David Jorge Garcia Mendes

Bachelor in Computer Science

Automated Testing for Provisioning Systems of Complex Cloud Products

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Advisers: Miguel João, Cloud Systems Architect, OutSystems
João Lourenço, Assistant Professor, NOVA University of Lisbon

Examination Committee

Chairperson: Prof. Nuno Preguiça, FCT-NOVA
Members: Prof. João Lourenço, FCT-NOVA
Prof. João Pascoal Faria, FEUP

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Automated Testing for Provisioning Systems of Complex Cloud Products

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

In loving memory of my grandmother, Deonilde Mendes.

Acknowledgements

First of all, I would like to thank the Faculty of Sciences and Technology from the New University of Lisbon and, in specific, the Informatics Department for providing me with the building blocks from which I can start building my professional career. Likewise, I want to thank OutSystems for providing a scholarship for this dissertation and allowing me to develop it in an environment with all the conditions to succeed.

I also want to thank my advisers, Miguel João and João Lourenço, for mentoring me through this work.

A big thanks to all my friends for supporting me and comprehending why I could not join them when work was urgent, and delivery dates got closer.

In particular, I would like to thank my thesis companions and OutSystems colleagues, that I can now proudly call my friends. Francisco, António, Ricardo, Lara, Miguel, Madeira, Michael, João, Calejo, Nuno, Joana, Magalhães and Mariana thanks for making my days always entertaining and for being a reason for me to want to go to work everyday. Thank you for the always exciting lunch conversations and for the breaks over at the pool table.

Lastly, I must thank my parents for giving me all the opportunities I needed to get this far. I don't think I could have asked for a better support at home and will be forever thankful. Also want to thank my sister for being such a nuisance every time I got home, quickly making me forget about the workday and freeing my mind from work topics.

Thank you all.

"No problem can be solved until it is reduced to some simple form. The changing of a vague difficulty into a specific, concrete form is a very essential element in thinking."

- J. P. Morgan

Abstract

Context: The proliferation of cloud computing enabled companies to shift their approach regarding infrastructure provisioning. The uprising of cloud provisioning enabled by virtualisation technologies sprouted the rise of the Infrastructure as a Service (IaaS) model. *OutSystems* takes advantage of the IaaS model to spin-up infrastructure on-demand while abstracting the infrastructure management from the end-users.

Problem: *OutSystems'* orchestrator system handles the automated orchestration of the clients' infrastructure, and it must be thoroughly tested. Problems arise because infrastructure provisioning takes considerable amounts of time, which dramatically increases the feedback loop for the developers. Currently, the duration of the orchestrator tests hinder the ability to develop and deliver new features at a desirable pace.

Objectives: The goals of this work include designing an efficient testing strategy that considers a microservices architecture with infrastructure provisioning capabilities while integrating it in a Continuous Integration (CI)/Continuous Deployment (CD) pipeline.

Methods: The solution applies multiple testing techniques that target different portions of the system and follow a pre-determined test distribution to guarantee a balanced test suite. The strategy was tested against a set of prototypes to evaluate its adequacy and efficiency. The strategy definition focuses on mapping the type of errors that each test level should tackle and is, therefore, independent of the employed technologies.

Results: The devised strategy is integrated in a CI/CD pipeline and is capable of comprehensively test the created prototypes while maintaining a short feedback loop. It also provides support for testing against commonly found errors in distributed systems in a deterministic way.

Conclusions: The work developed in this dissertation met the outlined objectives, as the developed strategy proved its adequacy against the developed prototypes. Moreover, this work provides a solid starting point for the migration of the orchestrator system to a microservices architecture.

Keywords: Infrastructure Provisioning, Microservices, Automated Testing, Testing Strategy, Continuous Integration, Continuous Deployment

Resumo

Contexto: A proliferação da computação em *cloud* permitiu que as empresas mudassem a sua abordagem em relação ao aprovisionamento de infraestrutura. A emergência do aprovisionamento em *cloud* possibilitado pelas tecnologias de virtualização possibilitou a aparição do modelo IaaS. A *OutSystems* aproveita esse modelo de serviço, para criar infraestrutura de forma automatizada, abstraindo a gestão da infraestrutura dos clientes.

Problema: O sistema de orquestração da *OutSystems* coordena a orquestração automatizada da infraestrutura dos clientes, e deve ser testada. Os problemas surgem porque a criação da infraestrutura leva um tempo considerável, o que aumenta o ciclo de feedback para os programadores. Atualmente, a duração dos testes do sistema de orquestração dificulta a capacidade de desenvolver e fornecer novas funcionalidades a um ritmo desejável.

Objetivos: Os objetivos deste trabalho incluem a definição de uma estratégia de testes eficiente que considere uma arquitetura de microsserviços para o aprovisionamento de infraestrutura e a integração da mesma numa *pipeline* de CI/CD.

Métodos: A solução aplica várias técnicas de teste que visam diversas partes do sistema e seguem uma distribuição predefinida para garantir uma bateria de testes equilibrada. A estratégia foi aplicada a um conjunto de protótipos de forma a avaliar a sua aplicabilidade e eficiência. A estratégia foca o mapeamento dos erros que cada nível de teste deveria endereçar e, por isso, é independente das tecnologias utilizadas.

Resultados: A estratégia criada está integrada numa *pipeline* de CI/CD e é capaz de testar compreensivamente os protótipos criados, mantendo um ciclo de feedback reduzido. A estratégia também suporta a definição de testes contra erros comummente encontrados em sistemas distribuídos, de forma determinística.

Conclusões: O trabalho desenvolvido nesta dissertação cumpriu com os objetivos delineados, visto que a estratégia provou a sua aplicabilidade nos protótipos desenvolvidos. Para além disso, o trabalho desenvolvido fornece um sólido ponto de partida para a migração do sistema de orquestração para uma arquitetura de microsserviços.

Palavras-chave: Aprovisionamento de Infrastrutura, Microsserviços, Testes Automatizados, Estratégia de Testes, Integração Contínua, Implantação Contínua

Contents

List of Figures xvii

List of Tables xix

Acronyms xxi

1 Introduction 1
- 1.1 Context and Description . . . 1
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 2
- 1.4 Solution . . . 2
- 1.5 Key Contributions . . . 3
- 1.6 Structure . . . 3

2 Background 5
- 2.1 OutSystems . . . 5
  - 2.1.1 OutSystems aPaaS Offering . . . 5
  - 2.1.2 OutSystems Cloud Architecture . . . 6
  - 2.1.3 OutSystems Orchestrator Architecture . . . 7
- 2.2 Testing Approaches . . . 7
  - 2.2.1 Infrastructure Testing . . . 10
  - 2.2.2 Microservices Testing . . . 11
    - 2.2.2.1 Pre-Production Testing . . . 12
    - 2.2.2.2 Production Testing . . . 20
- 2.3 Continuous Integration and Deployment . . . 23
  - 2.3.1 Continuous Integration . . . 23
  - 2.3.2 Continuous Deployment . . . 25
  - 2.3.3 Continuous Integration/Deployment Challenges in Microservices 25

3 Related Work 27
- 3.1 Testing Programmable Infrastructure with Ruby . . . 27
- 3.2 Spotify Testing Strategy for MicroServices . . . 28
- 3.3 SoundCloud Consumer-Driven Contract Testing . . . 29

3.4 Microservices Validation: Mjolnirr Platform Case Study 30
3.5 Waze Deployments with Spinnaker 32
3.6 Summary 33

4 Implementation **37**
4.1 Continuous Integration/Deployment Environment 38
4.1.1 Environments Architecture 40
4.2 Chef Testing Strategy 41
4.2.1 Implemented Test Levels 44
4.2.1.1 Static Analysis 44
4.2.1.2 Unit Testing 45
4.2.1.3 Integration Testing 47
4.2.2 Pipeline 49
4.3 Microservices Testing Strategy 50
4.3.1 Implemented Test Levels 51
4.3.1.1 Unit Testing 51
4.3.1.2 Integration Testing 53
4.3.1.3 Consumer-Driven Contract Testing 56
4.3.1.4 Component Testing 59
4.3.1.5 End-to-End Testing 60
4.3.2 Pipeline 61
4.3.2.1 Functional Tests 61
4.3.2.2 Contract Tests 62
4.3.2.3 Deploy to Pre-Production Environment 65
4.3.2.4 End-to-End tests 66
4.3.2.5 Deploy to Production Environment 66

5 Evaluation **67**
5.1 Chef Evaluation 68
5.2 Microservices Evaluation 69
5.2.1 Test Distribution 69
5.2.2 Virtualisation Usage 71
5.2.3 Feedback Loop 72
5.2.4 Contract Testing 73

