a certain level of 'correctness' in the system under development to achieve a higher satisfaction with the end users. To ensure complete correctness, it is essential to test and verify all possible inputs and outputs for our application. However, achieving complete correctness in a real-world software application is not feasible. Even moderately complex software introduces challenges that are difficult to manage, making it impossible to account for every potential error or system failure.

Moreover, studies show that this phase of the Software Development Lifecycle is expensive in terms of costs due to the fact that developers must plan an effective test suite not focusing on developing the main business logic. The cost is primarily due to the time-intensive nature of the testing phase, which must continue throughout the development process. Detecting a defect in the early phases of development helps reduce costs by reducing the effort and time required to resolve it later when the product is launched. In order to improve testing efficiency, reduce the time needed to identify bugs, and enhance team collaboration, DevOps practices are increasingly adopted to streamline the process.

In modern cloud-based software development on Amazon Web Services (AWS), adopting DevOps practices through services like AWS CodePipeline is crucial to enable Continuous Integration (CI) and Continuous Delivery (CD) while maintaining high software quality. However, integrating automated testing within these pipelines presents challenges, particularly in generating comprehensive, high-quality tests with sufficient coverage.

This study explores the integration of AI-driven test generation techniques within AWS CodePipeline, followed by a manual approval phase to validate the generated

tests. The primary objective is to leverage Artificial Intelligence (AI) to optimize DevOps workflows, improve software quality, and accelerate release cycles. The research focuses on designing a fully automated pipeline within AWS CodePipeline, encompassing all key stages of Software Development Life Cycle (SDLC) including source, build, test, and deploy, while emphasizing AI-driven test generation as a critical enhancement. Infrastructure provisioning will be managed using AWS CloudFormation, adhering to Infrastructure as Code (IaC) principles.

We will incorporate Generative AI (GenAI) into the process, ensuring strict interaction with developers. This approach is not only about automating test generation, but also about providing developers with the ability to review and decide whether the generated test suites are comprehensive, effective, and aligned with the software requirements. By maintaining this balance between automation and manual oversight, the goal is to ensure that the tests are relevant and high-quality, while still benefiting from the efficiency that AI can offer.

The experimentation will primarily target microservices architectures based on Java and Spring Boot, serving as the core application environment for this study. The proposed approach involves integrating AI-powered tools, such as OpenAI models or custom solutions, to automatically generate tests within the pipeline. These generated tests will then undergo a manual approval phase to evaluate their quality, relevance and effectiveness, ensuring robust validation before deployment.

For this study, we focus on unit testing as it provides a fundamental layer of testing that is essential for verifying the functionality of individual components in the system. Unit tests are typically fast to execute, easy to automate and help detect issues early in the development process, making them ideal for integration into an automated pipeline. Using unit testing, we ensure that each microservice component functions correctly in isolation, which is critical to maintaining high software quality within microservices architectures.

The results of this research aim to demonstrate the impact of AI-driven test generation on improving software quality, enhancing development efficiency, and streamlining the release process while adhering to DevOps best practices.

This work was conducted with Blue Reply to evaluate the effectiveness of this tool in supporting developers writing unit tests.

This thesis is structured into four chapters, aiming to provide the reader with a clear understanding of the solution's design and implementation, while also explaining the reasoning behind each technological choice.

In Chapter 2, we provide the reader with an overview of software testing, including how it is performed and the metrics used to evaluate the efficiency of a test suite.

We will also discuss the technologies chosen to implement this system, explaining the rationale behind our selections. Furthermore, we will explore the role of the DevOps methodology in facilitating a structured approach to test generation, along with widely adopted practices such as CI/CD pipelines. Lastly, we will review existing research on current approaches, covering solutions for unit test generation and common practices for enhancing the testing phase.

In Chapter 3, we will discuss the actual implementation of the system. Given the complex flow of the implemented system, we have chosen to first provide an abstract description of its behavior to help the reader understand its functionality. Following that, we will dive deeper into the detailed explanation of its implementation. However, the code is not included in the thesis, as we believe that incorporating it could lead to confusion. The complete code is available on the following GitHub repository: `https://github.com/francescocellamare/AI-driven-unit-test-generation`.

In Chapter 4, we will outline the results collection process by comparing the system's performance on the two most commonly used OpenAI models. Additionally, we will compare the artifacts generated by these models with a test suite manually written by a developer.

In Chapter 5, we reflect on the results obtained, examining whether the tool can effectively serve as a support for unit testing. We also discuss its limitations and suggest potential future improvements to improve its efficiency.

Chapter 2

Background

This chapter will explore Software Testing, emphasizing the importance of maintaining a robust test suite within a software project and how to evaluate its effectiveness in ensuring project quality. After providing a brief overview of software testing and its various categories, we will delve into the motivation behind this thesis and examine why testing is often a budget-intensive activity in software development. Next, we will discuss the DevOps methodology, highlighting its role in enhancing software quality by identifying bugs early, accelerating development processes, and fostering effective communication among developers. Following that, we will cover Cloud Computing, focusing on the AWS platform and its widespread use in creating pipelines and managing required hardware for deployment. Finally, we will review current techniques for test generation and explore how Generative AI emerged as a potential solution.

2.1 Software Testing and Software Testing Pipelines

Software testing is the process of performing an analysis in our code with the goal of identifying errors to ensure correctness. However, achieving correctness in a real-world application is not feasible. This is because even moderately sized software introduces complexities that are difficult to manage, making it impossible to account for every potential cause of errors or system failures. As the software grows in size, the number of possible test cases increases exponentially and so do the possible scenarios. It is essential to test and verify all potential inputs and outputs, as testing only boundaries is insufficient to guarantee correctness. Quality Assurance (QA) teams typically determine which functionalities require a certain level of correctness based on factors such as criticality of the feature, business priority or risk assessment. This approach helps establish minimum testing standards for critical areas, allowing testers to focus on more precise

and targeted tests.

Software is tested using automated software testing, which is the common approach used by testers to perform repetitive tasks that would be time-consuming if done manually and is fundamental to Continuous Delivery (CD) and Continuous Testing (CT). These techniques provide advantages such as scalability, consistency, and increased efficiency during the testing phase of the Software Development Life Cycle (SDLC), while also being cost-effective by minimizing the reliance on manual testing. In fact, relying on manual testing becomes unsustainable as the software project scales because it is not easily repeatable and is prone to human error.

2.1.1 Test Design and Classification

Software testing can be classified into five basic categories: unit testing, integration testing, system testing, regression testing and end-to-end testing. Each of these categories evaluates the software by focusing on different aspects and abstractions of the system. Below is an overview of each testing type.

- **Unit Testing**: ensures that each software unit, defined as the smallest isolated testable component of an application, functions correctly.
- **Integration Testing**: examines whether different software units operate together as intended to detect faults at the interfaces between components.
- **System Testing**: involves testing the entire software system as a whole.
- **Regression Testing**: concerns the re-execution of the whole test suite after changes are made, in this way we can check if those changes modified the system's behavior and resistance to potential crashes.
- **End-to-end Testing**: involves testing the entire software system and all its external dependencies in a simulated production environment. It is generally performed manually.

Further classification can be defined based on the used approach as follows:

- **Black-Box Tests**: software under testing is treated as a black box, internal implementation is not considered so results are derived only from the provided inputs.
- **White-Box Tests**: the opposite scenario, it involves full knowledge of the internal structure and logic of the code in order to explore all the possible branches. This approach may require the use of mocking. This technique allows testers to define

the behavior of a function inside a dependencies module which is assumed to be 'correct'. In this way, we can isolate different components.

This thesis focuses on unit testing performed using the white-box approach after each unit is considered complete by the developer, in order to verify whether it works as expected.

Let us now introduce the Software Testing Life Cycle (STLC), a structured process designed to ensure quality by planning a set of test cases (also known as test suite). The objective is to demonstrate that the software behaves as expected and appropriately handles intentional exceptions. STLC consists of the following phases:

- **Requirement Analysis**: QA team is responsible for analyzing the requirements document to identify the core functionalities on which the testing process will focus.
- **Test Planning**: activity to designate the testing goal, scope and overall strategy for the process. During this phase, the type of test to be developed is determined.
- **Test Case Development**: based on the deliverables of the planning phase, the QA team develops the test suite.
- **Test Case Execution**: test suite is executed having each test categorized as passed (if the outcome matches expectations from the planning phase) or failed (indicate that an error has been found).
- **Test Result Reporting**: QA team analyzes and discusses the bugs report deliverable to identify which scenarios caused unintentional behaviors and how to address those issues.

2.1.2 Coverage Metrics and Mutation Analysis

When discussing the effectiveness of automated tests, it is essential to evaluate their quality using specific metrics. Among the most commonly used, there are code coverage metrics, which measure the extent to which the code under test is exercised by the current test suite. Various code coverage metrics focus on different aspects of the code. The most prevalent in practical environments are Branch Coverage, Method Coverage, and Line Coverage, which respectively measure the ration of control-flow paths, methods, and lines of code that have been tested over the total possible ones. However, code coverage metrics have their limitations and can sometimes overestimate the quality of the code. For example, a test might be executed and adjusted to match incorrect expected outputs, merely to prevent it from failing.

Another approach to evaluating the quality of automated tests is called Mutation Analysis. This kind of analysis tries to overcome the limitations of the coverage metrics to obtain an esteem of the quality of the tests. The core idea involves modifying the source code to introduce deliberate bugs, referred to as "mutants". For example, a mutation can look like the expression x < y changed with x <= y, in this case a test to check boundaries should exist. The test suite is expected to detect these bugs by causing previously passing tests to fail after a new execution. Mutants are generated based on equivalence rules of the mutation analysis tools, which substitutes a new snippet of code inside the codebase and the whole test suite is executed again. If a test detects a bug, the mutant is considered 'killed'. Test suites that identify a higher number of faulty versions are deemed to be more effective than those that detect fewer. To quantify the quality of the test suite, a mutation score is calculated as the ratio of mutants killed to the total number of mutants introduced.

2.1.3 Cost of Testing

According to a report by Lionel Sujay Vailshery (2024), as of 2019, an average of 23 percent of organizations' annual IT budgets were allocated to quality assurance and testing. This represents a decline from the peak of 35 percent recorded in 2015, as shown in Figure 2.1. Quality assurance and testing aim to deliver IT products securely and without defects, a need that has grown alongside increasing digitalization. The rising reliance on cloud-based test environments, used by over half of companies by 2017, reflects the industry's focus on IT security testing. However, the cost of these functions remains a significant challenge, motivating organizations to look for ways to reduce their QA and testing expenses. [2]

Software testing is fundamental to ensure a high quality product to the end user. To reduce these costs, it is crucial to adopt effective techniques. One such approach is implementing Continuous Integration and Continuous Deployment (CI/CD) which enhances software quality while reducing costs and effort in testing through Continuous Testing (CT) approach. Adopting CI/CD with an Agile methodology, developers frequently commit code to a central repository after testing in the local environment. This practice helps reduce software delivery time and facilitates early bug detection.

Detecting a defect in the early phases of development helps reduce costs by reducing the effort and time needed to resolve it later when launching the product, particularly considering the interconnections between software modules and user dissatisfaction when encountering these issues. Other units may depend on the faulty component, so addressing the scenario which leads to the error can lead to other errors when performing a regression testing. This is why such a technique must be adopted by developers and it forms the core principle that drives this thesis. A qualitative description of the correlation between hidden bugs and relative costs to fix those during the software lifecycle

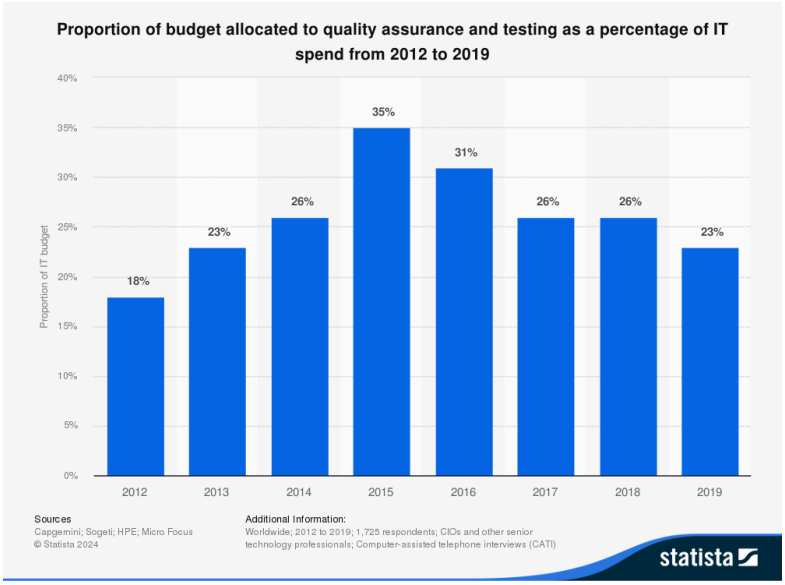


Figure 2.1. Proportion of budget allocated to quality assurance and testing from 2012 to 2019 [2].

is shown in Figure 2.2.

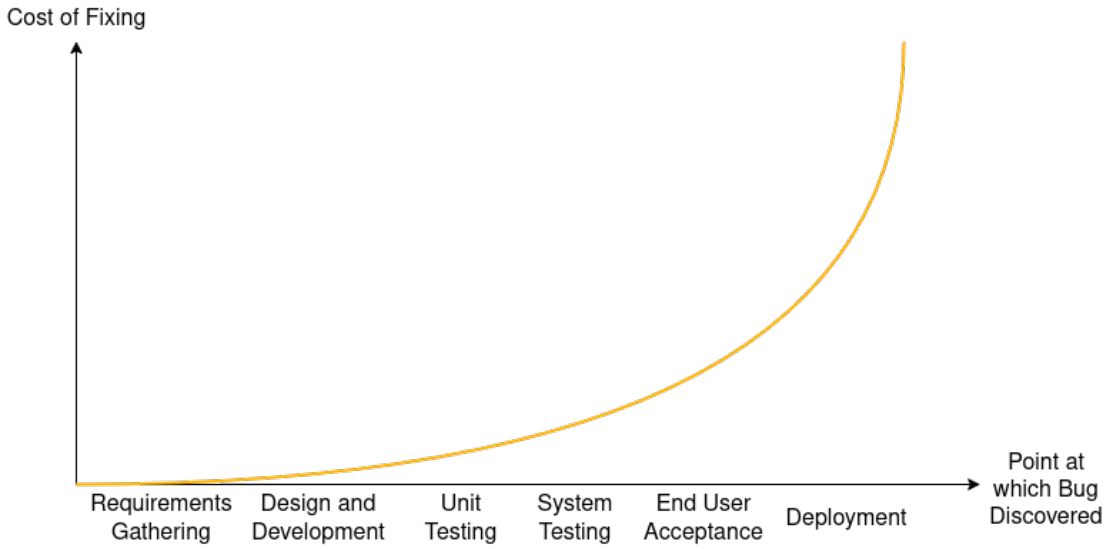


Figure 2.2. Cost of fixing at different stages of development

2.1.4 DevOps Practices

DevOps is the combination of practices, tools and approaches to development which had its origin in the Agile practices. It emerged to address the need for stronger synergy between the development and operations teams. The Agile Manifesto emphasizes key principles of Agile development such as customer satisfaction, rapid adaptation to changes and continuous delivery through iterative process. However, it does not explicitly include engineering practices for automating operations and infrastructure. To reduce this gap, DevOps was introduced. DevOps fosters collaboration by breaking down silos between development and operations, streamlining processes, and enhancing automation. Today, Agile and DevOps are recognized as complementary practices that work together to improve software development, deployment, and operations. For the purpose of this thesis, we will focus on DevOps practices rather than providing an in-depth explanation of Agile methodology.