6 Conclusions **75**
6.1 Contributions 76
6.2 Future Work 76

Bibliography **79**

A Appendix **85**

List of Figures

2.1 OutSystems cloud starting configuration [54]. 6
2.2 OutSystems cloud starting configuration physical architecture [53]. 7
2.3 Pact consumer testing [30]. 15
2.4 Pact provider testing [30]. 15
2.5 In-process component testing schematic [73]. 16
2.6 Out-of-process component testing schematic [73]. 17
2.7 Ideal test distribution pyramid [73]. 19
2.8 Testing on the deployment phase of the production rollout [22]. 21
2.9 Deployed version and released version of software [22]. 22
2.10 Differences in release strategies [58]. 24
2.11 Continuous integration pipeline [31]. 24
2.12 Continuous deployment pipeline [19]. 25

3.1 Microservices testing honeycomb [72]. 29
3.2 SoundCloud's continuous integration pipeline with Pact [48]. 30
3.3 Waze pipeline template for continuous deployment [40]. 33

4.1 CI/CD environment architecture. 40
4.2 Pre-production and Production environment architecture. 41
4.3 Download executable recipe mapped to control flow graph. 43
4.4 Service install recipe mapped to control flow graph. 44
4.5 Chef pipeline. 49
4.6 Microservice architecture schematic. 51
4.7 Unit tests coverage. 52
4.8 Integration tests coverage. 53
4.9 Contract tests coverage. 57
4.10 Pact file documentation. 58
4.11 Component tests coverage. 59
4.12 End-to-End tests coverage. 60
4.13 Microservices pipeline. 61
4.14 Functional tests in detail. 61
4.15 Pact Matrix. 62
4.16 Provider contract tests in detail. 63

4.17 Consumer contract tests in detail. 64
4.18 Provider Production verification job in detail. 64
4.19 Deploy to Pre-Production environment in detail. 65
4.20 Deploy to Production environment in detail. 66

5.1 Prototype test distribution. 70
5.2 Microservices pipeline with time statistics. 72

A.1 Microservices detailed pipeline. 85

List of Tables

5.1 Detailed Chef test distribution. 68
5.2 Detailed Microservices test distribution. 70

Acronyms

AMI	Amazon Machine Image.
aPaaS	Application Platform as a Service.
API	Application Programming Interface.
AWS	Amazon Web Services.
CD	Continuous Deployment.
CI	Continuous Integration.
DSL	Domain Specific Language.
EC2	Elastic Cloud Compute.
GUI	Graphical User Interface.
IaaS	Infrastructure as a Service.
IaC	Infrastructure as Code.
PoC	Proof of Concept.
RDS	Relational Database Service.
RoI	Return on Investment.
SDK	Software Development Kit.

SUT System Under Test.

TDD Test Driven Development.

URI Uniform Resource Identifier.

VPC Virtual Private Cloud.

Chapter 1

Introduction

This chapter aims at providing the required context for understanding the problem at hand and describing the motives that drove the need for devising a solution. It will also underline the goals of this dissertation and the fundamental contributions. The chapter concludes by highlighting the remaining structure of the document.

1.1 Context and Description

The world of Dev-Ops[1] has been subject to constant paradigm changes in the last decade. The evolution of infrastructure-related technologies enabled a faster provisioning space, and the notion of infrastructure has also been continually evolving and mutating. Infrastructure is no longer bare metal. The concept has been abstracted, and the infrastructure definition has become much broader [85]. Infrastructure related technologies such as virtualisation made it so provisioning can take minutes instead of days like it once did [89].

Infrastructure provisioning is a crucial element of the service model OutSystems offers. It relies on IaaS vendors to provision infrastructure it does not own in a highly automated manner. Infrastructure provisioning systems like the ones OutSystems built for its Application Platform as a Service (aPaaS) product (to be discussed in more detail in section 2.1.1) suffer from several challenges. This type of systems are incredibly complex, span over multiple technologies and interface with many external systems.

Over 90% of the infrastructure provisioning operations in the OutSystems orchestrator system have some infrastructure pre-requirement to run successfully, which has mostly to do with the nature of cloud provisioning systems. With the intrinsic complexity

[1] "A set of practices that automates the processes between software development and IT teams, in order that they can build, test, and release software faster and more reliably ... It's a firm handshake between development and operations that emphasises a shift in mindset, better collaboration, and tighter integration." [81]

of the solution, and like every software system, testing is a necessity and a fundamental approach to keep moving forward at an increasingly faster pace, in an industry that demands it, to stay ahead of the curve.

1.2 Motivation

Currently, testing the infrastructure provisioning software system consists mainly of end-to-end tests[2]. These tests are costly, both in the time they take to run and the cost of spinning up real infrastructure to perform the tests. Provisioning real infrastructure can take several minutes. Thus, using a standard testing approach is extremely challenging as the setup time of most tests would be unacceptable.

To allow the fast evolution of the OutSystems aPaaS provisioning system, and the change of architecture that it requires, a new testing strategy needs to be devised to accompany this evolution.

1.3 Objectives

This work aims at devising an efficient strategy to automatically test cloud provisioning systems both in terms of the effort needed to write the tests as well as the efficiency when running them. It is imperative to explore methods that allow to detect errors sooner and shorten the feedback loop for the developers. It is also relevant that the devised strategy permits the injection of errors commonly found in distributed systems, such as latency or servers unresponsiveness, to test the robustness of the system.

Beyond infrastructure testing and due to an architectural change that will take place in the orchestrator system, while moving from a monolithic to a microservices architecture, the devised strategy must consider the new architecture. This change will affect the testing methods but also the entire lifecycle of the application, from development to production.

Furthermore, continuous integration and deployment techniques must be incorporated into the definition of the solution to enable faster development and deployment cycles, and automatically assert the quality of the delivered services.

1.4 Solution

The solution devised in this dissertation was tested against a set of prototypes, developed alongside the strategy definition, that emulate the functionality and limitations of the orchestrator system to prove its applicability in the OutSystems context.

[2] End-to-end testing involves testing a user workflow from beginning to end, Section 2.2.2.1 covers this topic in more detail.

The solution was able to reduce the exacerbated feedback loop by applying a pyramid distribution to the implemented test levels and by using virtualisation to replace some dependencies in the testing stage.

The created strategy focused on having a clear architectural notion of the components under test to provide a better mapping between the components and the errors to detect at each level. The strategy encompasses multiple types of tests to guarantee coverage of the system, and prioritises using lower-level tests whenever possible to provide fast, focused and reliable feedback.

The CI/CD pipeline implemented provides support for the application of the strategy and utilises patterns and techniques to reduce the feedback loop and improve the reliability of the deployments.

1.5 Key Contributions

By the time of writing this dissertation, there is a running prototype that includes a set of microservices fully integrated into a CI/CD workflow. The microservices functionality incorporates a small subset of cloud provisioning operations required to support the OutSystems aPaaS and exercises the same dependencies of the original system, like IaaS vendors for instantiating infrastructure.

Beyond describing the prototype, the dissertation also documents the development and deployment methodologies of these microservices, alongside the appropriate testing strategy that considers the infrastructure dependencies and presents solutions to reduce the feedback loop, while assuring an adequate test coverage for the developed components.

The definition of the testing strategy helps in the upcoming migration of the OutSystems orchestrator and the lessons learned in the development of the prototype, and the strategy itself will help to design the new architecture to achieve better test-ability.

This work lays the foundation for that migration and provides an appropriate strategy for the OutSystems aPaaS orchestrator context. The testing strategy is not bound to the OutSystems context, as it aims to be as generic as possible. Any other microservices-based systems can also apply the same concepts presented in this dissertation, being particularly useful for those whose dependencies significantly increase the overall cost of testing.

1.6 Structure

The remainder of the document is structured as follows:

- Chapter 2 - Background: focuses on the OutSystems segment directly related to the dissertation theme, testing approaches that address the problematic and continuous integration and deployment techniques;

- Chapter 3 - Related Work: focuses on existing work in the area, including infrastructure and microservices testing approaches, as well as continuous integration and deployment pipelines;
- Chapter 4 - Implementation: depicts the implemented testing strategy and the supporting components in detail, alongside the lessons learned in the process;
- Chapter 5 - Evaluation: presents the results obtained for this work;
- Chapter 6 - Conclusions: sums up the biggest contributions of the work and identifies future work areas that can improve the solution even further.

Chapter 2

Background

This chapter aims at providing background context concerning topics that are related to the dissertation theme, covering: the OutSystems aPaaS Offering and the underlying architecture, an overview of testing approaches that can be incorporated in the problem's resolution, and continuous integration and deployment methodologies that bring additional benefits and increase the robustness and agility of the software delivery.

2.1 OutSystems

OutSystems is a leading software company that focuses on improving the IT professionals development experience and speed of delivery with the number one low code platform in the market [52]. The OutSystems platform, alongside the developed visual programming language, allows the development of enterprise-grade applications at a higher abstraction level, dramatically boosting the attained productivity [51].

2.1.1 OutSystems aPaaS Offering

The OutSystems aPaaS service model provides customers with the ability to develop and deploy their mobile and web applications without having to worry about maintaining and setting up infrastructure. The service also grants the customers the ability to quickly scale up and down their applications (and the underlying infrastructure) according to the experienced load as well as keep track of the different development and production environments with the LifeTime centralized console. "LifeTime manages the deployment of applications, IT users, and security across all environments" [42].

2.1.2 OutSystems Cloud Architecture

Figure 2.1 represents the initial cloud configuration delivered to the client, which is supported by Amazon Web Services (AWS). This configuration has three environments to which the user can deploy his applications and manage the entire lifecycle within LifeTime [53]. The development environment's primary purpose is to create and develop applications, and is optimized to have a fast feedback loop. Publishing applications to the development environment is faster than to any other environment as this process is optimized for speed, using techniques like differential compilation[15]. The quality environment is where the developers can test the application in an environment that is very similar to production while avoiding the risk of affecting real users. After being tested in the quality environment, the developed applications are published to the production environment where they become available to the end-users.

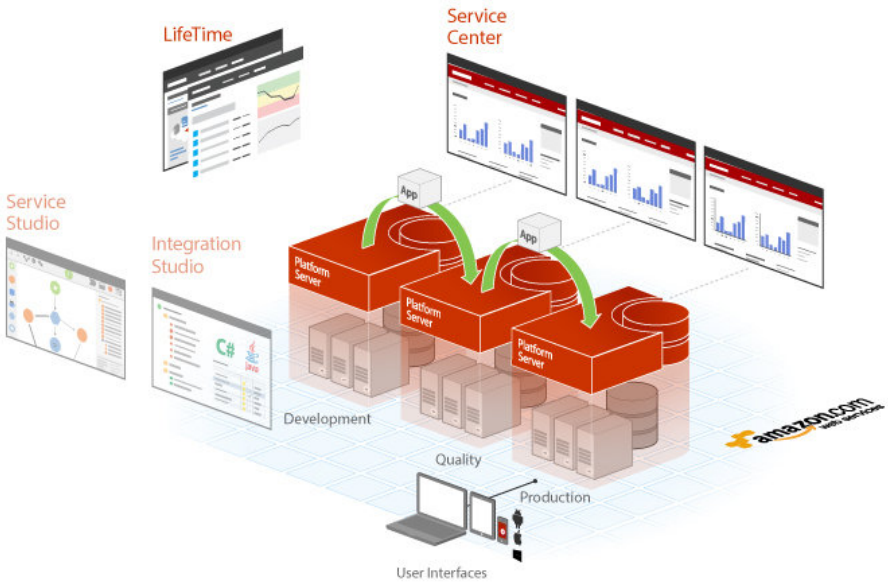


Figure 2.1: OutSystems cloud starting configuration [54].

The delivered configuration is extensible and grants the customers self-provisioning capabilities, which enable customer-specific configurations, like spawning more environments to meet the team's specific business needs. The physical architecture within AWS, represented in Figure 2.2, is comprised of a Virtual Private Cloud (VPC) that grants a logically isolated private network for the instanced environments. The environments are created using Elastic Cloud Compute (EC2) instances connected to multiple database servers to support production and non-production environments. The databases are served by Amazon's Relational Database Service (RDS), and the user can opt by either Microsoft SQL or Oracle for the underlying database engine.

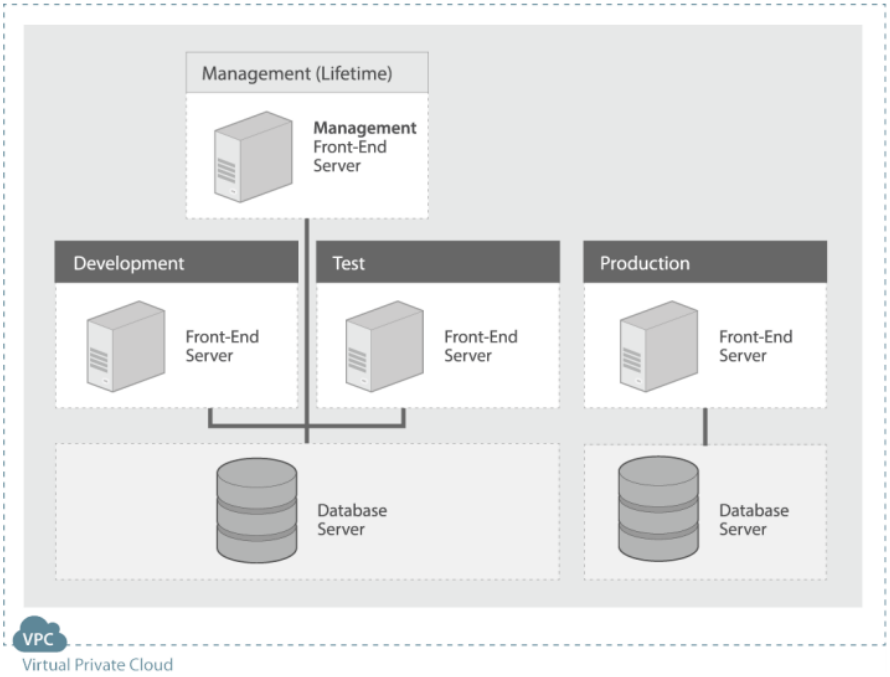


Figure 2.2: OutSystems cloud starting configuration physical architecture [53].

2.1.3 OutSystems Orchestrator Architecture

The basis for the infrastructure provisioning lies in the OutSystems Cloud Orchestrator. This system is responsible for both provisioning the infrastructure and for doing maintenance and upgrade operations that ensure the correct functioning of the customers' environments. Currently, the system comprises multiple subsystems, some of which follow an intricate monolithic design. The monolithic nature of some subsystems is slowly becoming a thorn regarding scalability and maintainability, while also increasing the effort required for new developers to understand and add value in the software evolution.

The problematic subsystems will be broken down into multiple microservices to address those issues. This change will reduce the mentioned problems but also enable faster development cycles, independent scaling and more flexibility on the tech choice to implement each microservice. [44].

2.2 Testing Approaches

There are many misconceptions on what software testing is. Some argue that software testing is an activity in which we check that the software system is defect-free [87], while some see it as the process of evaluating the developed software and verify it meets the defined requirements [86].

Back in 1969, Edsger Dijkstra proclaimed the following:

> "Testing shows the presence, not the absence of bugs (in Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969 [8])".

This quotation perfectly covers the concept of testing. Testing should be seen as an attempt to validate the developed software, but just because all tests developed against a component run without errors that does not mean the component is defect free. It only means that it covers the described scenarios in the test cases.

Therefore, software testing is always a conflict between the effort needed to define the tests and the possible return they may provide by pinpointing software bugs. Moreover, we should strive for understanding where the Return on Investment (RoI) is located for any given System Under Test (SUT), and write tests accordingly.

Beyond that, it is pivotal that test results provide useful feedback to the developer as that is the most significant value of the tests. The information they provide to the developer should be as accurate as possible, meaning that, whenever a test fails, it should allow the developer to determine exactly which piece of code was responsible for the failure, and in result, reduce the amount of time it takes to identify and correct the issue [74].

Furthermore, tests should also aim to be reliable in the sense that we can trust the test result. Flaky tests, whose results vary from one execution to the other, reduce the confidence in their feedback and the confidence in the component itself. Last, but not least, the feedback should be as fast as possible, which means fast test execution times. The faster the developer can identify the problem, the quicker and the cheaper it is to fix it.

Testing should always focus on getting feedback that helps the developer. Still, to get the best possible feedback from testing, it is always necessary to understand the context of the system and adjust the testing approach to the system's singularities.

In order to tackle the challenges of infrastructure testing, specifically within a microservices architecture, different techniques must be used. Due to the complex nature of the architecture, it is necessary to break down the testing problem into several more manageable problems. Multiple techniques exist to address each one and must be used in a complementary manner to assure a complete solution.

Provisioning infrastructure at the scale OutSystems does, lends itself to thorough testing, but thinking at the scale of a business which only provisions infrastructure to deploy and test their application or service, it is not a common practice [59]. There are multiple reasons why the methodologies for testing infrastructure are not standard across the industry. Namely, there is a lack of established tooling to test infrastructure, and there are also issues that arise from a RoI standpoint. In a business model that usually charges by the hour, instantiating the infrastructure needed to run tests can increase the monetary costs of testing at a high pace. Coupled to these costs we also have the hefty time investment needed to instantiate infrastructure to test. Adversely to the difficulties,

the arrival of the Infrastructure as Code (IaC) concept was the main catalyser for the ability to test infrastructure.