DevOps is guided by five core principles, represented by the acronym CALMS. In developing our solution, we closely followed each of these principles [1], which include:

- Collaboration and shared responsibility between the development and operations teams
- Automation tools to reduce errors, increase efficiency and automate repetitive tasks
- Lean strategy to eliminate time-consuming process which would increase time to delivery
- Measurement of performance by collecting and analyzing data
- Sharing information and learning across teams

We introduce here the practices Continuous Integration and Continuous Deployment (CI/CD) and Continuous Testing (CT). Using the CI/CD pipeline, SDLC has been streamlined by automating the build, testing, and deployment processes. The concept behind CI is to shift away from the traditional approach, where developers work independently for extended periods before merging their changes. In contrast, CI encourages frequent integration of code, which helps prevent the accumulation of bugs and the slow delivery of features that can result from the challenges of merging developers' work. With CI, a Versioning Control system manages developers' code, merging changes each time a new feature is tested so that the pipeline is triggered for each modification. By triggering a build phase and a testing phase each time a new feature is considered completed, any potential bugs are detected at an early stage. Addressing issues as soon as they are introduced in the codebase, which is the definition of CT, prevents building additional code on these faulty components. If no issues arise, the deployment phase is started.

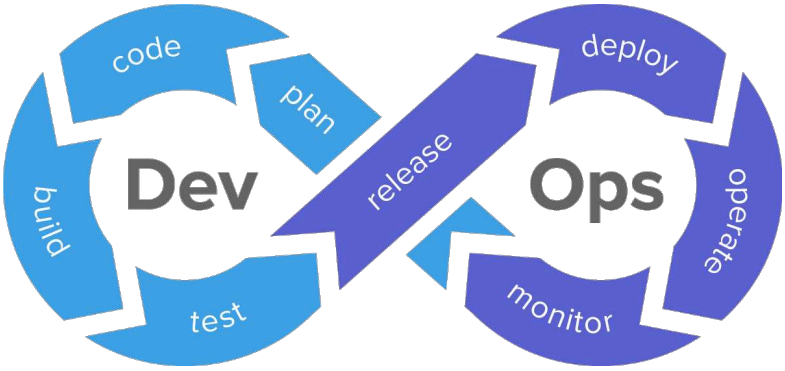


Figure 2.3. CI/CD pipeline

CI/CD paradigm is a crucial aspect of a modern DevOps ecosystem. In addition to improving software reliability, it also fosters better collaboration and communication within the team, enabling faster time-to-market for products. Under a DevOps model, developers and operations teams collaborate closely so that a single merged team works across the whole SDLC of the application, resulting advantages like writing code that takes into account the environment in which it is run or reducing the needed time period between developing and deploying. In a DevOps's approach, it is common to design a software application based on the microservices architecture. A microservice can be defined as a self-contained unit that exposes a set of operations over its own data. This architectural pattern organizes a software application into a collection of loosely coupled, self-deployable services that interact through inter-service APIs interfaces. However, it introduces further complexity in the design phase compared to a monolithic approach which involves a unified unit but it brings several benefits such as:

- **Modularity**: the application is easier to understand, develop, test and maintain while also enabling a distributed development by allowing separated autonomous teams to independently develop each service. This shortens development cycle times.
- **Flexible Scaling**: services' infrastructure may be resized according to the needs if there is a spike in demand. This approach reduces costs compared to a monolithic application, which would require provisioning a larger quantity of resources to ensure fault tolerance during periods of increased demand since it is treated as a whole rather than at the granularity of microservices.
- **Resilience**: service independence increases application's fault tolerance. In a monolithic approach, if a single component fails, it will cause the whole application to fail. On the other hand, if a single microservice fails, the application can still operate, though the failed service may become unavailable.

- **Technology Flexibility**: each autonomous team can select the most suitable technology to develop their service, without impacting the other teams.

Adopting a microservices architecture can accelerate our application's time to market. By allowing teams to work in parallel on independent microservices while simply defining API interfaces for system interactions, the development process becomes significantly faster. This aligns with the DevOps's principles.

2.1.5 Spring Boot Framework

In the field of software development, a framework is a well-structured set of libraries that provides a flexible range of software components that help developers accelerate software development to production deployment. More specifically, for our case we adopted the Spring Boot framework.

Based on the Spring framework, Spring Boot adds functionalities that allow users to create stand-alone application based on the Spring environment and the Java platform. We chose Spring Boot because it is widely adopted in real-world applications and offers several benefits to developers to streamline the development of microservices-based applications. [12]

The fundamental goals of Spring Boot are [7]:

- **Rapid Development:** Spring Boot provides defaults and auto-configuration for various components most used for projects. This reduces the boilerplate code, allowing developers to get started quickly and focus more on implementing business logic.
- **Microservices Architecture:** as we said, Spring Boot is well suited for microservices-based applications, providing an easy way to configure and let those communicate.
- **Auto Configuration:** Spring Boot automatically configures various components based on dependencies and annotations while also providing a wide set of starter dependencies for software projects within the Spring ecosystem. Additionally, it extensively leverages the Inversion of Control (IoC) design principle, which reduces class coupling within the project, enhancing modularity and code testability.
- **Testing Support:** Spring Boot provides excellent support for writing tests, making it easier to perform unit tests, integration tests, and end-to-end tests.

All these aspects are essential for developing a microservice application without worrying about common default practices such as the login phase, logging, and security

role assignments. Additionally, they contribute to building a well-tested and easily maintainable application. Spring Boot provides essential components for both unit and integration testing. Among these, we used JUnit 5 and Mockito, which are the de facto standards for writing automated unit tests and mocking components without relying on real dependencies, respectively.

The adoption of DevOps practices yields significant benefits for software teams. However, CI/CD pipeline must rely on an infrastructure that is properly configured and adapted to the evolving needs of the development process.

2.2 Service Architectures

Cloud computing is a paradigm for enabling network access to a scalable to resources by dynamically provisioning, configuring and de-provisioning servers as needed. These resources can be physical machines or virtual machines as well as advanced cloud also includes other kind of computing resources such as network and security equipment [6]. Cloud computing enables customers to access resources by outsourcing them through a provider, eliminating the need to purchase and maintain their own hardware infrastructure.

Infrastructure as Code (IaC) refers to the practice to provision, support and update a computing infrastructure through code rather than manual configuration. Setting up an infrastructure can be a time-consuming activity, as system administrators need to carefully manage servers, networks and other components required to run the system while also adapting to any necessary configuration changes. IaC is used to automate infrastructure creation, streamlining the process of setting up environments and minimizing the manual configuration and the potential errors associated with it.

Additionally, among the benefits, the same environment can be easily deployed on a different system at another location using the same IaC definition. For example, you could use the same configuration template to define and then deploy the exact same infrastructure in multiple regions, ensuring consistency across environments. Declarative IaC allows us to describe the needed resources and settings as well as define how these components are going to interact in a configuration template. The IaC tool then analyzes it and creates the required infrastructure to instantiate each service on the chosen cloud computing platform.

2.2.1 AWS Cloud Computing

This thesis focuses on using Amazon Web Services (AWS). AWS is a comprehensive cloud computing platform provided by Amazon, offering a wide range of cloud-based services such as computing power, storage, networking and databases that allows companies to deploy, manage and scale their application using a pay-as-you-go pricing model so that instead of buying, owning and maintaining a physical infrastructure, such services can be purchased and used as needed. This is the main idea of cloud computing, offering also the following benefits:

- **Elasticity**: the ability to scale computing power up or down based on demand.
- **Cost Savings**: fixed costs for maintaining physical infrastructure are replaced with variable costs based on actual usage.
- **Agility**: the flexibility to quickly adopt and utilize different technologies from the available set of services making the cost and time to experiment and develop significantly lower.
- **Fast Global Deployment**: the application can be deployed in multiple regions around the world, improving performances and reducing latency due to geographical redundancy.

Cloud computing is generally categorized into three service models:

- **Infrastructure as a Service (IaaS)**: allows developers to specify their infrastructure requirements by defining the desired architecture. The IaaS cloud provider will instantiate each required resource and configure it properly according to the developer's needs. With IaaS, the responsibility to obtain, install, configure and maintain the architecture's hardware is moved from developers to the provider with many benefits such as fast deployment, no need for fixed costs to maintain the hardware and hardware's faults are handled by the provider.
- **Platform as a Service (PaaS)**: consumer does not manage the underlying infrastructure such as virtual servers, operative systems and networks. Instead, he requests a development environment from the PaaS cloud provider. The consumer's primary responsibility is to configure and utilize the provided development environment effectively. This is the basis of serverless applications which do not require a proper handling of the used resources but they adapt according to the demand. AWS Lambda is an AWS service based on the serverless approach.
- **Software as a Service (SaaS)**: the end user interacts directly with the software without managing the underlying infrastructure, development or maintenance.

They simply use the software through a web interface without having to worry about updates, patches or ongoing maintenance.

AWS is designed to be used with IaC so complex cloud architecture can be managed safely by defining them in code. AWS CloudFormation is an IaC tool that helps with the creation, deletion and updating of AWS resources. Resources are declared in a structured template file in either yaml or json, or through the CloudFormation UI. The next step involves organizing these resources into a logical group called stack so that a stack is created for each CloudFormation template. This tool enables developers to manage the infrastructure required to implement DevOps practices, such as defining a CI/CD pipeline declaring each step of the pipeline inside the template without the need to declare how the resources depend on each other. The most commonly used AWS services to create a pipeline include AWS CodePipeline, AWS CodeCommit, AWS CodeBuild and AWS CodeDeploy. Let's briefly review these services, with a more detailed discussion provided in the chapter 3.

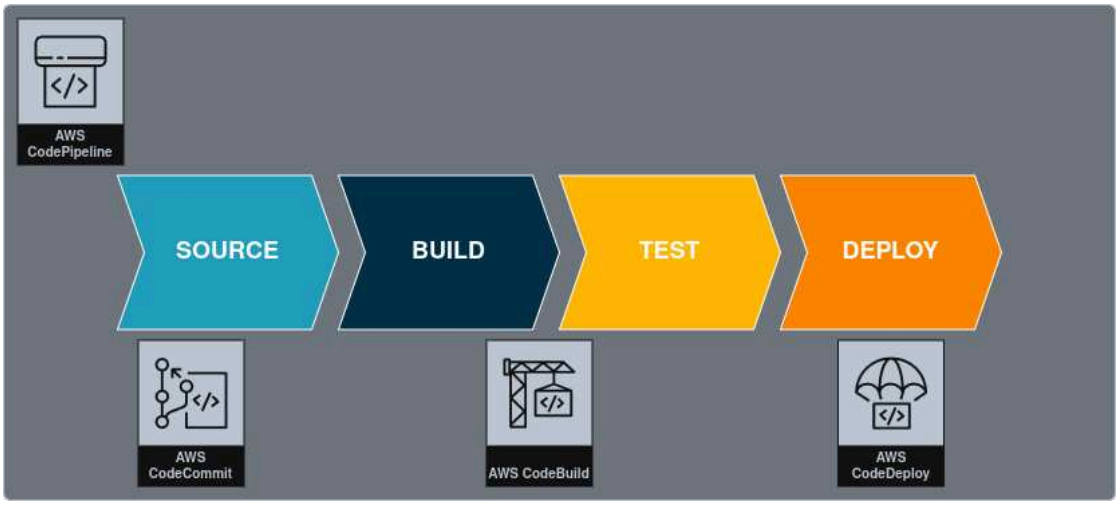


Figure 2.4. CI/CD pipeline on AWS

A typical CI/CD pipeline workflow is made up of four stages as illustrated in 2.4. Each stage contains specific actions performed on the application artifacts, such as the source code or a compiled Java project. CodePipeline is a service designed to model and automate the steps necessary to release software. Each pipeline stage can be a different AWS service, allowing developers to build the operation chain according to their needs. In the example, the first stage called Source uses AWS CodeCommit. CodeCommit is a version control service for storing assets like source code, so that whenever changes are committed to the remote repository, the pipeline is automatically triggered to initiate execution. The second and third stage, respectively Build and Test, can be performed

using AWS CodeBuild. CodeBuild is a service to compile source code, execute unit tests and produce deployment-ready artifact. Using this service, an organization can avoid the need for a private build service and instead pay only the required time to perform the action. It also offers a pre-configured build environment and most common build tools such as Apache Maven or Gradle. The final stage is the Deploy, here the build artifacts are deployed to a remote running server. AWS CodeDeploy is a deployment service that integrates with various compute platforms such as AWS EC2, AWS Lambda or AWS ECS.

Although the cost to prepare and maintain the infrastructure is reduced using the AWS's pay-as-you-go model, we have seen that CT practice is essential for identifying errors as early as they are introduced in the codebase in order to reduce bug fixing costs. However, this approach requires significant computational power to constantly execute regression tests using the test suite. AWS services address this need by providing build servers without requiring the ownership of dedicated hardware. Nevertheless, the primary challenge remains the time required to create a test suite that ensures the correctness of key application functionalities while maintaining overall quality. The literature explores various tools and techniques to assist testers in creating tests and preventing errors early in the development phase.