Infrastructure as Code

IaC refers to the management and provisioning of infrastructure using a descriptive model, similar to programming scripts, to generate an environment [84]. IaC is based on the principle of idempotence, which means that the same script will always set the environment into the same configuration and that it is possible to execute it multiple times without adverse effects. IaC brings many benefits that derive from the fact that if the infrastructure setup is code, we can also employ some of the same techniques that apply to application code to infrastructure code, such as source control versioning, automated testing, and inclusion in CI and CD pipelines. IaC brought to the fold configuration management and orchestrator systems. These systems use the principles of IaC to provision and manage infrastructure environments.

Configuration management tools are designed to install and configure software on existing servers (e.g., EC2 instances). Tools like Chef [9], which uses a ruby Domain Specific Language (DSL) for writing system configuration files, that are colloquially called "recipes" [2], provide a way to force configurations on multiple servers. Chef is the OutSystems' tool of choice for configuring machines, and it is a fundamental part of the system. For that reason, testing research of machine configuration will focus mainly on the Chef tool.

Orchestration tools are responsible for creating and managing cloud systems, including servers, services, and middleware [80]. There are commercially available orchestration tools like AWS CloudFormation[5] or Google's Cloud Deployment Manager [12] who provide a declarative way to describe and provision infrastructure. In our particular case, OutSystems built their orchestration tool, which among other advantages prevents a vendor lock-in as there is an abstraction layer between the vendor and the orchestration operations [53]. The orchestration operations make use of the available Application Programming Interfaces (APIs) and Software Development Kits (SDKs) provided by the IaaS vendors that expose infrastructure provisioning operations to assemble the demanded infrastructure.

The following subsections focus on showcasing different techniques that are relevant to test the complex infrastructure provisioning system. Firstly are introduced infrastructure-related testing techniques and then the scope changes to microservices testing approaches.

Please note that terminology in the testing area is not uniform. So, the following definitions are not meant to be seen as absolute truths but interpretations of the concepts through the scope of the problem accompanied by critical thinking.

2.2.1 Infrastructure Testing

When we focus on infrastructure testing, we focus mainly on testing the scripts and code developed with configuration management tools and orchestrator systems. Beyond looking at the code itself, since they set up infrastructure, we can always try to assert that the infrastructure is correctly set up, but since that process is slow, we should strive to use other testing techniques that provide much faster feedback.

As we have seen, infrastructure provisioning scripts are code, and so, we can draw a parallel to application code in some phases of testing.

The first phase of testing usually comes in the form of static analysis. With static analysis tools, there is no dynamic execution of the software under test [66]. This type of analysis is usually the first step the code goes through, as it is usually the fastest and can weed out most basic issues. It can, for example, detect unreachable code or unused variables along with style issues that do not conform to the teams pre-determined set of rules. It, usually, translates into running some language-specific analysis to check the validity of the script or code.

Static analysis also encompasses linting. It is widely accepted as a good practice to perform code linting. Linters provide a way to detect possible errors based on known non-optimal constructs(anti-patterns[1]) and also provide a way to uniform the style code of teams, which enables better cooperation. Linting tools are abundant for every programming language, and most have configurable rule applications, so it is possible to set them up so the organisation follows pre-established coding rules.

After static code analysis, there is usually a phase in which we perform unit testing. Unit testing provides a way to quickly test the desired functionality of the smallest part of the SUT. While there is a discussion on what should be the test scope for unit testing, there is a consensus in the fact that they should be fast enough to avoid people refusing to run them [75].

Something particular to infrastructure provisioning scripts is that they are slow by nature because they instantiate real infrastructure in the cloud, which will increase the duration of the tests. That fact is prohibitive to run unit tests at the desired speed and frequency.

Some configuration management tools like Chef [9] have unit testing frameworks (e.g., ChefSpec [10]) that work on the basis that they do not set up real infrastructure but run the tests in a sort of simulation, which drastically increases the test execution speed.

While some tools exist for testing at this level, unit testing for configuration management frameworks main problem lies in the fact that most of these languages are of pure declarative nature and therefore are quite straightforward to read and write (that is one of their advantages). With that in mind, the most typical unit testing scenario would merely

[1] "An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences[4]"

restate the declarative nature of the code, which raises questions to the actual benefits of unit testing at this level.

Since configuration management tools intend to apply a particular script to a server, it is possible to test these scripts in a local sandbox environment. By making use of virtualisation technologies, we can run tests that incur fewer costs and skip the network interactions needed for testing in the cloud providers. Still, these tests take a considerable amount of time but make it possible to assert the correct functionality of the scripts.

Tools like ServerSpec[62] and InSpec[32] use a ruby DSL to define the desired target conditions of a server and then connect to the provisioned machine via SSH to determine if the server is correctly configured.

Testing the orchestration of multiple servers, middleware and systems has some similar problems, enlarged by the fact that most cloud providers do not provide a simulated local instance of their services. There are some services with available local implementations like Google's Pub/Sub [69] or AWS Lambda functions [6] but most services do not have these local implementations.

Some open-source projects like LocalStack [41] attempt to create entire local simulations of the cloud environments but their fidelity to the original service brings some doubts when it comes to confiding in these for developing and testing. Therefore, testing orchestration usually tends to rely heavily on the real cloud providers, with diminished speed and increased costs.

2.2.2 Microservices Testing

A microservices architecture comprises multiple small independent services with each microservice, ideally, representing a single responsibility within the system. The main advantages against the monolithic design include the ability to develop, deploy and scale each microservice independently [16]. While each service becomes more independently testable, and we can apply different testing techniques depending on its function, the interactions between the services become a new error surface. Beyond that, there is a difficulty in determining how to test services in isolation when they depend on other services.

Regarding testing, in general, there are two main stages in which the tests can run, pre-production and production. The pre-production stage is, as the name suggests, when tests run in environments that are predecessors to production, and therefore, do not affect the end-users. The production testing stage is when tests run against the same environment that the end-users have access to, meaning that when tests fail, the causes of the failures are, probably, also affecting the end-users.

Pre-production testing main goal is to detect bugs and flaws with the designed systems before they can affect the end-user experience. However, ultimately, the designed software must go into a production environment. For as much as testers try to build production-like environments to test applications, these environments are the mental

model people have of the production environment, which, might not always be entirely accurate, especially in a highly distributed system.

Still, pre-production testing is indispensable, as it is an essential part of software validation and plays a major role in detecting critical bugs before they can harm the end-user experience.

2.2.2.1 Pre-Production Testing

Pre-production testing encompasses different types of testing. Although literature differs on the testing scopes and boundaries are not set in stone, particularly in a microservices architecture, it is reasonable to divide them into unit, integration, component, contract and end-to-end tests [67][73].

Unit Testing

> "A unit test exercises the smallest piece of testable software in the application to determine whether it behaves as expected (in UnitTest by Martin Fowler [73])".

The unit's scope is not strictly defined, but unit tests are usually designed at the class level or against a group of closely related classes. In the microservice scope, it is reasonable to think of the microservice components as the units in themselves.

Moreover, unit testing will always be specific to the component under test and its implementation, and will change accordingly. As it would be unreasonable to showcase every testing technique that could apply to every service component, this section will focus more on the interaction between the components exposed to the tests.

Martin Fowler [75] popularised the terms sociable unit testing and solitary unit testing. Sociable unit testing acts as a black box[2] testing technique, in which the unit under test is tested uniquely through its API and by observing changes in its state. This technique uses the real dependencies of the unit under test, reason why it is called a sociable technique.

Opposed to this technique there is solitary unit testing, where the tests are focused solely on the unit under test and the dependencies are replaced by test doubles to ensure a dependency malfunction does not influence the result of the test.

Test doubles [46] are a generic term for an object used in place of a real object for testing purposes. Other terms exist, such as fakes, mocks, and stubs. These terms are usually used interchangeably, but some nuances differentiate them, as explained in more detail by Gerard Meszaros [43].

[2]Black box testing is a method in which the internal implementation of the SUT is not known to the tester. The SUT is seen as a black box inside which the tester cannot see through [7].

Integration Testing

Integration tests intend to verify the interactions between components and expose potential interface defects associated with their communication. These are at a lower granularity level than unit tests as they no longer strive to test the individual unit functionality but instead, to collect multiple units and test them as a subsystem. Integration level testing exercises the communication paths in the subsystem to determine if there are any incorrect assumptions between components.

Typically, in a microservices architecture, these tests exercise the interactions between the microservice and the external dependencies, other services or data stores. It is essential to notice that these type of tests should only guarantee that the components can communicate with each other clearly. Meaning, they should only exercise essential success and error paths. The goal is not to functionally validate the components but to cover the communication between them. The functionality of the subsystem components should be covered with unit tests. Particularly noteworthy are persistence integration tests and gateway integration tests.