2.3 Test Generation and GenAI

2.3.1 Static Analysis

Code inspection is the process of performing an analysis of the code without the execution of any automated test. This is also called static analysis and, even though it can be helpful for developers because it does not require code execution and allows a basic early bug detection, runtime errors cannot be detected as well as it does not provide a valuable measure for the code correctness. Static analysis can be considered as a precursor to other testing methods and should be used as a basic tool for developing. In fact, static analysis is concerned in analyzing the structure of the code and it is useful to detect possible logical mistakes or questionable coding practices. When performed prior to dynamic test it can be really useful to reduce the number of test cases with benefits on test planning. [3]

A tool available on the market is SonarQube. Sonar's Clean as You Code approach is a software development practice based on the principle that new code (code that you added or modified recently) needs to comply with quality standards. The Sonar solution implements Clean as You Code by warning you whenever issues are detected in your new code. [11] Even though it accounts benefits to developers by ensuring that new added code is always 'clean', this approach is still not enough.

2.3.2 Static Test Generation and Dynamic Test Generation

Code inspection can be considered as a valid complementary technique; in particular, we want to automate the generation of tests. In the literature, two main approaches have been explored: static test generation and dynamic test generation. Static test generation consists of analyzing a program statically reading the program code and simulated its execution using symbolic expression. The goal is to define every possible execution path and explore the entire tree of computations that the program can produce by considering all potential values of the input parameters. All possible paths are enumerated by considering each branch in conditional statements. However, this approach is not always feasible. Consider the following code snippet 2.3.2:

```
1 int isHashOf(int x, int y) {
2     if( x == hash(y) ) return 1;
3     else
4         return 0;
5 }
```

This function is designed to return a true result if the first parameter is the hash of the second. It relies on a mathematical hashing function, which is known to be mathematically non-invertible. This is where the limitation of this approach becomes apparent. Static test generation cannot produce two input values, x and y, which are guaranteed to satisfy or violate the condition x == hash(y), because the hash function is specifically designed to prevent such reasoning. Unfortunately, the same issue arises with other complex program statements, such as pointer manipulations or system calls, which are quite common. Moreover, symbolic execution does not scale, and the number of feasible paths can become potentially infinite in the case of unbounded loops, leading to path explosion. Using this approach, we can actually generate a test suite but there is still static analysis limitation to detect runtime issues because it does not require code execution. Dynamic test generation complements static generation.

Dynamic test generation consists of executing the program with some random inputs while collecting symbolic constraints at each conditional statement along the execution. The program is then re-executed and variants of the previous inputs are determined based on system's response to explore alternate control-flow paths. This process is repeated until a specific program statement is reached. Dynamic test generation seems to resolve the previously shown problems because it actually creates a test suite with high branch coverage based on failed execution but, on the other hand, it is computationally expensive. [5]

Directed Automated Random Testing (DART) is a tool for automatically testing software that implements dynamic test generation. Let us delve into the mechanics of

dynamic test generation using DART with the following code example 2.3.2 from the article [4]:

```
1 int f(int x) { return 2 * x; }
2 int h(int x, int y) {
3     if (x != y)
4         if (f(x) == x + 10)
5         abort(); /* error */
6     return 0;
7 }
```

The function h() may lead to an abort statement for some value of x and y. The probability of detecting an error state through random input generation is very low. In contrast, DART is able to dynamically gather knowledge about the execution in what it is called a directed search. Starting with random input, DART calculates an input vector for the next execution. This vector contains values that represent the solutions to symbolic constraints collected from predicates in branch statements during the previous execution so that, the new input vector attempts to force the execution through a new path. The execution of DART proceeds as follows:

1. **Initial Inputs**: We start with the inputs $x = 123$ and $y = 456$.

2. **First Execution**: During the first execution, the program evaluates the first `if` statement. The execution path does not enter the `then` block of the second `if` statement. Based on the conditions encountered, the sequence of symbolic constraints is:
$$\langle x_0 \neq y_0,\ 2 \cdot x_0 \neq x_0 + 10 \rangle$$

3. **Second Execution**: For the second execution, the last constraint is negated:
$$\langle x_0 \neq y_0,\ 2 \cdot x_0 = x_0 + 10 \rangle$$
The system then solves for this new scenario, resulting in input values $x_0 = 10$ and $y_0 = 456$. In this execution, the abort() function is thrown leading to the error.

Another tool that uses dynamic test generation is called Pex. Developed by Microsoft, Pex is a tool that helps developer in writing Parametrized Unit Test (PUT) for .NET language. PUT is a different methodology to write tests. It is essentially a method with parameters in which developers model at a higher level of abstraction what the tested unit behavior is. Then Pex is going to translate it in test cases to cover all the reachable branches. Here is an example of PUT:

```
1 void AppendToListTest(ArrayList list, Object element) {
2     Assume.IsNotNull(list);
3     list.Add(element);
4     Assert.IsTrue(list.Contains(element));
5 }
```

For this snippet, Pex generates two test cases that cover all the reachable branches based on the branch on line 2.

```
1 void AppendToListTest_g1(ArrayList list, Object element) {
2     AppendToListTest(new ArrayList(0), new Object());
3 }
4 void AppendToListTest_g2(ArrayList list, Object element) {
5     AppendToListTest(new ArrayList(1), new Object());
6 }
```

The key difference between Pex and DART lies in the required input: Pex needs PUT as input from the developer, which involves more effort upfront but allows for the explicit definition of the unit's behavior. In contrast, DART relies on random testing and feedback from program execution failures to generate test cases. [5].

2.3.3 Model-Base Test Generation

A third approach discussed in the literature is Model-Based Testing (MBT). MBT is an automated test generation technique that relies on models of the System Under Test (SUT) to create tests. During the early stages of software development, system specifications are often represented using models such as UML. These models can then be used to generate a corresponding test suite. Since test suites are derived from models rather than source code, MBT is typically considered a black-box testing approach, as the testing tool does not have access to or knowledge of the underlying code.

Model-based Integration and System Test Automation (MISTA) is a tool that applies the MBT approach. It utilizes Predicted-Transition (PrT) nets as an expressive formalism for building functional models of our SUT. The context diagram of MISTA, illustrated in 2.5, provides an overview of its operation. The tool receives as input the Model-Implementation Description (MID) that consists of abstract models of our system, also known as test models, which describes SUT's behavior and by a Model-Implementation Mapping (MIM). The MIM specification maps the element of the test model to their target implemented constructs. In addition, users must define the desired coverage criterion and specify the target programming language. [13]

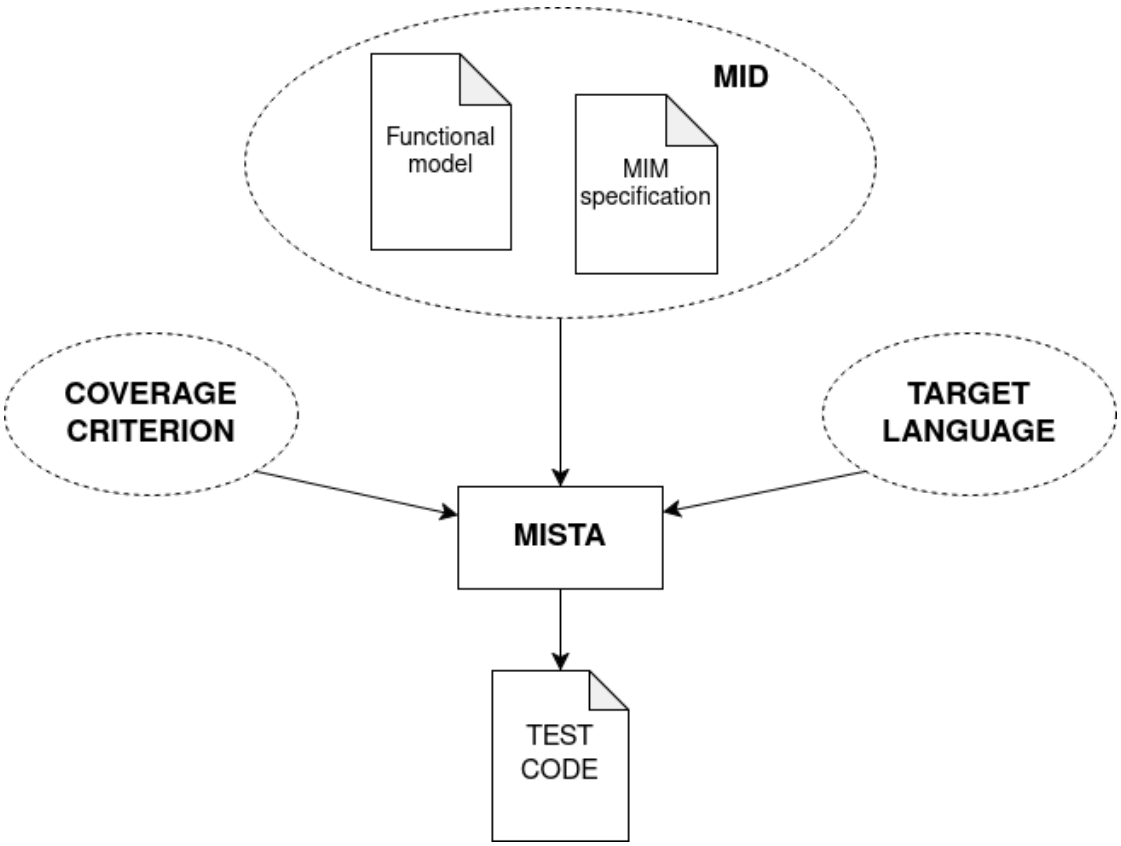


Figure 2.5. Context diagram of MISTA

So far, we have examined various techniques described in the software testing literature, each of those offering advantages and disadvantages. Code inspection with static analysis is useful in bug detection in early stages of development but not enough to provide a qualitative description of our system and its correctness. It is often used as a complementary tool alongside other techniques to generate test suites. One such technique is static test generation, which is capable of creating a test suite by just analyzing the code and going through all reachable control-flow path without its execution. However, this is also its limitation. It is not capable of adapting to runtime errors as well as it cannot analyze code whenever a more complex statement is used such as system calls to the operative system or pointers manipulation. Both static analysis and static test generation require minimal input from developers and do not demand significant computational power, though only static test generation produces concrete testing artifacts. Dynamic test generation tries to overcome the problem of static test generation and it actually does but it is computationally expensive to produce a test suite. For example, tools like DART generate test cases by leveraging runtime feedback or need

a way to define general behavior of the under testing unit as developer input and then produce the test suite (Pex). The primary drawback of dynamic test generation lies in its high computational requirements. Finally, we considered Model-Based Testing (MBT), which generates tests from abstract models representing the System Under Test (SUT). These models must accurately capture the system's behavior, requiring developers to invest effort early in the development process to create and maintain them so it shifts the workload toward upfront modeling, which can be challenging and time-consuming.

2.3.4 Deep Learning and GenAI

In the years since its wide deployment, Artificial Intelligence has significantly influenced various aspects of human life and has numerous applications across a variety of industries. AI refers to the ability of machines to perform tasks that would normally require human intelligence such as speech recognition, decision-making, accurate medical imaging analysis, weather forecasting and so on. One of its primary advantages lies in automating repetitive and time-consuming tasks, freeing up humans to focus on more complex work. Generative AI (GenAI) is a form of artificial intelligence capable of producing new content or artifacts, including images and videos. It is designed to understand and generate both natural languages and formal languages, such as mathematical expressions or programming languages so that applications include also code generation. The central focus of this thesis is to evaluate whether GenAI models can effectively generate test suites and assess their potential usefulness to developers in the software development process.

These models, known as Large Language Models (LLMs), are designed to process natural language inputs and generate text based on the knowledge on which they have been trained. LLMs are trained using a self-supervised technique that does not rely on labeled data. Instead, the model generates its own labels, enabling it to learn patterns related to syntax, semantics, and predictive capabilities. However, this approach can also introduce biases and inaccuracies resulting from incorrect data or training. Our focus is not to explain how these model works but to highlight that these are not immune to errors.

2.3.5 Tokenization and Context Window

OpenAI is a company focused on AI research and development. It offers a diverse set of Generative Pre-trained Transformers (GPTs) with different capabilities and prices by providing a web interface or exposing structured APIs to interact with their models. The two models used for the purpose of this thesis are GPT-4o-mini and GPT-4o. However, OpenAI imposes limitations on its models, such as reasoning capacities and generation sizes. These constraints are closely tied to the tokenization process, which

involves breaking down input text into smaller units (tokens) that the model can process efficiently. Each token is assigned a unique integer that is used as a key inside a lookup table. The row indexed by the token's unique integer is its vector representation which the model uses to compute responses by predicting the next token. Machine learning models process numbers instead of text, so tokenization mapping is essential; without it, these models would not be able to know on raw textual input.

Let us describe these according to the provided OpenAI documentation:

Model	Context Window	Max Output Tokens
gpt-4o-mini	128,000 tokens	16,384 tokens
gpt-4o	128,000 tokens	16,384 tokens

Table 2.1. Specification on token usage

Table 2.1 outlines the token limitations for both the produced output and the context window. The context window refers to the maximum number of tokens that can be used in a single request for both input, output and reasoning tokens. Figure 2.6 shows how the context window is applied to the request and its potential impact on the response. When the reasoning process becomes computationally intensive, the response output may be truncated as a result. Both models share the same values for these properties; however, a key distinction lies in the number of tokens required for reasoning and planning responses. According to OpenAI, GPT-4o-mini has reduced versatility and reasoning capabilities compared to GPT-4o. This limitation will be further analyzed in the chapter 4.

2.3.6 Fine-Tuning and Prompt Engineering

Moreover, OpenAI gives the possibility to perform fine-tuning on its model. It is suggested to apply fine-tuning to a smaller model in order to improve its performances on a specific task to make those comparable to a bigger and more expensive model. Fine-tuning is a method that allows us to train our custom version of a GPT model through an additional learning process. By preparing tailored training data and fine-tuning the model, it is possible to reduce costs and improve request latency. However, we chose not to implement fine-tuning in our solution. Instead, most common practices in prompt engineering such as few-shot learning, which involves including examples directly in the prompt to help the model better understand its task.