Persistence integration tests [73] aim at validating that the data schemas defined in the code match the schemas in the external data stores. Beyond that, and since this communication is also performed across the network, it is essential to verify that network-related errors are handled accordingly and do not compromise the SUT.

Gateway integration tests [73] target the communication with another service. Integration testing with these services tends to weed out protocol-level communication errors, such as missing HTTP headers, security encryption discrepancies or even request/response mismatches in the communication. This type of testing should also target special error cases to ensure the communication does not break down in unusual circumstances. Because generating these type of errors (e.g., latency, timeouts) can be elaborate, their generation can be accomplished by using a test double that is pre-determined to generate special error cases. One thing to be wary of is the drift that can occur between the developed test doubles and the real implementation, that may lead to the tests passing in the testing environment and the system failing when faced with production traffic. To avoid this situation, tests should be run occasionally, ensuring that the test double still provides the same results as the real instance of the service.

Integration Contract Testing

Contract testing is another particular case of integration level testing, which is particularly relevant in a microservices architecture. To test the interaction between different services, both the consumer and the provider must have a clear notion of the interaction protocol. Contract testing is a way to test the interaction between those two services. Two services agree on a contract, defined as the set of the possible pairs request/response between a provider and a consumer. It is a formal agreement on the communication

protocol between a provider and a consumer [28]. This technique enables testing of the specified APIs promised in the documentation.

Contract testing main focus is that the requests and responses contain the required attributes and that response latency and throughput are within acceptable limits [73]. Contract tests give confidence to the consumers of services, but they are also precious to the producers because, ultimately, the goal of the producers' services is to feed consumers' services. For that reason, it is extremely valuable that the producer services understand how the consumer services are using their service. If the producer receives the contract test suites from all their consumers, they can make changes without impacting the consumers. That is the basis for consumer-driven contract testing.

Consumer-Driven Contract Testing

Consumer-driven contract testing is a specialisation of contract testing. In this type of testing, the consumer specifies the expected format and content of the requests and responses. One could say that this type of testing is implementation agnostic as it only cares about the structure and syntax of the interactions [16]. This specification will then generate a contract that is shared with the provider. Once the provider obtains the contract, he can test against it to guarantee his development does not break the communication with the consumer.

The consumer and the provider must have a close collaboration. Ideally, the same organisation controls the development of both services. Moreover, this method assumes that the requirements of the consumer are used to drive the provider. That is why it only makes sense to use this approach when the consumer team can influence the evolution of the provider service [55].

An advantage of this approach is that only the functionality that is used by the consumers is tested. This fact provides a way to detect unused functionally and also allows to change the API without breaking the consumers. Since the contract contains all the consumer expectations, the provider can make changes freely as long as the contract remains valid [33].

Consumer-driven contract testing has been around for a while now, and mature tools are available. Pact [55], an open-source tool for consumer-driven contract testing, has become the go-to tool to enable consumer-driven contract testing. Pact works by mocking the consumer when testing the provider and mocking the provider when testing the consumer, which provides isolation on the tests.

Figure 2.3 and 2.4 show a brief overview of the workflow of this tool, from the point of view of the consumer and provider, respectively.

Firstly, the consumer defines the expected request/response pair, and that interaction is registered with the mock service. Generally speaking, Pact enforces contracts by example, which means there is a need to define interactions with credible data. Secondly, the consumer dispatches a real request targeting the mock provider. The request is then

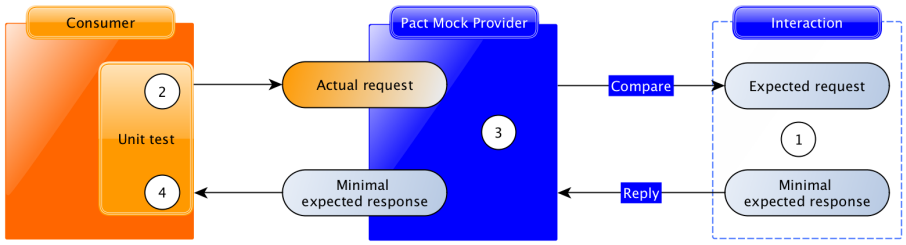


Figure 2.3: Pact consumer testing [30].

compared with the expected request and if their syntax matches, the expected response is forwarded to the consumer that is now able to parse it and verify his behaviour accordingly.

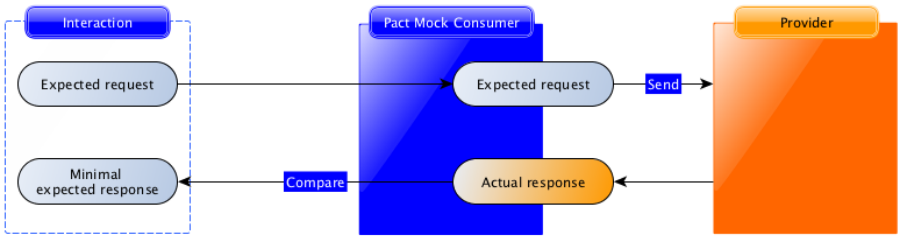


Figure 2.4: Pact provider testing [30].

In the provider portion of the tests, the mock consumer initiates the interaction by sending the previously registered request to the provider. The provider processes the request and responds accordingly. The response is accepted if it contains, at least, the data described in the expected response. Consumer-driven contracts only establish the minimum functionality the consumer needs. This fact allows for multiple consumers to define contracts with the same provider while maintaining leeway for API changes, because there is a contract per consumer the provider knows which fields are essential for each consumer.

Some more benefits of this approach include: being able to develop the consumer service, even if the provider is not fully developed yet; the tests are fast to run as there is minimal infrastructure setup, because of the use of test doubles and the provider knows when it breaks consumers functionality as they verify the contracts in their CI pipeline.

Component Testing

A component test in a microservices architecture refers to verifying the behaviour of a single microservice by exercising the exposed API as a consumer would. Component testing narrows the scope by isolating the microservice through the use of test doubles to replace any existing dependencies [14]. The concept of this technique is very similar to solitary unit testing, but instead of testing a single class or method, we are testing the entire microservice. The primary goal is to make sure the individual service accomplishes what it aims to do according to the business requirements.

The use of test doubles in this type of testing is paramount to truly achieve test isolation. However, the way the test doubles are used is not straightforward, as different approaches to their usage bring different benefits. There are two options: in-process and out-of-process component tests. In-process tests make use of test doubles that run in the same process as the service itself, meaning there is no real communication over the network with the test doubles, while out-of-process tests also exercise the communication over the network.

Emulating the behaviour of an external service requires some thought as, in most cases, there are no implementations for testing purposes. In the case the external service is developed within the same organisation, some authors propose that the same team creating the service should also develop a test double that should be kept on par with the service, while taking some shortcuts, like in-memory database implementations [1]. When we do not control the development of the service, the creation of an implementation for testing purposes would require some assumptions on the way the service operates internally, as the service is usually a black box.

A more common approach is to create a collection of pre-defined interactions, that resemble the real interactions the services would make and use these to drive the tests. The test doubles are then programmed to return the pre-defined responses when matched against a specific request. These work in a similar way as we have seen before for consumer-driven contract testing (section (2.2.2.1), but the intent is no longer to check the interactions syntactically but rather functionally, with tests using credible data. The use of test doubles in component testing is also beneficial to simulate error conditions that are hard to recreate, like server failures, delays and other unusual error cases.

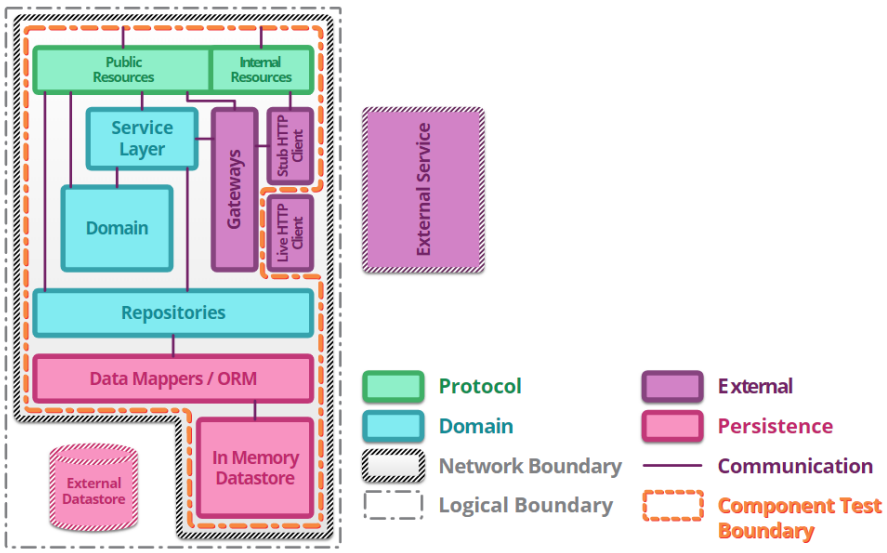


Figure 2.5: In-process component testing schematic [73].