Additionally, other prompt engineering strategies were utilized, such as using delimiters to distinguish different parts of the input, structuring properly the prompt and specifying the steps required to complete the task. Together, these approaches eliminated the need for fine-tuning and its potential application will be explored in chapter

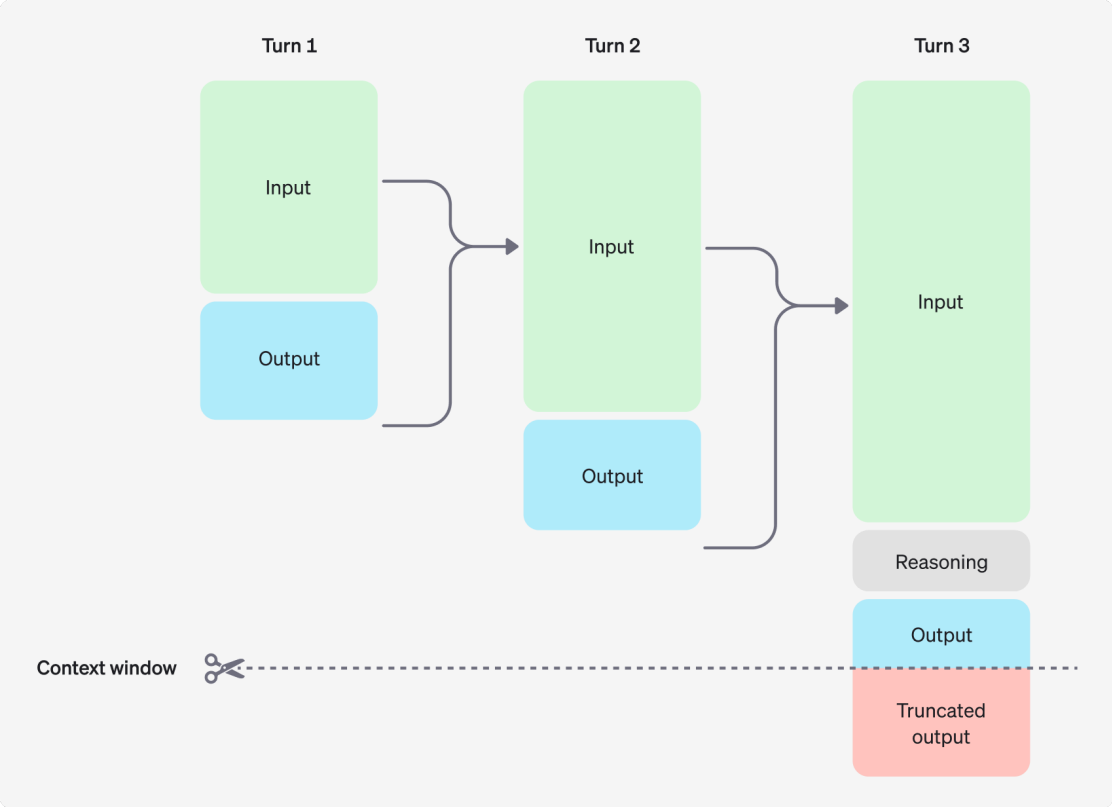


Figure 2.6. Token distribution for the context window [9].

5.

To further optimize response size, allowing for larger output within the context window, and reduce system costs, prompt caching was also considered. Prompt caching allows us to reduce latency and costs by optimizing the structure of the prompt without the need to change model configuration. It is based on the idea of caching repetitive sections of our prompt so that, if a cache hit occurs, the benefits of caching are fully realized. To effectively use prompt caching, the prompt must be structured with all the static content placed at the beginning of the request.

2.3.7 Models Pricing

OpenAI also provides usage costs for its models, as summarized in Table 2.2. As anticipated, GPT-4o is the most expensive model due to its advanced intelligence and but

with high latency. However, OpenAI suggests that through fine-tuning and model distillation, GPT-4o-mini can achieve comparable results at a lower cost. The main focus of this thesis was not on performance, so these techniques were not applied.

Model	**Input Price**	**Cached Input Price**	**Output Price**
gpt-4o	$2.50	$1.25	$10.00
gpt-4o-mini	$0.15	$0.075	$0.60

Table 2.2. Pricing Details of Models (for 1M tokens)

Chapter 3

Methodology

This chapter will delve into the practical implementation of our solution. We will begin by introducing each AWS service, explaining their roles and how they interact within our architecture. Next, we will break down the actions performed at each pipeline stage, with a particular focus on the State Machine. We will first provide a high-level overview of the process using a flowchart representation, illustrating the overall workflow. Then, we will analyze each state of the machine, detailing the logic behind its implementation. Finally, we will explore the two deployment options provided by AWS and explain our decision to use Fargate over EC2 as ECS capacity layer for our application.

3.1 Selected Technologies

As discussed in Chapter 2, this section will provide a detailed description of the AWS services used, their interactions within the overall architecture, and their specific roles. To give the reader a clearer understanding of the design and implementation, an overview of each service is presented in Table 3.1.

Table 3.1: Overview of AWS Services Used in the Project

Service	**Category**	**Purpose**	**Used For**
CloudFormation	IaaS	Automates the setup, deployment, and management of AWS resources	Defines all AWS resources in the main template

Service	Category	Purpose	Used For
CodeCommit	Developer Tools	Source control repository for storing code in Git	Serves as the source repository for triggering the CI/CD pipeline
CodeBuild	Developer Tools	Build service for compiling and testing code	Compiles and tests the source code initially, after each test generation, and builds the Docker image
CodePipeline	Developer Tools	Automates CI/CD workflows	Defines and manages the CI/CD pipeline, integrating and handling artifacts between services
IAM	Security	Manages user access and permissions for AWS resources	Configures permissions and access control for each AWS resource
SNS	Messaging	Simple Notification Service for sending messages and alerts	Sends email notifications for reports and alerts
StepFunctions	Compute	Coordinates multiple AWS services	Defines the main workflow logic using a state machine
Lambda	Compute	Serverless compute service that runs code in response to events	Executes Python scripts in response to events, used within Step Functions to implement business logic
S3	Storage	Scalable object storage for storing data and backups	Stores pipeline artifacts, reports, and statistics generated during the process
ECS	Compute	Scalable container management service for running Docker containers	Deploys the finalized Dockerized application using AWS Fargate
ECR	Storage	Elastic Container Registry for storing Docker images	Stores the finalized Docker images for deployment

Service	Category	Purpose	Used For
EC2	Compute	Virtual servers for running applications on AWS infrastructure	Hosts the actual server that runs the application logic
CloudWatch	Monitoring	Monitors and logs AWS services and resources	Serves as the centralized logging system for all events and metrics
API Gateway	Networking	Manages APIs for creating, deploying, and securing RESTful APIs	Manages API requests for email redirection and routing
VPC	Networking	Virtual network in the cloud for securely connecting AWS resources	Provides the underlying network infrastructure for securely connecting and exposing the application
EventBridge	Messaging	Event bus for building event-driven applications by routing events	Acts as an event bus to trigger the pipeline based on incoming events

These are the selected services used in our design, with each one chosen based on its cost-effectiveness, ensuring the most convenient pricing option. In the next section we will talk about how the whole pipeline has been designed and how these services interact with each others.

3.2 Architecture and Design of the Solution

Our solution required the implementation of a four-stage pipeline. Using CodePipeline, we designed each step of our software release process to include the most common phases of a typical CI/CD pipeline. Each phase, referred to as a stage in CodePipeline terminology, represents a logical unit that executes a set of operations called actions within an isolated execution environment. Actions can perform various tasks, consuming artifacts or generating new ones to be used in subsequent stages. An artifact, in this context, refers to any data or file transferred between stages. Actions pass their output to subsequent actions for further processing through the pipeline's artifact bucket. CodePipeline manages these artifacts by copying them to the artifact store using AWS Simple Storage

Service (AWS S3), where they are accessed by the respective actions.

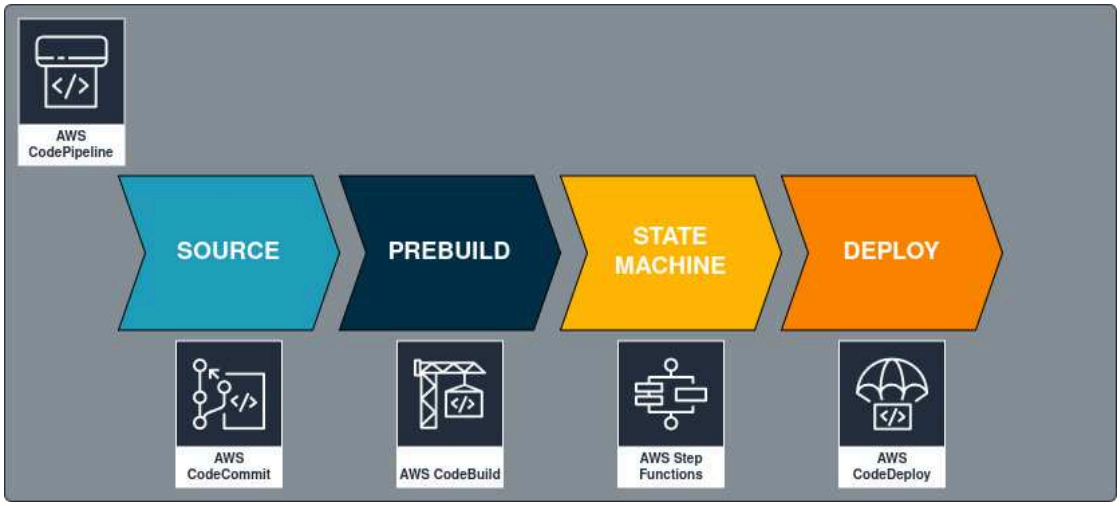


Figure 3.1. Implemented CI/CD pipeline

Figure 3.1 shows an overview of the designed pipeline. As a typical CI/CD pipeline there is the first Source stage based a remote CodeCommit repository. In this stage, the source code is retrieved and passed to the next stage as an output artifact. In the Build stage, project is compiled and current test suite is executed, we will see that compilation errors would lead the pipeline to failure whereas failing tests do not. The third stage, called State Machine, contains the core logic for test generation. This stage is implemented using AWS StepFunctions, a service designed to orchestrate complex execution flows by coordinating multiple AWS services according to the defined state machine. Upon a successful completion of the State Machine stage, the code is expected to have a new or updated test suite with no failing tests, making it ready for deployment. This finalized version of the application is referred to as the final revision.

The following sections detail the design and functionality of the state machine, highlighting its role in managing the complexities of the process and ensuring seamless integration with other components of the architecture.

3.2.1 State Machine

Third stage of the pipeline is where the code logic is implemented. StepFunction is based on state machines, which are also called workflow. Workflows are composed by a series of event-driven steps called states. Using Amazon States Language (ASL), a JSON based language to declare each state inside the state machine, is possible to model complex

flow easily. The following is an example of state definition:

```
{
  "HelloWorld": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",
    "Next": "AfterHelloWorldState",
    "Comment": "Run the HelloWorld Lambda function"
  },
  "AfterHelloWorldState": {
    "Type": "Pass",
    "End": "True"
  }
}
```

each state is identified by a unique name within the scope of the StepFunction and described by a set of properties. States can perform whatever possible operation using AWS services, those are categorized into two categories:

- **Flow states**: controls the execution flow of the state machine. Operations includes mapping data, choice selection based on variables and busy waiting.
- **Task states**: represents a unit of work that another AWS service performs; for example, a task state can initiate a project build using CodeBuild or execute a serverless function with AWS Lambda

To ensure a clear and smooth explanation, we will begin by introducing the state machine with a flowchart that illustrates its execution flow at a higher level of abstraction. Afterward, we will delve into the details. Figure 3.3 provides an overview of the entire state machine implemented using StepFunction.

Our CI/CD pipeline is triggered each time a developer commits its work on CodeCommit, more precisely a good programming practice would require to use a parallel development branch called testing in our case. Let's assume that the codebase of the project is without any previously compilation errors and the current test suite is correct or totally missing. The initial step is to verify this assumption by performing a full compilation of the project and running the existing test suite. If a compilation error occurs, the pipeline will fail entirely. This aspect is not depicted in the flowchart because, as mentioned at the start of this chapter, the focus is on testing and ensuring its correctness rather than on compilation errors. However, if any test fails, the pipeline is not going to fail its execution.

At this point, two possible outcomes can occur:

1. A compilation error occurred so our assumptions are wrong and the pipeline fails
2. One or more tests failed, workflow goes on
3. No tests failed, workflow goes on

Based on the latest test suite results and commit metadata, a structured report will be produced and analyzed in order to handle each possible case. If case 2 occurs, the system will handle the failing tests, and this scenario will be explored in more detail later. Otherwise, the execution can continue.

If no tests failed, the corresponding section of the report will present this information. Then, the system will start to properly handle each committed file by analyzing each file's metadata and operating according the logic illustrated in Figure 3.2.

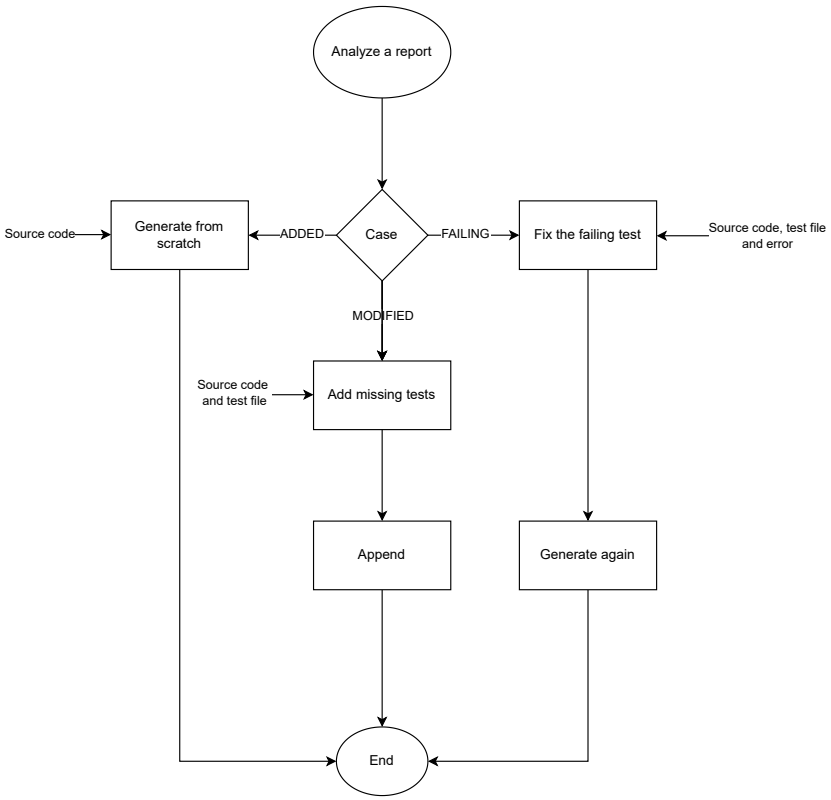


Figure 3.2. Analyze flow chart

A commit to a remote repository can involve adding, updating, or removing files. A file is categorized as "added" when it is pushed to the repository for the first time,

"updated" if any modifications are made to it (such as additions or deletions within the file), or "deleted" when it is removed from the repository. These information are taken from the commit related metadata and included inside the previously created report.

Following this, the test generation phase begins. The process adapts dynamically, creating prompts tailored to the specific files committed, as categorized earlier. For a newly added file, the system generates tests from scratch based on the fetched source code. As well, an updated would cause the already existing test suite to be adapted with the new code. For instance, an update can be a newly added method inside a class. In this case, this method is not going to have a related set of tests yet so the system will generate only the missing tests within the corresponding test class. An update can also be a removal of an already tested method. However, this scenario cannot be considered because the first build would make the whole pipeline fail because a compilation error occurs so that, the user is responsible to remove the related tests when a method is deleted. The third and most general case of an update involves any modification within a method. Detecting this situation without storing history of each file is challenging. However, it can be reasonably assumed that updates to a method's logic may cause some tests to fail since unit testing closely aligns with the statement-level details of a method. In this situation, the subsequent build and test execution phase will likely identify these failures, initiating a feedback loop for addressing and fixing the failing tests.

The purpose of our solution was not to automate queries to a GPT model as a means of speeding up the testing phase. Our goal was not to shift the tester's responsibilities onto a generative AI model but rather to provide a tool that assists testers in this time-intensive task. This is the reason why system will always asks user's approval at least one time. We designed two possible cases where this may happen. A first optional user approval is required whenever at least one test fails, initiating the feedback loop and prompting the user for guidance on how to proceed. This initial check ensures the quality of the generated tests. The second user approval occurs at the end of the entire test generation phase. The system requests the user to verify the coherence of the generated test suite, providing testers with a clear understanding of what has been generated by approving the pull request merge.

To further clarify the process, we will now delve into the first user approval step in more detail, focusing on the specific branch of the flowchart dedicated to this check. This branch handles the scenario in which at least one test fails, triggering the feedback loop. A new report containing information about test suites execution is going to be created and it will pair failing test method and the error that caused its failing. This test failing report will be sent to the user using its registered email in order to let him analyze which test has failed and the reason behind this failing. Inside the email, the user can choose how to proceed. Three cases have been designed as shown in Figure 3.3:

1. **Generate**: the system attempts to resolve the reported error by re-querying the

GPT model, providing it with the necessary information to correct the issue. This process is repeated for each raised error.

2. **Fail**: tester makes the whole system fail, the pipeline will end here its execution and the current revision is not going to be deployed later.

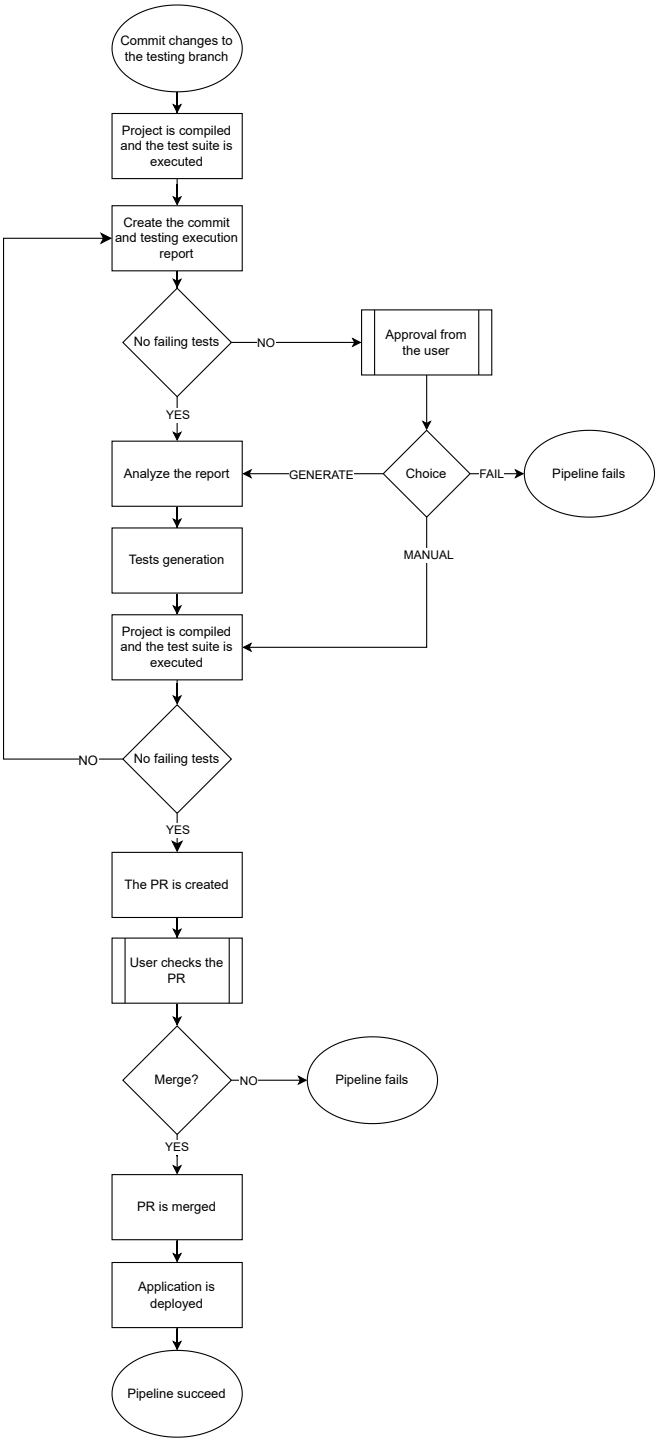
3. **Manual**: tester notifies the system he provided to fix the reported issue. In this way, the next phase is not going to be a new test generation but the system will try to compile and execute tests again in order to check tester's work.

Manual execution was designed because, as we said in the previous chapter, GenAI models are not deterministic. As we shall see later, this issue can arise in situations where the GPT model generates code with incorrect syntax. To prevent the pipeline from failing every time this happens and to maintain the tool's usability, we allow testers the option to manually adjust the generated tests before merging them.

All these three cases will always lead the system to a new compile and test execution phase, this is required to guarantee a stable and tested version to be deployed. If no errors occur at test generation or the developer's commit did not require to add or update tests at all, the system will proceed with the final user's approval. A pull request of the current version is created to the main branch and another communication is sent to the user asking to check and manually merge the created request. In this way, the user is aware of what has been generated and can choose to approve the merge or reject it. The latter will lead the pipeline to fail its execution. Otherwise, the pull request is merged and the deploy can start. Here we proceed from the StateMachine stage of the pipeline to the final Deploy stage.

This was a high-level overview of the system's behavior, its interaction with the user, and the conceptualization of the feedback loop. In the next section, we will dive into the details of the actual implementation.

Figure 3.3. Pipeline flow chart



3.2.2 Implementation Logic

The entire implementation relies on CloudFormation to manage the infrastructure. A CloudFormation template describes all the required resources including their properties and configuration. Using a CloudFormation template significantly simplifies infrastructure management compared to manually provisioning each service via the AWS Management Console or AWS CLI. CloudFormation instanciates each template definition into a logical group called stack allowing operations like stack update or deletion with a single instruction. This simplifies infrastructure management, offering a quick and secure way to delete an entire stack and automatically cleaning up all its dependencies. It also enables seamless stack updates, providing an efficient way to add resources to the infrastructure and configure those for communication. Without this automation, the developer would be required to instantiate each of those conscious of constraints such as the correct instantiation order, which can be complex and error-prone.

CloudFormation allows us to create a set of resources using a visual interface, also known as AWS Infrastructure Compose, to select, configure and link services. However, we opted to define our architecture using a written template in YAML syntax for flexibility and control. Due to the template's length, we will not present it in its entirety here. Below is a snippet that illustrates the structure and format of a typical CloudFormation YAML template:

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Template to define a CodePipeline pipeline and related resources.

Resources:
  # CodePipeline resource
  PipelineInstance:
    Type: AWS::CodePipeline::Pipeline
    Properties:
      Name: !Ref PipelineName
      RoleArn: !GetAtt CodePipelineServiceRole.Arn
      ArtifactStore:
        Type: S3
        Location: !Ref ArtifactStoreBucket
      Stages:
        - Name: Source
          Actions:
            - Name: SourceAction
              ActionTypeId:
                Category: Source
                Owner: AWS
                Provider: CodeCommit
                Version: '1'
              OutputArtifacts:
```

```
          - Name: SourceOutput
        Configuration:
          # ...
          PollForSourceChanges: false
        RunOrder: 1
    - Name: PreBuild
      Actions:
        - Name: PreBuildAction
          ActionTypeId:
            Category: Build
            Owner: AWS
            Provider: CodeBuild
            Version: '1'
          InputArtifacts:
            - Name: SourceOutput
          OutputArtifacts:
            - Name: DeployOutput
          Configuration:
            # ...
          RunOrder: 2
          Namespace: BuildVariables
    - Name: StateMachine
      Actions:
        - Name: FeedbackStateMachine
          ActionTypeId:
            Category: Invoke
            Owner: AWS
            Provider: StepFunctions
            Version: '1'
          Configuration:
            # ...
          InputArtifacts:
            - Name: DeployOutput
          OutputArtifacts:
            - Name: MergedDeployOutput
          RunOrder: 3
    - Name: Deploy
      Actions:
        - Name: ECSDeployAction
          ActionTypeId:
            Category: Deploy
            Owner: AWS
            Provider: ECS
            Version: '1'
          InputArtifacts:
            - Name: MergedDeployOutput
          Configuration:
            # ...
          RunOrder: 4
```

```yaml
# AWS S3 bucket for artifacts
ArtifactStoreBucket:
  Type: AWS::S3::Bucket
  Properties:
    BucketName: !Ref BucketName

# IAM role for CodePipeline
CodePipelineServiceRole:
  Type: "AWS::IAM::Role"
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: "Allow"
          Principal:
            Service: "codepipeline.amazonaws.com"
          Action: "sts:AssumeRole"
    Policies:
      - PolicyName: "CodePipelinePolicy"
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
            - Effect: "Allow"
              Action:
                - "s3:*"
                - "codecommit:*"
                - "codebuild:*"
              Resource: "*"
            - ...
```

The template snippet shows the declaration of a CodePipeline instance with its related bucket for storing artifacts and role. This is the actual pipeline declaration of our solution. The first stage includes a CodeCommit action, which retrieves source code from a remote repository. Instead of relying on continuous polling, an event-driven approach is used to trigger the pipeline. Polling has been disabled in the stage configuration and replaced with an event bus created using AWS EventBridge. This event bus acts as a connector between the CodeCommit event source and the pipeline, filtering incoming events based on predefined criteria. In our setup, the bus is configured to forward all commit events to the pipeline.

For each event received, the pipeline is triggered and can proceed with the second stage. A CodeBuild project is configured to perform this first build phase. Figure 3.4 shows what happens when a build is run with CodeBuild:

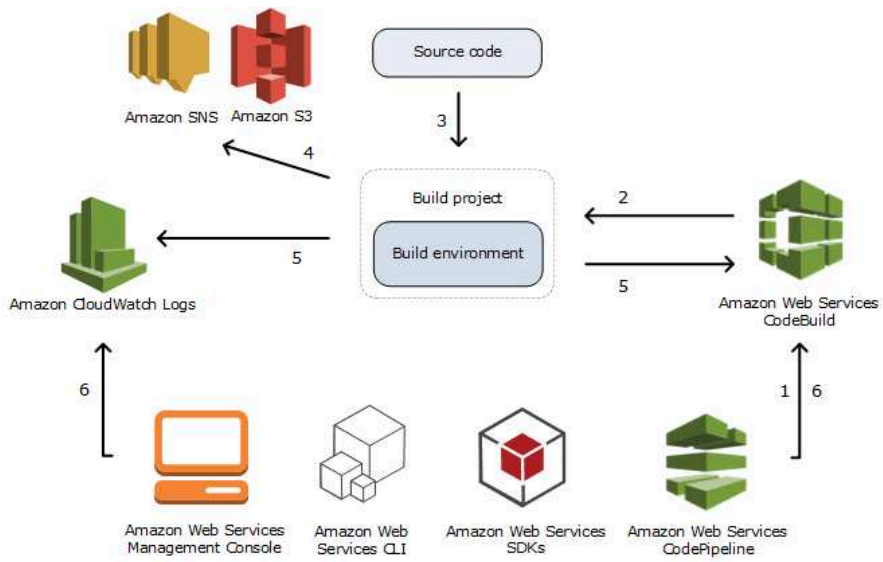


Figure 3.4. CodeBuild workflow [10].

A configured CodePipeline triggers the CodeBuild to start the build process (transition 1). A build project includes information about how to run a build, including where to get the source code, which build environment to use, which build commands to run, and where to store the build output. Based on these configurations, the build environment is created (transition 2). The source code is then retrieved and incorporated into the project definition (transition 3), and the build process begins, guided by the build specification (buildspec). The buildspec file is a collection of build commands and related settings in YAML format. Within it, we can define output artifacts, which are transferred to the associated project bucket (transition 4), while the build logs are streamed to AWS CloudWatch (transition 5). CodeBuild pricing is based on the pay-as-you-go model, build calculation is calculated in minutes from the time of submission until the build is terminated. Moreover, AWS offers us a wide range of compute EC2 instances having different computational power as well as pricing per minute.

The following snippet illustrates how the build has been designed, with some unnecessary code omitted for clarity:

```
version: 0.2
env:
shell: /bin/bash
exported-variables:
  - BUILD_STATUS

phases:
    install:
```

```
      on-failure: ABORT
      commands:
        # Maven and Java configurations

    pre_build:
      on-failure: ABORT
      commands:
        # The S3 Bucket is cleaned

    build:
      # compilation errors lead to failure
      on-failure: ABORT
      commands:
        - echo "Compiling the Spring Boot project"
        - mvn clean compile -q

    post_build:
      # test errors let the pipeline continue
      on-failure: CONTINUE
      commands:
        - echo "Running the test suite"
        - mvn test -q && export BUILD_STATUS="SUCCEEDED"
                              || export BUILD_STATUS="FAILED"
        - echo "Tests completed"
        - echo "Uploading Surefire reports to S3"
        # ...
        - echo "Uploaded Surefire reports"
        - git show --name-status HEAD > report.txt
        - echo "Generated report"
        - echo "Building the Spring Boot project"
        - mvn clean package -q -DskipTests
        - echo "Creating dependency graph with jdeps"
        # ...
        - echo "Ending"

artifacts:
    files:
      - '**/*'
    name: DeployOutput
    secondary-artifacts:
      ReportOutput:
        files:
          - report.txt
        name: ReportOutput
        discard-paths: yes
      DeployOutput:
        files:
          - target/*.jar
        name: DeployOutput
```