Figure 2.5 shows the topology of an in-process component testing scenario. In this case, we can observe that the external datastore and the external service do not partake in

the test. In-memory databases usually replace the external database dependencies, and the external services are replaced by test doubles that reside inside the process, so the HTTP client segment is not tested in this type of tests.

The absence of network interactions results in reducing both the tests duration and build complexity, as there are fewer artefacts to build. Because this technique abdicates the network altogether, the service must adapt and change for this type of tests. Usually, services use dependency injection to start the service in testing mode according to some configuration parameter.

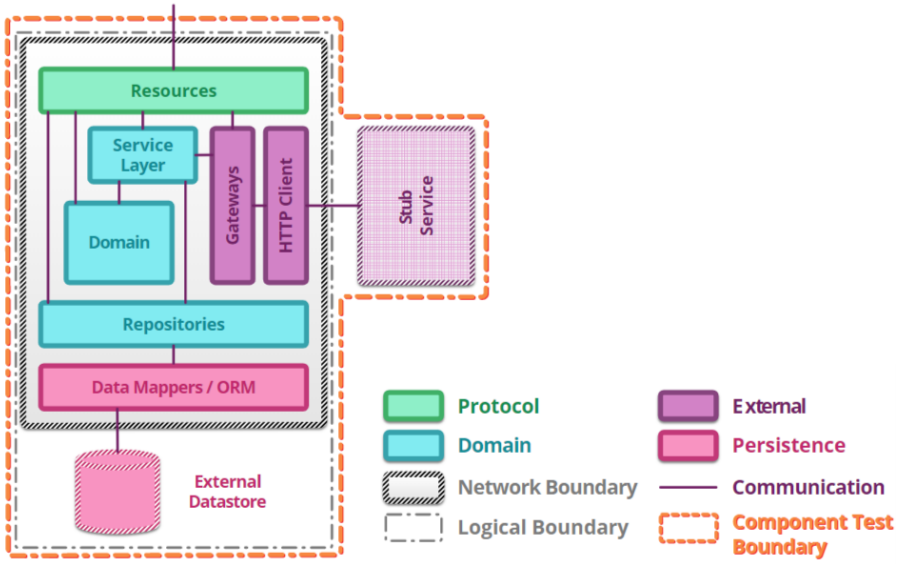


Figure 2.6: Out-of-process component testing schematic [73].

Figure 2.6 shows the schematic of an out-of-process component testing schema. Unlike the in-process component testing, the Http client segment of the service is tested as there are network interactions between the external doubled service and the SUT. Along with testing network interactions with the external service, there are also interactions with the external data store. Testing these interactions gives more confidence that the component is working correctly, at the cost of an increase in the duration of the tests.

Out-of-process component testing brings the complexity associated with testing into the test harness and out of the component itself. The test suite is responsible for coordinating the requests into the target service as well as to spin-up the fake services and needed datastores. The SUT is operating as it would usually because it is mostly unaware of the fact that it is communicating with fake services.

Tools like mounteBank [47] or HoverFly [82] ease the creation of fake services as they can spin-up a test double in a specific address/port, intercepting the requests usually made to a real service and responding in a programmable fashion.

Test Doubles

We have seen the significant advantages that test doubles bring into testing microservices, but there are some caveats to their use.

Test doubles are the reflection of what we think the real service/dependency will be, but they also bring a development challenge because we should not allow the test doubles implementations to drift apart from the real services, so that the system passes the tests and fails when in production. To avoid this drift teams usually run tests regularly comparing their mock implementations with the real services to guarantee that they have not drifted apart and that the doubles are still trustworthy. The tests usually are not run in every pipeline because they would, probably, result in a considerable increase in test duration.

Therefore, test doubles have some weaknesses related with ensuring parity between the real dependencies and the doubled implementation, but this means that we need to be aware of their problems and avoid falling into known pitfalls. Different problems require different solutions, and in some cases the use of test doubles can be a perfect fit, while in others they might not be worth the effort to create and maintain.

End-to-End Testing

End-to-end tests exercise the entire system from an outside perspective. This type of tests resemble component tests, in the way the system is a black box, but they have a much broader scope and, ideally, make no use of test doubles. End-to-end tests are meant to test the entire system as much as possible, including all back-end processing engines and messaging or data transfer services [23], by manipulating the system, through its public APIs or Graphical User Interfaces (GUIs).

End-to-end tests mostly disregard the underlying system architecture, as they pretend to operate the system as an end-user would. This fact exposes one of the advantages of end-to-end tests: they give confidence that the business requirements are met even during large scale architectural refactorings [73].

Occasionally, an external service makes it impractical or impossible to write end-to-end tests without side-effects. Some even bring stability issues to the test suite, when reliability is a problem. End-to-end tests might fail for undetermined reasons outside of the team's control. In these particular situations, it can be useful to replace the problematic dependencies by test doubles, which increases the stability of the test suite at the cost of reducing the confidence in the overall system.

Due to the nature of end-to-end tests involving network interactions and the possibility of having many moving parts (e.g., services, datastores) and having to account for asynchrony in the system, this type of tests can dramatically increase the time cost of the test suite. End-to-end tests also make it harder to debug when errors occur, as they are usually more complicated than other types of tests.

Because end-to-end tests are expensive and harder to debug, it is essential to avoid writing unnecessary end-to-end tests. End-to-end tests are meant to make sure all components of the system integrate seamlessly with each other and that there are no high-level misunderstandings between the components. A strategy that is usually used to make the test suite small is to establish a time budget with which the team is comfortable. Then, as the test suite exceeds the determined time budget, the least valuable tests are removed to remain within the established time limit.

Another strategy to avoid writing unneeded tests is to focus on user journeys. Modelling the tests around realistic usage scenarios brings much more value than focusing on hypothetical scenarios. The lower level testing types should be the ones testing the hypothetical error scenarios as those tests are less expensive.

End-to-end tests should also be as data-independent as possible. They should not rely on the pre-existence of data in the test environment as it can create flakiness in the test suite. If the tests rely on data to execute, the test should also be responsible for making the data available in the test environment.

Due to the inherent problems of end-to-end testing, it is widely accepted that end-to-end tests should be the least represented in a test suite.

Tests Distribution

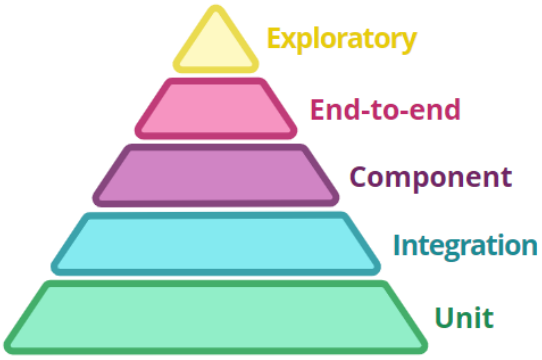


Figure 2.7: Ideal test distribution pyramid [73].

Figure 2.7 shows the test pyramid. Mike Cohn firstly introduced the test pyramid concept in his book "Succeeding with Agile: Software Development Using Scrum" [13] and it has been subject to many changes and interpretations since. The figure shows the testing pyramid adapted by Martin Fowler [73], to fit a microservices architecture. The basic idea behind the testing pyramid is that the higher in the pyramid, the higher the execution time and maintenance costs. Therefore there should be more tests on the lower levels of the pyramid and less on the upper levels.

We can see that, ideally, we should have a higher representation of unit tests than any other type of tests. It goes along with what we have seen previously as unit tests are more focused, faster and cheaper to run than any other test type. Going up the pyramid,

the trend is that each test covers a broader scope, runs in a lesser isolated environment, requiring more resources and increasing the duration of the tests.

In the apex of the pyramid, we find exploratory tests. Exploratory tests are a manual approach to software testing, where the tester is free to explore ways to test the software. It simultaneously tests the functionality of the software and identifies technical problems. Because it is a manual approach and requires a dedicated tester, it is usually the most expensive and least used type of test [71].

The test distribution is a fairly good guideline for designing a test suite, but some adjustments can be made to fit different use cases. Still, some anti-patterns really should be avoided, namely, the inverted pyramid, where the vast majority of the tests are end-to-end tests, with fewer mid-level tests and even fewer unit tests. The over-reliance on end-to-end tests leads to a hard to maintain test suite, unable to focus on the specific implementation details and one that takes unreasonable amounts of time [26].

> "Just like a regular pyramid tends to be the most stable structure in real life, the testing pyramid also tends to be the most stable testing strategy (in Just Say No to More End-to-End Tests [26])."

2.2.2.2 Production Testing

Production testing provides the ability to test with production data and traffic, which is hard to replicate in other environments, and also enables testing in an environment that is 100% accurate with the production environment [90]. Moreover, testing in production develops the ability to build systems that identify and alert for errors promptly, as well as build tools that enable a fast recovery from bugs that end up in production.

It requires an acknowledgement that bugs in production are inevitable, and the ever-faster release-cycles lead us to move away from completely trying to eradicate all bugs in pre-production but to develop the mechanisms that reduce as much as possible the time to recover from these errors in production environments [68].

This section will focus on introducing some relevant techniques for production testing. These techniques will be split into two phases: deployment and release. Because monitoring crosses both phases, we will approach it first.

Monitoring

Monitoring consists of having tools in place that can report the state and overall health of the systems. It is easy to understand that monitoring is an essential part of testing in production. However, aiming to monitor too many aspects of the application can prove to be an anti-pattern.

Instead, the focus should be in identifying a small number of metrics, such as HTTP error rates, requests latency or changes in the services requests rate to avoid false-positive failure detections [77].

A small number of signals decided at design time according to the service requirements, should be representative of the need to roll back or roll forward a software release [70].

Deploy

Deployment is the process of installing a new version of a service on a production environment. A service is deployed when it started running successfully and passes the pre-defined health checks. By definition, deployment does not expose users to the new version [22].

In practice, this means that deploying a new version of a service can occur without the end-users being affected even if the deploy did not end up being successful. It requires that engineering teams develop their services in order that a single failure does not produce a propagating error chain that ends up affecting other services, and most importantly, the end-users [70].

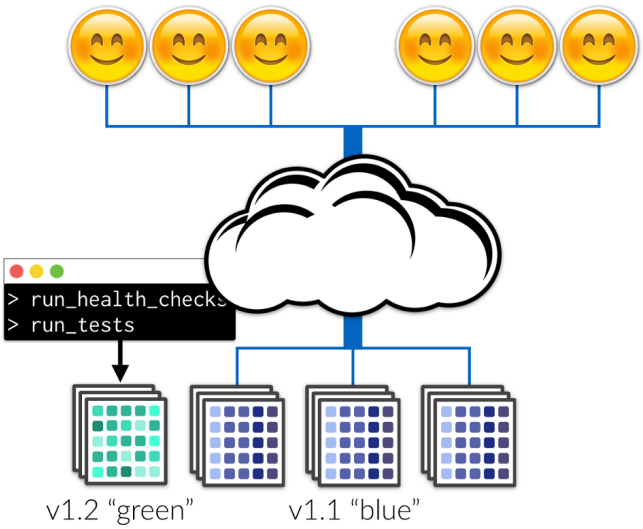


Figure 2.8: Testing on the deployment phase of the production rollout [22].

Figure 2.8 portrays the environment at the deploy phase that enables tests in a production environment to be executed. It is essential to make sure that these tests are treated specially while executing, when compared with the actual production traffic. Stateless services lend themselves particularly well to production testing, but stateful services need to make sure that test data does not get persisted as real user data. Either by discarding the data before it hits a persistence layer or by distinguishing it at that level with additional metadata [70].

Integration Testing

Integration testing in production shares the same goals as integration testing in pre-production, detailed in section 2.2.2.1. The obvious advantage over doing the same in pre-production includes working with an accurate environment without having to develop and maintain another environment. It can also help to verify more than one level of interaction. It can help determine if the services chain does not break down due to any incorrect assumptions past the first level of interaction.

Shadowing (also know as Dark Launch, Dark Traffic or Mirroring)

Shadowing works by capturing production traffic and replaying it against the new version of the service. In a way, it is similar to load testing[3], but the intent is not to only measure performance but to verify the correct handling of the requests.

This technique can be used with different goals, including not only the verification that the requests are correctly interpreted and handled but also to compare service-related metrics with the previously deployed version of the service. These metrics can be used to automate the decision process of the service release.

Release

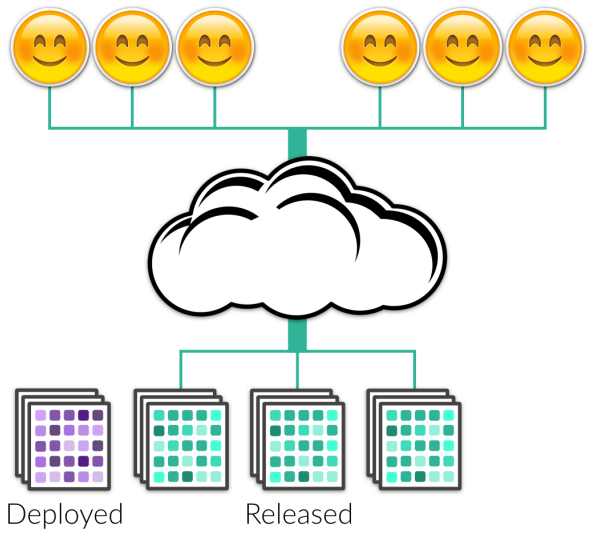


Figure 2.9: Deployed version and released version of software [22].

The release phase, also known as rollout, is the process by which the new version of the service serves production traffic. If an error happens to slip through to this phase, it will directly impact the consumers of the service. In this phase, it is crucial to reduce the time of recovery from such faults, which usually means having a rollback mechanism, a

[3]Load testing is the practice of sending simulated HTTP traffic to a server in order to measure performance[3]

way to return the environment into a previously known good state. Figure 2.9 shows the difference between the release and deploy phases.

Red-Black (also know as Blue-Green)

Red-Black is a strategy in which a new version of the service is spun-up alongside the current live version. The strategy consists in re-routing the users' traffic from one version to the other, while maintaining the previous version idle. This strategy allows for an easy rollback if any issues arise with the newly released version, as it is only required that the load balancer re-routes the traffic to the idle version.

Progressive Rollout (also known as Rolling Red-Black)

Progressive rollout builds on top of the Red-Black deployment premise of having two versions of the service up, but instead of re-routing all traffic to the new version, traffic slowly shifts to the new version. The re-routing can be made simply by increasing the percentage of traffic directed to the new version, or it can use more sophisticated models to determine which users get the new version. These can be, for example, only routing company employees to the new version (dogfooding) and only then the remaining users.

This technique uses a validation gate between every increment that can be anything that seems reasonable to move the service to the next level (e.g., running a smoke test or functional probe).

Canarying

Canarying is very similar to the progressive rollout, but instead of having validation gates, it has a canary analysis between increments. With canarying, the new version of the service is monitored and compared to the non-canary version, the baseline. Only if the metrics (e.g., latency/error rates) are within a defined threshold, more traffic is directed to the new version. If the metrics are not acceptable, a rollback is performed.

Figure 2.10 shows the differences between the mentioned release strategies.

2.3 Continuous Integration and Deployment

Continuous integration and deployment encompass a set of practices that enable development teams to deliver new code more frequently and reliably. These practices are vital to achieving the desired agility that is one of the motives for using a microservices architecture.

2.3.1 Continuous Integration

Continuous integration is the practice of frequently integrating code changes into the main branch of a repository, and testing them as early and often as possible [18].

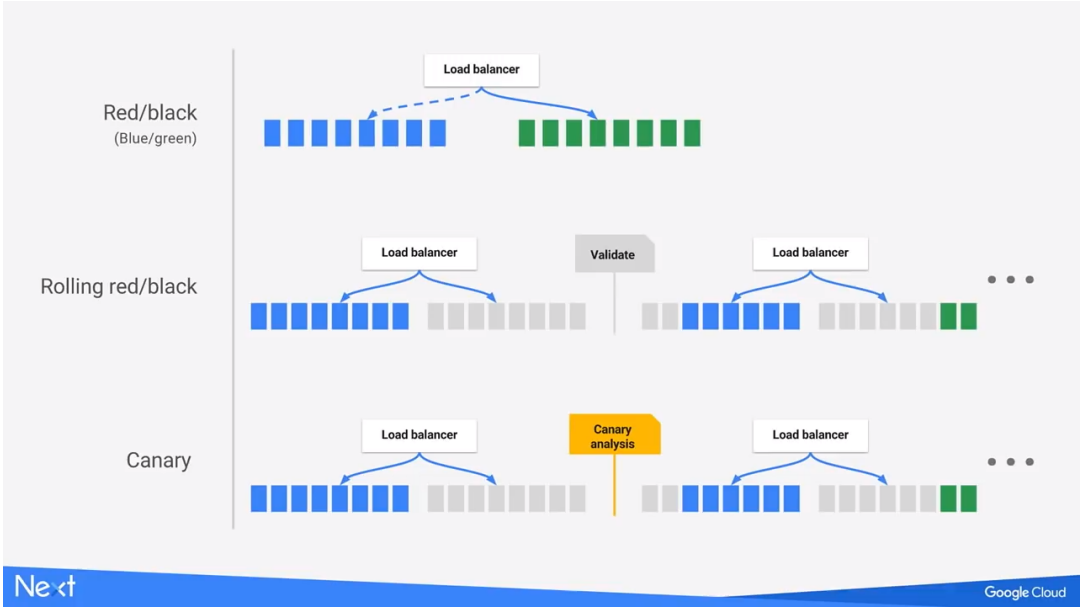


Figure 2.10: Differences in release strategies [58].