```
        discard-paths: no
```

Based on Maven, we configured the environment to build Spring Boot projects based on Java 17. The project first verifies the environment, followed by a cleanup phase before it starts compiling the project and running the existing test suite, causing the pipeline to fail only for compilation errors. Additionally, several artifacts are generated and stored in the associated bucket, such as Surefire reports for failed tests, details about committed files using the git command, and information about the internal project's dependencies with JDeps, a Java tool that fetches dependencies for each defined Java class. After the build, the state machine is started.

We decided not to present the full ASL definition of the state machine, as it would be too lengthy. Instead, Figure 3.5 illustrates its actual implementation using AWS services filtering out flow states to map data between transactions. Moreover, task states and its related service have been highlighted. Each used task state can be either a AWS CodeBuild service or a AWS Lambda service. AWS Lambda is an ideal compute service based on serverless approach, we do not need to manage physical servers. Instead, just choosing an environment and configuring it, we can run our code according to the PaaS computing model. All the Lambda functions are written using Python language based on a python3.10 runtime, we will highlight the most important ones.

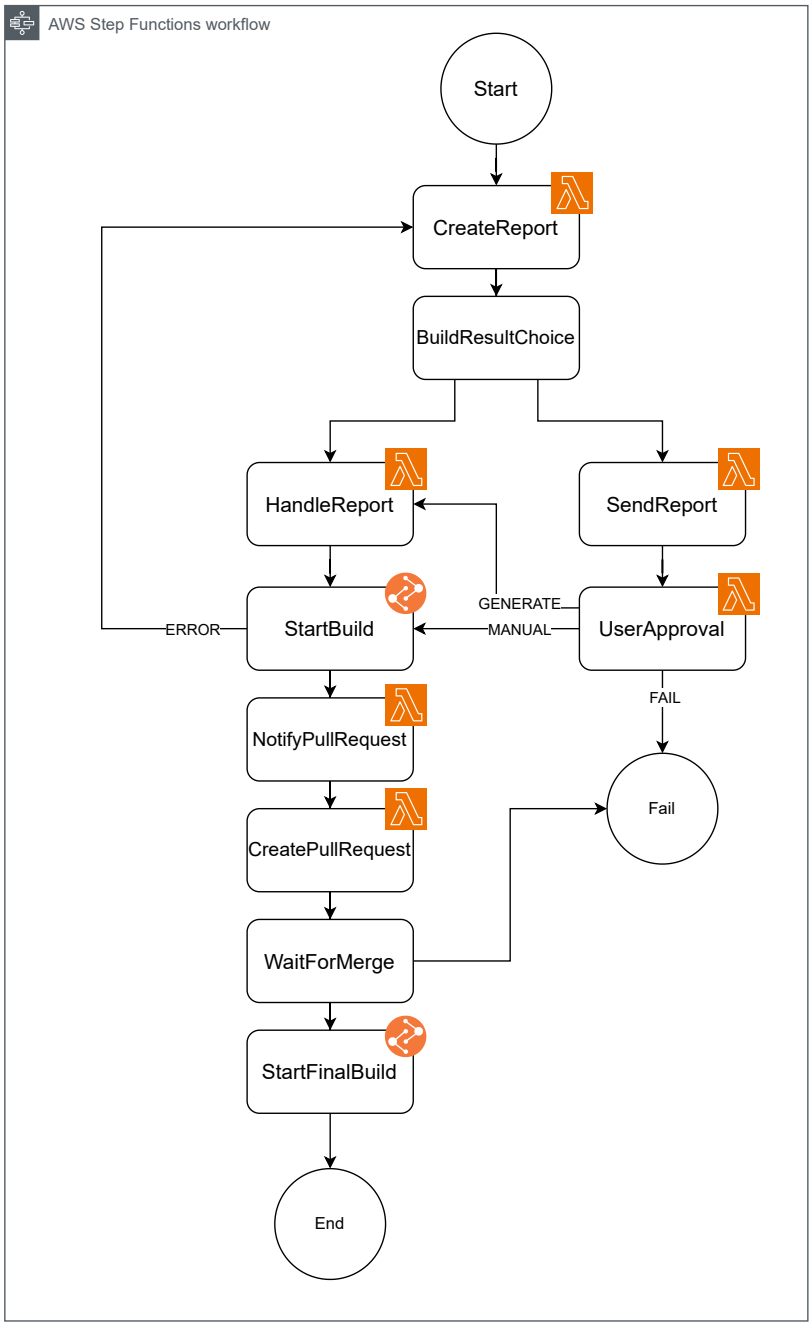


Figure 3.5. State machine workflow

CreateReport

First state is CreateReport, it is a Lambda function used to create reports needed in the pipeline execution. Based on the results of the previously build phase, this function elaborates the produced artifacts and prepares a structured report as the following:

```
{
    "testReport": [
        {
            "TestSet": "com.example.demo.testsuite",
            "TestsRun": 5,
            "Failures": 1,
            "Errors": 0,
            "Skipped": 0,
            "TimeElapsed": 10.32,
            "TestFailures": [
              {
                "test": "com.example.demo.testsuite.test_n1",
                "error": "assertion failed"
              }
            ]
        },
        // ...
    ],
    "commitReport": [
        {
            "path": "src/main/java/com/example/demo/GenericClass.java",
            "action": "A"
        }
    ]
}
```

The testReport object provides details about the most recent test execution, as generated by Surefire, a Maven plugin commonly used during the testing phase to run an application's unit tests. This plugin produces reports summarizing the test results, including the number of tests that passed, were skipped, or failed. In cases in which tests fail, the report includes an array having the failed test IDs along with their corresponding error messages. The commitReport object contains details about the commit that initiated the pipeline by parsing the result of the git show command. It lists the files affected by the commit and specifies the operations performed on them, such as additions, updates, or deletions.

HandleReport

The main logic of test generation is implemented inside this task. Here we handle what has been received by the CreateReport task so that, test generation from scratch, test appending and test fixing is addressed according to it. We can classify each of them based on the action field of each object inside commitReport. Action's allowed values are:

- **A**: The file has been added.
- **D**: The file has been deleted.
- **M**: The file has been modified or updated.
- **Test identifier and relative error**: Specifies tests that failed and need fixing, including the identifier of the test and the associated error details.

Each of those will lead the system to construct an appropriate prompt for invoking OpenAI APIs. In order to do so, managing the runtime was necessary as the OpenAI client is not present inside default dependencies provided by AWS. However, AWS Lambda allows the creation of layers. Those usually contain library dependencies, custom runtime or configuration files. We created a custom layer based on python3.10 runtime and adding the necessary dependencies to instantiate the OpenAI client. Additionally, we incorporated another AWS-provided layer to securely store secrets. This was needed because OpenAI API requests require a generated API key for authentication. Storing it in clear inside the script is not a viable solution, so we used the AWS Secret Manager to securely manage and access the API key.

```
Define a function to handle the report event.

    Extract repository and branch details from the stack parameters

    Filter the list of files in the commit report based on the following:
        - The file matches the regular expression for Java files.
        - The file path is approved (using the tested paths JSON file).

    If there are filtered files:
        - Determine the branch name for testing:
            - If on the master _testing_, create a new branch with a
              timestamp.
            - Otherwise, use the existing branch name.

        Create a dependency tree to track file dependencies.

        For each file in the filtered list:
            If a new testing branch is needed:
```

```
            - Create the branch using the repository name, original
              branch, and new branch name.
            - Set the flag indicating the testing branch has been
              created.

        Handle different types of file actions:
            For added files:
                - Skip files that are test files.
                - Fetch the source code for the file being added.
                - Fetch dependencies for the file from the dependency
                  tree.
                - Generate a test suite for the file using the source
                  code and dependencies.
                - Commit the generated test suite to the new branch.

            For modified files:
                - Fetch the source code for the modified file.
                - Fetch dependencies for the file from the dependency
                  tree.
                - Fetch the existing test suite for the modified file
                  .
                - Update the test suite using the source code, test
                  suite, and dependencies.
                - Commit the updated test suite to the new branch.

            For fixing tests:
                - Identify the source code file corresponding to the
                  test file.
                - Fetch the source code and dependencies for the file
                  .
                - Fetch the test suite for the file.
                - Update the test suite based on the changes and
                  dependencies.
                - Commit the fixed test suite to the new branch.

Otherwise, if no relevant files are found:
    - Use the original branch name.

Prepare the next event and return.
```

Listing 3.1. Pseudocode for the HandleReport Lambda Function

The pseudocode for the HandleReport Lambda function is provided in Listing 3.1. The files received in the commitReport are filtered to exclude non-Java code files and those outside the defined testing boundaries. Specifically, the tool can be configured to test only specific directories within the project, as not all parts of the project are suitable for testing. This filtering is carried out here and is configured using a JSON file located in the project's root directory, where the directories to be tested are listed. Moreover, before starting with test generation, a dependencies tree is created by analyzing the previously produced JDeps artifacts.

Next, for each item in the list of testable and allowed files, we proceed with test generation. The abstract flow of this process is illustrated in Figure 3.2. Additionally, to overcome certain limitations in GPT generation, we implemented logic to construct a prompt that includes the code for the project's internal dependencies. This was necessary to provide the generation model with the context of what is being tested. However, this approach also introduces a limitation in the solution. We will discuss this in Chapter 5.

The prompt is generated using the fetched source code. By applying best practices in prompt engineering, the prompts are structured to minimize costs and latency, produce a well-organized output, and allow the user to interact with the test generation process. Here is the template prompt 3.2 for creating a test suite from scratch:

```
Provide a complete test suite for the given Java code using the Spring
    framework with maximum code coverage. Use JUnit5 and Mockito for
    creating the pure unit tests, focusing on Mockito mocks and skipping
    Spring context loading. Focus on covering all possible branches and
    edge cases. Try to understand the code first and never assume the
    method behavior according to the function name.
In some cases, there could be comments to document the method to test,
    such as the following example:

/*
Function to test the absolute value of the sum of the parameters
*/
public int AbsOfSum2Int(int a, int b) {
    return Math.abs(a + b);
}

Try to understand the code first and never assume the method behavior
    according to the function name.

You will receive the source code in the following format:

// source code section
public class Controller { code... }

###

// dependencies section (it can be empty)
public class ControllerDependency { code... }

Moreover, the path of the file is {path}, which allows you to determine
    the package and import the entire project generically. For example,
    you can import all classes in the project using import com.example.
    demo.*;. When adding imports, ensure they are as generic as possible
    to cover all required classes.

# What you are testing
```

```
It is a Spring Boot project made in the java language which is able to
    interface with other external resources i.e. databases, so do not test
     edge cases like setter for the Ids of the entities/models inside the
    tables.

# Output Requirements
- Output only the Java code, do not write the ```java and ``` quotes
- Exclude any comments or explanations if not already written
- Ensure all dependencies needed for the test are appropriately managed
    and configured.
- Print the related package as first line according to the received {path
    }
```

Listing 3.2. Prompt template for generating from scratch

The prompt includes predefined static code at the beginning, which helps with caching hits. Additionally, it clearly instructs the GPT model on how to approach the task, specifying what it will receive and what it is expected to produce. The dynamic part of the prompt is driven by the user's request and contains the source code to be tested, along with any relevant dependencies' code that will be appended. Now we can call the OpenAI client interface:

```
response = client.chat.completions.create(
    model=Model.GPT4o.value,
    messages=[
        {
            "role": "system",
            "content": prompt
        },
        {
            "role": "user",
            "content": input_content
        }
    ],
    temperature=TEMPERATURE,
    top_p=TOP_1
)
```

Listing 3.3. Create request OpenAI APIs example

The input_content contains the formatted request to the GPT model. Moreover, we can manually adjusts relevant parameters like the chosen model, the temperature and the top_p parameters that, respectively, the degree of randomness in the output, and the scope of the candidate words considered during the text generation process. The temperature controls the variability of the generated text, where a lower temperature ensures more deterministic and focused output and a higher temperature promotes creativity and randomness in the generated responses. Meanwhile, the top_p parameter

fine-tunes the selection of the next word, limiting it to a subset of words that together represent the most probable candidates. We put as values, respectively, 0.5 and 1 so that it is possible to strike a balance between creativity and coherence in the generated output. A temperature of 0.5 provides a moderate level of randomness. Setting top_p to 1 means that the model considers the full range of possible candidates for the next word, without restricting it to a smaller subset.

StartBuild

After the test generation phase, a post-generation build process is initiated. This step is crucial for verifying the GPT-generated test suite by compiling and executing it against the updated version of the project. By doing so, we can ensure that the generated tests are syntactically correct and functionally valid. There is nothing to highlight in this build phase, the project is just compiled and test suite is executed so the relative buildspec is omitted.

SendReport and UserApproval

Using AWS SNS, a human-readable report is sent to the developer to provide details about issues found during the last build. This task is triggered whenever a build fails. The email contains key information about the execution, including a unique identifier for the pipeline execution, a link to review the code at the commit state, and three API calls to the AWS API Gateway for further investigation. This ensures that developers have the necessary details to quickly identify and address build failures.

The three exposed endpoints allow the user to decide how the system should handle the situation. The Generate endpoint triggers test generation again, enabling the model to adapt the failing tests to address the identified issues. The Manual endpoint notifies the system to proceed directly with the build phase, indicating that the developer has resolved the reported problems. The Fail endpoint stops the current execution of the pipeline.

The UserApproval can be a long running task potentially, the user have to check the generated test or eventually fix them using the manual approach. This functionality is implemented using an asynchronous job within the state machine. A Lambda function pauses execution while waiting for the user's response via a task token, which resumes the state machine's workflow. The process is triggered when the user clicks on the links provided in the report, sending a request to the API Gateway that activates the Lambda function and continues execution.

NotifyPullRequest and CreatePullRequest

Using AWS SNS, notifies the developer of the successful end of test generation phase. Two endpoints are exposed this time, respectively, to approve the pull request creation or to reject it. Here the second checks should happen. The developer should check the current state of the branch and eventually approve the pull request merge. The state machine stops its execution as the restart is triggered in the same way as explained for UserApproval.

StartFinalBuild

This is the final build phase before deployment. During this phase, test-related statistics are exported to an S3 bucket, and the software project is packaged as a Docker image. The Docker image is created based on the Dockerfile located in the root directory which basically sets up the Java 17 runtime environment to execute the Spring Boot project and exposes the 8080 port to allow communication with the container. Once the image is built, it is pushed to the ECR storage where it will be retrieved during the next deploy phase by AWS ECS. The following is the buildspec for this process:

```
version: 0.2
phases:
    pre_build:
      commands:
        - echo Logging in to Amazon ECR...
        # ...
    build:
      commands:
        - echo "Creating Jacoco report"
        # ...
        - echo "Creating PIT testing report"
        # ...
        - echo Building the Docker image...
        - docker build -t $REPOSITORY_URI:latest .
        - docker tag $REPOSITORY_URI:latest $REPOSITORY_URI:latest
    post_build:
      commands:
        - echo Pushing the Docker images...
        - docker push $REPOSITORY_URI:latest
        - echo Writing image definitions file...
        - echo "Creating artifact for the next phase"
        # ...
        - echo "imagedefinitions.json file created."
artifacts:
    files:
```

```
    - imagedefinitions.json
```

3.2.3 Deploy

Final stage of our pipeline is Deploy stage. At this point, we should have a tested project, build and packaged in a Docker image stored in ECR. Two possible solution are used to deploy our application in AWS platform. First requires the usage of Elastic Computer Cloud (EC2). EC2 provides on-demand, scaling computing capacity to host our applications. Using EC2, we can launch as many virtual server instance as we want. Each EC2 instance is fully configurable by the customer, allowing him to set various configuration of CPU, memory, storage and also design the whole inner network infrastructure. The latter can be achieved using Virtual Private Cloud (VPC), another AWS service that allows to create logically isolated virtual network using the AWS environment. Each defined network must be placed inside a region chosen in the list of AWS provided regions. We can declare whatever instance we need inside the VPC as well as expose something to be reached. Figure 3.6 shows a context diagram of a VPC implementation where a subnet has been defined and EC2 instances are exposed to the internet through an internet gateway. Several networking components can be instantiated inside a VPC network but it is developer responsibility to configure those properly.

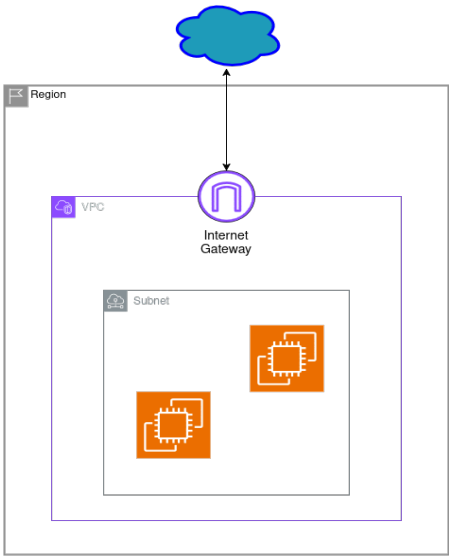


Figure 3.6. Virtual network example

There's no additional charge for using a VPC, only components usage is paid and each EC2 instance uptime according to the instance configuration. However, we opted

to use another AWS service in the final implementation of this thesis. As we said, we relied on AWS ECS as deployment service.

AWS ECS is a service for containers orchestration that helps in deploy, manage and scale containerized applications. It is well integrated with AWS ECR as public repository for storing images and Docker. The main advantage of using ECS is its simplified way of deploying software using containers instead of configuring a proper environment in a EC2 instance. ECS is based on 3 layers as shown in Figure 3.7.

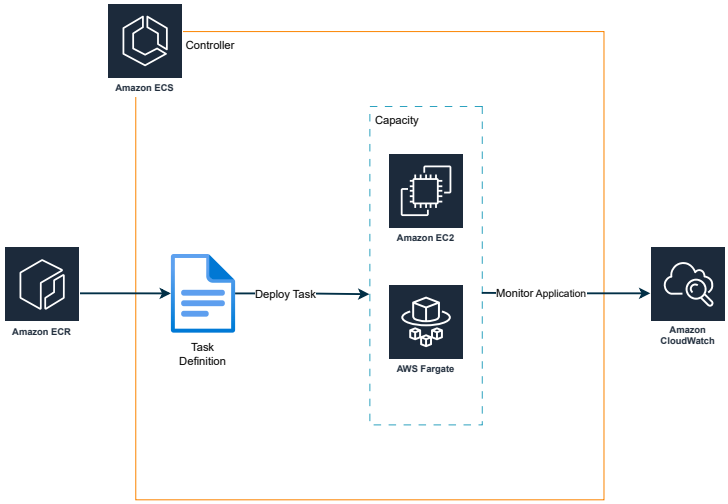


Figure 3.7. ECS context diagram

The controller is the software responsible for managing applications, which in our case is Amazon ECS. Capacity refers to the underlying infrastructure where containers run, with options including EC2 instances or AWS Fargate. Monitoring is handled through services like AWS CloudWatch or by observing the runtime execution. After creating and storing the image in ECR, a task definition must be declared. A task definition is a blueprint for our application and tells ECS what to instantiate and configure. For example, the following snippet of yaml is the task definition of a simple Spring Boot backend using Postgres as database service. Two containers are going to be run and one of those is going to be the resulting image generated inside the state machine while the Postgres image is pulled from Docker Hub. We can also define the execution environment to configure the used images.

```
Task:
    Type: AWS::ECS::TaskDefinition
    Properties:
      Family: spring-boot-app
```

```
Cpu: 256
Memory: 512
NetworkMode: awsvpc
RequiresCompatibilities:
  - FARGATE
ExecutionRoleArn: !GetAtt ECSTaskExecutionRole.Arn
ContainerDefinitions:
  - Name: backend-service
    Image: "ECR-URI"
    PortMappings:
      - ContainerPort: 8080
    Environment:
      - Name: DB_HOST
        Value: "postgres-db"
      - Name: DB_PORT
        Value: "5432"
      - Name: DB_USER
        Value: "myuser"
      - Name: DB_PASSWORD
        Value: "mypassword"
      - Name: DB_NAME
        Value: "demo_db"

  - Name: postgres-db
    Image: "postgres:latest"
    PortMappings:
      - ContainerPort: 5432
    Environment:
      - Name: POSTGRES_PASSWORD
        Value: "mypassword"
      - Name: POSTGRES_USER
        Value: "myuser"
      - Name: POSTGRES_DB
        Value: "demo_db"
```

Moreover, in our task definition, we selected Fargate as the capacity option. Fargate is a technology that serves as the infrastructure layer for ECS, implementing a PaaS model. Developers only need to specify the required resources, and Fargate automatically manages their allocation, scaling efficiently to handle high-demand peaks. This approach simplifies the deployment process, allowing developers to focus on application logic rather than spending time on manual resource configuration, which is automatically handled by AWS.

On the other hand, Fargate does not allow for deep system customization, but this is outside the scope of this thesis. However, the trade-off between flexibility and ease of use makes Fargate an ideal choice for deploying containerized applications quickly and

efficiently.

This marks the final stage of our deployment pipeline. Once this step is completed, our tested application will be fully operational, and the latest code changes will be merged into the testing branch.

Chapter 4

Validation

This chapter presents the results of this thesis project, focusing on the analysis of the AI-driven test generation solution and its application in a real-world scenario. We will evaluate the effectiveness of the developed approach and compare the performance of OpenAI's two most widely used GPT models, 4o and 4o-mini, based on the selected metrics. The evaluation will be conducted on two case studies: a medium-sized Spring Boot microservice without an existing test suite and a project that includes test suites manually developed by human developers.

4.1 Evaluation Metrics

In order to demonstrate the potential of AI-driven test generation techniques in improving software quality and development productivity, we conducted a series of tests and extracted results based on the most important quality testing metrics. We used common coverage metrics, specifically line, branch, method, instruction, and complexity coverage. Additionally, to assess the quality of the generated tests, we performed mutation analysis by injecting bugs into the codebase and testing the project's test suite. This approach ensures that the tool not only generates tests that achieve high coverage but also effectively validates their fault-detection capability.

As discussed in Chapter 2, mutation testing ensures that the test suite effectively catches faults in the code by comparing test suite results against the original and mutated code versions. Mutation testing provides an objective measure of test suite quality.

The two key mutation analysis metrics used to validate the results are:

- **Mutation Coverage**: This metric represents the ratio of mutants killed by the test suite to the total number of mutants injected into the codebase, including those not covered by any test cases. High mutation coverage indicates that the test suite is effective in identifying potential bugs.

- **Test Strength**: This metric measures the effectiveness of the test suite by calculating the ratio of mutants killed to the number of mutants actually covered by the tests. It evaluates how well individual test cases detect various types of mutations, such as those targeting different input space partitions.

Mutation analysis was conducted using a Maven plugin called PIT. By adding this plugin to the ‘pom.xml‘ file, we can inject bugs, run tests, and generate statistical data for the analysis.

4.2 Grafana

To visualize and track this data more intuitively, we used Grafana. Grafana is an open-source analytics and interactive visualization web application that enables real-time monitoring and interaction with data through customizable dashboards. It allows for data exploration from multiple sources, performing complex queries and transformations. Moreover, Grafana allows an easy way to share dashboards within the team to foster collaboration and transparency. For our needs, we used this tool to display data locally, but it can also be deployed on AWS as a Docker container. We chose Grafana for our solution to effectively implement the CALMS principles of Measurement and Sharing of information. Its ability to provide real-time insights and a facilitate collaboration, made it the ideal choice for our monitoring needs.

A dedicated dashboard was created for the test generation process, providing insights at different levels of granularity, including the entire project and individual packages.

Figure 4.1 and Figure 4.2 provide an overview of the feedback loop, illustrating how test failures evolve across different iterations, with the x-axis representing the n-th iteration. The first chart displays the number of failing tests in green and compilation errors in yellow, while the second chart represents the trend of the error rate over multiple iterations.

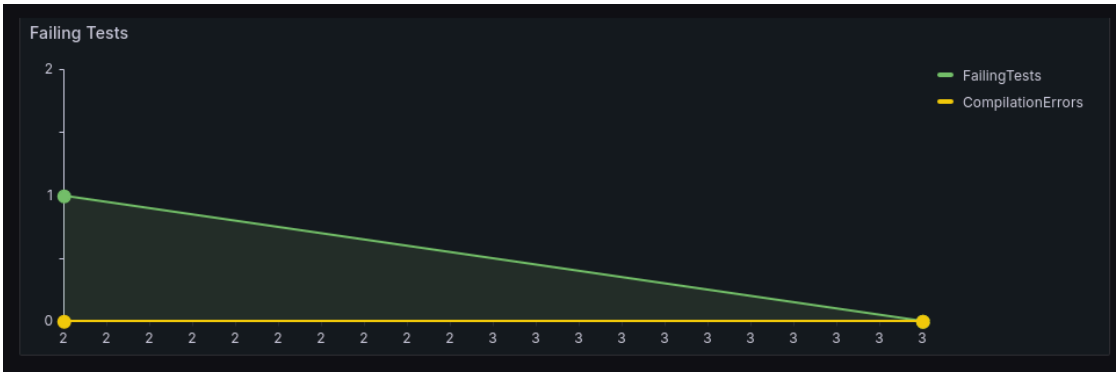


Figure 4.1. Failing tests and compilation errors.

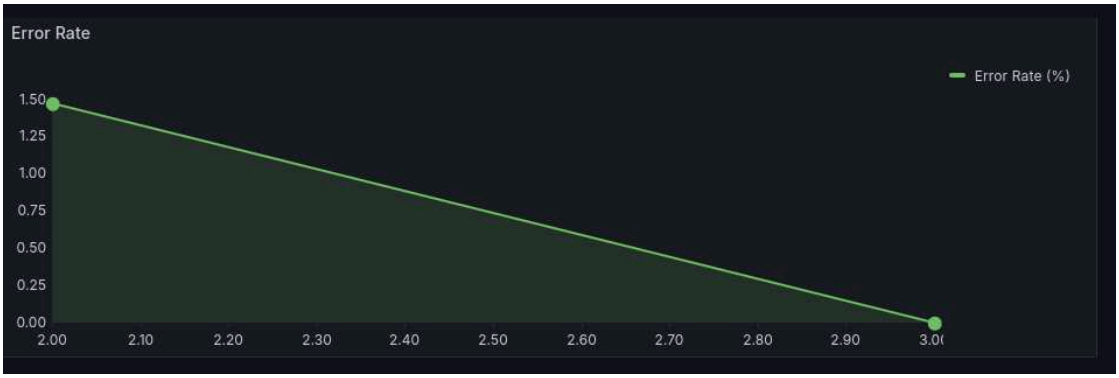


Figure 4.2. Error rate trend.

The second section of the dashboard is presented in Figure 4.3. This section features multiple gauge charts, each representing a different coverage metric, along with a pie chart displaying the percentage of mutation coverage. It can be configured to show data for specific packages within the project and helps in identifying weaker areas in the test suite at the package level. Similarly, Figure 4.4 presents the same type of data but at the project level, offering a comprehensive view of test coverage and mutation analysis throughout the codebase.

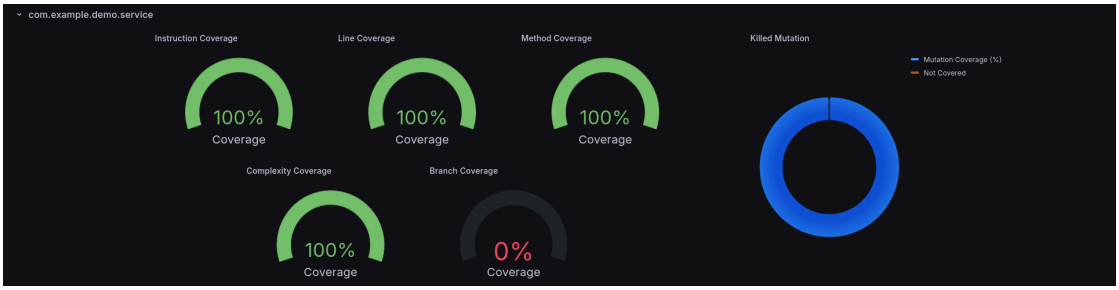


Figure 4.3. Package-Level Coverage and Mutation Analysis

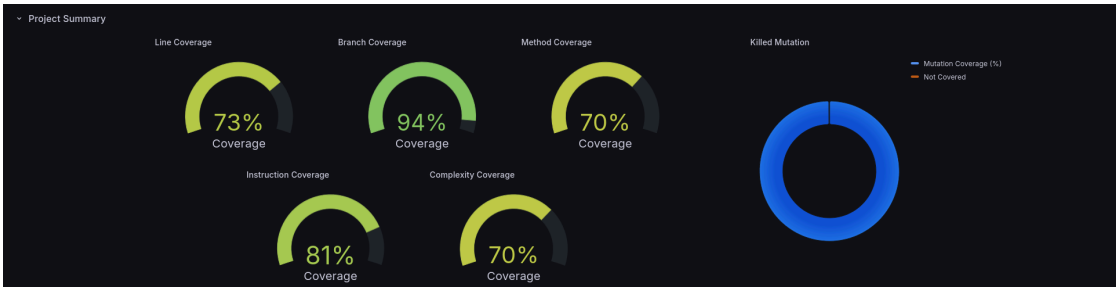


Figure 4.4. Project-Level Coverage and Mutation Analysis

The final section, shown in Figure 4.5, provides a more detailed analysis of coverage and mutation testing across different granularity levels. The upper chart visualizes coverage metrics at the class level using a bar chart, with the coverage percentage displayed on the y-axis, while the lower chart illustrates mutation coverage and test strength for each package within the main project. This breakdown helps in identifying areas with lower test effectiveness and provides insights into the distribution of code quality across different parts of the project.

Figure 4.5. Class and Package-Level Coverage and Mutation Analysis

4.2.1 Testing Methodology

The process for obtaining the results follows a set of rules to ensure it is consistent, well-structured and repetitive in order to facilitate the comparison. These are:

1. The entire project is loaded into a clean repository with a single initial commit, without any existing tests.

2. Compilation errors are addressed manually by the developer.

3. Errors in test generation are first analyzed to determine if they result from the test setup (as the model provides comments when developer intervention is required; in such cases, tests are fixed manually). If the issue persists, a single attempt to fix the test using the model is made. If the test still fails, the test is removed.

We will assess the results from three key perspectives: Feedback Loop, Mutation Analysis, and Project Summary. The first perspective helps determine whether the tools are a viable option for an AI-driven testing approach by examining their usage, common errors, and the level of developer intervention required. The second perspective evaluates the quality of the generated test suite using mutation testing metrics. Finally, the third perspective provides an overview of coverage across the project.

4.2.2 Evaluated Scenario: Comparing GPT Models on an Untested Project

First case study focused on applying test generation on a project lacking of test suites in order to compare performances of the two currently most used GPT models on writing unit tests. As mentioned earlier in Chapter 2, Section 2.3.4, we identified certain limitations in using GPT-4o-mini due to its reduced versatility and reasoning capabilities compared to GPT-4o. This outcome was expected based on OpenAI's documentation, but it could have been mitigated through fine-tuning. However, we did not apply this technique in this scenario and instead used the base model.

Let us provide a brief overview of the project. It is a backend application designed to manage product orders placed by users. Built with Spring Boot and Java 17, the project follows the Controller-Service-Repository pattern to maintain clean code, simplify testing, and ensure a clear separation of concerns. The data access layer is implemented using Spring Data JPA. Four key entities have been defined: User, Order, Product, and OrderItem. Additionally, a bug was intentionally introduced into a utility class method to test the ability of the test generation to detect it. Specifically, the bug would araise when performing a GCD operation on two zero values (GCD(0, 0)), which is typically defined as 0 but it is not handled by the code.

Feedback Loop

GPT-4o

1. First iteration discovered the bug by providing a test for that particular case which failed, manual fix inside the code to handle the scenario.
2. The loop concluded, with no further test failures.

GPT-4o-mini

1. The first iteration failed due to compilation errors caused by missing imports, which were manually added.
2. Second iteration discovered the bug by providing a test for that particular case which failed, manual fix inside the code to handle the scenario.
3. A test case invoked an undeclared method within a custom class of the project, removed manually.
4. The loop concluded, with no further test failures.

Mutation Analysis

Metric	GPT-4o	GPT-4o-mini
Mutation Coverage	100%	100%
Test Strength	100%	98%
Line Coverage	97%	97%

Table 4.1. Mutation Analysis Comparison

Project Summary

Metric	GPT-4o	GPT-4o-mini
Generated Tests	68	70
Time for Generation	4m 20s	3m 6s
Method Coverage	70%	70%
Line Coverage	73%	74%
Branch Coverage	94%	94%

Table 4.2. Comparison of Project Summary Metrics

Both models successfully identified the injected bug, demonstrating a high level of domain exploration in their input. Moreover, GPT-4o-mini appears to generate tests faster but requires more developer intervention to adapt test cases through multiple feedback loops. In fact, the testing phase often fails due to missing or incorrect imports, despite having provided the models with all the necessary code dependencies inside our project. While mutation analysis and primary code coverage metrics for the whole project indicate similar performance, with strong method, line, and branch coverage. The perfect mutation coverage confirms the validity of the generated tests.

4.2.3 Evaluated Scenario: Comparing GPT Models on an Tested Project

The second case study focused on applying test generation to a project with an existing test suite, aiming to compare the performance of the two most widely used GPT models for writing unit tests and evaluate those against tests manually written by humans.

Let us provide a brief overview of the project. It is a REST endpoints backend application without UI designed to manage articles and authors. Built with Spring Boot and Java 21, this project also follows the Controller-Service-Repository pattern

as well as the data access layer is implemented using Spring Data JPA. Three key entities have been defined: Article, Author and LoggedRecord. The project already has a comprehensive test suite covering many classes, though not all tests are unit tests since many are integration tests. To facilitate comparison, we focused test generation solely on the service layer, which is the only part with unit tests, and excluded the existing tests that were not relevant to our analysis.

Feedback Loop

GPT-4o

1. The first iteration failed due to compilation errors caused by ambiguity in a method call. The model incorrectly mocked a result using the Object() class instead of the project's custom entity, leading to confusion about which method was being referenced.
2. The loop concluded, with no further test failures.

GPT-4o-mini

1. The first iteration failed due to compilation errors caused by the missing of all imports of java.lang package and invoked an unimplemented method inside a project's custom class
2. Three tests failed, and the system was tasked with fixing these issues.
3. One test was fixed by the model, while another was corrected by the developer following the model's request due to an incorrect setup in the generated method. One test remained failed and was subsequently removed.
4. The loop concluded, with no further test failures.

Mutation Analysis

Metric	GPT-4o	GPT-4o-mini	Human-coded
Mutation Coverage	89%	58%	74%
Test Strength	89%	76%	100%
Line Coverage	100%	79%	71%

Table 4.3. Mutation Analysis Comparison

Project Summary

Metric	GPT-4o	GPT-4o-mini	Human-coded
Generated Tests	17	16	15
Time for Generation	1m 15s	57s	-
Method Coverage	100%	95%	90%
Line Coverage	100%	77%	70%
Branch Coverage	88%	50%	38%

Table 4.4. Comparison of Project Summary Metrics (Service Layer)

The same discussion can be applied to the features of GPT-4o and GPT-4o-mini, such as latency, the extent of developer involvement in feedback loops, and the most frequent compilation errors. Our focus will be on comparing the results between the generated tests and those created by developers.

Let us start with mutation analysis comparison. The model GPT-4o achieves the highest mutation coverage indicating that it catches more faulty code mutations compared to both the human-coded tests and the GPT-4o-mini model. However, this does not ensure that the tests are effectively identifying all potential faults, particularly in more critical or subtle areas. On the other hand, test strength is higher for the human-coded tests comparing to the others. It means that these tests are more effective at killing mutants even though its overall mutation coverage is lower.

Regarding the coverages for the service layer, which is the only part covered by the unit testing, the metrics are more effective for those generated by the GPT-4o model. In fact, line coverage is perfect for the GPT-4o model while GPT-4o-mini and human-coded tests are not as good. This suggests that GPT-4o runs through more of the code but does not necessarily test it better since test strength is higher in human-coded tests. As well as for line coverage, also method coverage has higher metrics in GPT-4o for method coverage while the other two are slightly behind. Lastly, branch coverage highlights that tests generated by OpenAI's most advanced model cover more decision-making logic.

Chapter 5

Conclusion

This thesis has explored a possible solution to support developers in writing unit tests. We started explaining why testing is an expensive and time-consuming phase of the SDLC and presenting techniques such as static analysis and dynamic test generation to speed up the process. Then, we described the designed system based on a LLM for handling test generation, in our case we opted for OpenAI most used models. Finally, in Chapter 4 we presented the collected results. In this Chapter we will analyze those.

Performances

Our goal was to create a tool to be used alongside the testing phase by developers without implementing an automated way to query a LLM for generating test. Our focus was on the interaction with the developers who is conscious of what is generated each time and who should approve or reject. We will first analyze the quality of the generated tests. Looking at the results we can see that both GPT-4o and GPT-4o-mini models produced a test suite with effective test cases. We obtained high coverages metrics from models for both the case studies highlighting a higher values for the generated test suites compared with the human-coded test suite. However, we found that the covereges metrics are not enough to determine the quality of a test suite. Mutation analysis results shows that mutation coverage is higher for the GPT-4o model meaning that it tests are capable of identifying more mutants than the human-coded test suite but test strength is better for the human-coded resulting in a test suite more focused on testing particular test cases than the generated one. We can see that quality of the models' work is comparable with the quality of a human's product.

Ease of Use

Let us now briefly analyze the ease of use of the tool. We cannot define a precise duration for the whole generation process due to the fact that there are interactions with the developers which may lead to delays. We highlighted the time required for generating from scratch a test suite with a full load of the project. These times are comparable between the two models. However, most of the time spent by developers is used in the feedback loop whenever there are faulty tests to be fixed. Here, the GPT-4o models has better performance compared to the other model because it produces less error-prone test suite requiring less interactions with the developers resulting in less time spent for fixing those.

Pricing

Now, we turn our attention to the cost analysis of the process. The effective costs depends on the usage so we will propose an esteem based on the costs of the deployment and testing of the system. The pay-as-you-go model offered by AWS helps in our goal to reduce costs for the infrastructure also due to the fact that the AWS Free Tier offers monthly a set of resource to be consumed before starting to pay leading this expenses to be negligible. The majority of the costs are attributed to the use of the models. GPT-4o model is way more expensive than the GPT-4o-mini based on token usage with the pricing per 1M token shown in Chapter 2. For our esteem, we performed calculations on the first case study. It is a basic analysis using a single full load of the project to generate the test suites and without considering possible faulty test generated that would lead to a fixing phase. In addition to the source code of the class, we have to consider also its dependencies and the static part to instruct the model in the total amount of tokens used for the input resulting in a total of 16136 tokens. Instead, to determine the number of output tokens, we analyzed the generated test suites, resulting in a total of 7840 tokens. Based on the prices shown in Table 2.2, the total cost amounts to $0.12 for GPT-4o and $0.007 for GPT-4o-mini. We acknowledge that this is a very limited estimate of our system's usage as it considers only a simple scenario. However, it highlights that the GPT-4o model is 1567% more expensive than the GPT-4o-mini which, in more realistic scenarios having a whole team sharing a single pipeline leading to an high demand, would lead to significant costs. In order to saves those costs and obtain the same performances of the most expensive of the OpenAI models using GPT-4o-mini we can perform fine-tuning.

Future Works

Until now, we have analyzed the system focusing on the three main aspect of performances, ease of use and pricing. We have seen that both OpenAI models generate test suites with comparable metrics to evaluate the effectiveness. However, GPT-4o generates test suites without an high need of human interaction leading to a better ease of use. We prioritized the system's usability over pricing, as intense usage was not required. Now we will analyze possible solution to solve these problems.

According to the OpenAI documentation, fine-tuning technique let us obtain higher quality in results than using simple few-shots learning by training the model with more examples that would not fit inside a prompt and leading to lower latency in requests and token savings due to shorter prompts. [8] Performing fine-tuning on the GPT-4o-mini model would help us in creating a new custom model that is more capable of generating test suites. Another cost-saving solution would be to implement a custom Java parser to analyze the code before generating the request prompt. This parser would examine the source code of the committed file and break it down into Java language objects. By doing so, we could avoid constructing a prompt containing all the classes of source code and dependencies when only a small portion is needed by the model. Implementing a parser would shift the system's granularity from the class level to the method level, reducing the number of tokens used and, consequently, lowering costs.

Moreover, the application currently lacks integration testing generation. As we discussed earlier, having a strong test suite with unit and integration tests to cover possible faults in our application is crucial in order to perform regression testing whenever a new feature is added to the codebase or some already existing one is updated. However, we did not take this idea into account for the current system that focuses exclusively on generating unit tests. This can be considered as an additional feature for future versions of the system due to easily configurable environments for integration testing offered by the Spring Boot framework. It provides a simple way to perform requests to exposed endpoints while also interacting with a lower data layer, such as an H2 in-memory database, helping to prevent tests from polluting the actual database.

Bibliography

[1] Amazon Web Services. The difference between agile and devops, n.d. Accessed: 10 February 2025.

[2] Website (techbeacon.com) HPE Micro Focus Capgemini, Sogeti. Proportion of budget allocated to quality assurance and testing as a percentage of it spend from 2012 to 2019. Statista, 2019. Accessed: January 13, 2025.

[3] R.E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.

[4] Klarlund N. Godefroid, P. and K. Sen. Dart: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 05)*, pages 213–223, 2005.

[5] P. Godefroid. Automating software testing using program analysis. *IEEE Software*, Sept.-Oct. 2008.

[6] Won Kim. Cloud computing: Today and tomorrow. *J. Object Technol.*, 8(1):65–72, 2009.

[7] M. Mythily. An extensive review of spring boot testing based on business requirements of the software. In *Proceedings of the 2023 4th International Conference on Smart Electronics and Communication (ICOSEC)*. IEEE, September 2023.

[8] OpenAI. Fine-tuning guide, 2024. Accessed: March 3, 2025.

[9] OpenAI. Gpt models documentation, 2025.

[10] Amazon Web Services. Aws codebuild user guide: Concepts, 2025. Accessed: 23 January 2025.

[11] SonarSource. Sonarqube server documentation, 2025. Accessed: 16 January 2025.

[12] Spring Boot. Spring Boot - Spring Framework, 2025. Accessed: 2025-03-12.

[13] Xu W. Kent M. Thomas L. Xu, D. and L. Wang. An automated test generation technique for software quality assurance. *IEEE Transactions on Reliability*, 64(1):247–268, 2015.