With continuous integration, ideally, every time a change occurs, the build is tested. This practice is crucial to speed up the release process and to fix bugs earlier in the development cycle. Continuous integration relies heavily on testing automation to guarantee that the newly committed changes maintain the build in a healthy state.

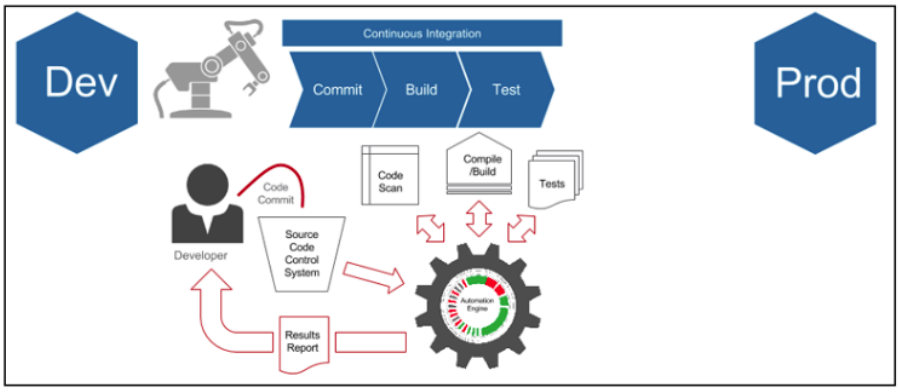


Figure 2.11: Continuous integration pipeline [31].

Figure 2.11 highlights the significant steps in a continuous integration pipeline. After the developer commits his code, it is automatically analysed, packaged and tested by an integration engine, and the developer receives feedback from his changes.

Teams usually use some integration engine like Jenkins [37] or CircleCi [17] to automate the entire continuous integration process.

2.3.2 Continuous Deployment

Continuous deployment extends the process of continuous integration by automatically publishing approved code changes to the production environment. With this process, there is no human intervention, and only a test failure can prevent the changes from being served to the end-users.

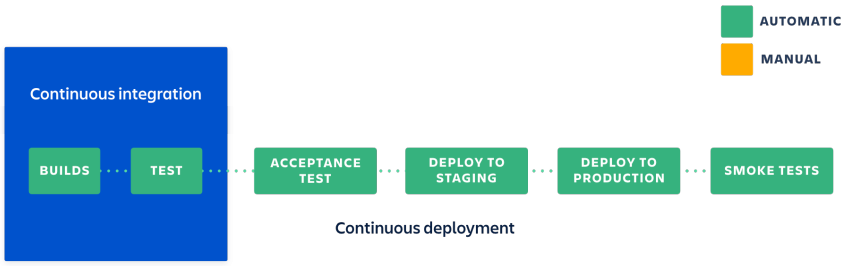


Figure 2.12: Continuous deployment pipeline [19].

Figure 2.12 represents the typical steps of a continuous deployment pipeline. While there is no definitive pipeline structure, the figure shows a generic approach that showcases the intent of this phase.

After the continuous integration pipeline concludes, the service is usually subject to some acceptance test to ensure it is ready to be deployed to a staging environment. After the service deployment to the staging environment, the service is then deployed to production.

In staging and production, it is usual to run a battery of smoke tests in order to ensure the system can deliver its critical functionality. Additionally, in this stage, we can run different types of tests, including the production tests discussed earlier.

Similarly to the continuous integration phase, teams tend to utilise software to automate the continuous deployment process. Spinnaker [64] is one of the tools that supports continuous deployment pipelines. Open-sourced by Netflix in 2018 and backed up by Google, it supports multiple cloud providers and integrates with most continuous integration tools. It comes with built-in support for deployment strategies like red-black and progressive rollout to put releases to the test.

2.3.3 Continuous Integration/Deployment Challenges in Microservices

One of the most significant advantages of following a microservices architecture is to be able to release each service faster and independently. While this is an advantage, it brings along some engineering challenges that must be reasoned about when embracing the concept.

Firstly, for every microservice, we also have a separate code repository, a CI/CD pipeline, and a responsible team. When this happens, it may lead to silos among the

teams, and eventually the system is so spread out that nobody in the organisation has enough knowledge to deploy the entire application, which can result in problems when faced with a disaster recovery scenario [11].

Another challenge is to be able to fit multiple languages and frameworks. While each team is responsible for choosing the language and tools that best fit the individual microservice, it is critical to have in consideration the fact that creating a build process that works transversely between stacks can be hard [11] as the build process should be flexible enough that each team can adapt it to its particular needs.

With the increase of microservices and their development, it is reasonable to believe that service versioning will come into play. Along the CI/CD pipeline, there is a need to build multiple images for testing. While some microservices are pushed to production, some are not. It is essential to have a versioning strategy to have a clear understanding of which versions are currently in production and test accordingly [11].

The fact that teams can independently deliver a new version of the service also creates a risk of deploying incompatible service versions. The release of a service must not break other services functionality. Creating conventions for versioning and tagging services makes it easier to diagnose deployment and release issues.

The use of the deployment strategies referred in the production tests section can also help mitigate some of the risks associated with the independence of the releases.

Chapter 3

Related Work

This chapter aims to showcase work that encompasses the problematic at hand featuring testing strategies related to infrastructure, microservices architectures and continuous integration and deployment pipelines.

3.1 Testing Programmable Infrastructure with Ruby

At QCon 2017, Matt Long presented an approach to testing programmable infrastructure [34]. The SUT was a cloud Broker which created an abstraction layer between the end-user and the cloud service providers, namely AWS and Google Cloud. The system intended to allow the developer teams to quickly provision the needed test infrastructures while maintaining the independence of a single cloud provider which could eventually lead to a lock-in.

This project resembles the OutSystems Orchestration Service as it is not designed to create a single infrastructure and deploy it alongside a developed application, but instead to provide the end-users a way to create, manage and destroy infrastructure on-demand in a way that it embraces the concept of self-provisioning.

The cloud broker utilisation workflow revolved around the users exercising a GUI and inputting the specifications for the required environment, and then the broker would create and bootstrap the resources accordingly.

With that workflow in mind, the followed approach was to separate the workflow into Web testing and infrastructure testing. Web testing only exercised the web page in isolation, meaning it would not spin-up real infrastructure. The set of tools included Cucumber [21] and Selenium [61].

For the infrastructure test framework, it bypassed the web GUI and directly called the APIs for the cloud broker. For this portion of the tests, the tools used were Cucumber

and Serverspec [62].

Serverspec tests allow the tester to define conditions on the desired state of the machine. After the configuration of the machine, the tests run and assert if the machine fulfils all the necessary functional requirements.

In this work, Serverspec tests acted as a sort of smoke tests, designed to detect obvious problems and not complex tasks since it is designed to check single instances and not an entire infrastructure. To perform acceptance testing over the whole system, the tool used was cucumber. Cucumber is not specific to infrastructure testing but has its focus on the end-user perceived behaviour of the system, so it was possible to use it for acceptance testing.

The proposed approach had some positive aspects like the use of a full programming language (Ruby) to create the infrastructure tests which gave more liberty and flexibility to design the tests. The separation introduced in the SUT also enabled the selection of the best tools for each component.

The testing strategy aimed to follow the test pyramid distribution, but it ended up not being respected. The test distribution followed the inverted cone anti-pattern due to the over-reliance on end-to-end tests. They justified it with the fact that unit testing is immature in the area of infrastructure testing and had RoI problems. Beyond that, and since the end-to-end tests ran on real infrastructure, the tests were really slow and expensive, which led to the team only running the tests nightly.

3.2 Spotify Testing Strategy for MicroServices

Spotify's strategy [72] for testing microservices steers away from the traditional test pyramid as they argue it can be harmful. Spotify suggests it presents an ineffective test distribution for testing microservices. Their argument lies in the fact that the most significant complexity within microservices architectures is not within the service itself but in the way it interacts with the other services. They also believe that having too many unit tests can restrict code change, as most code changes would require the rewriting of the tests.

For these reasons, Spotify utilises a Honeycomb test distribution highlighted in figure 3.1. At the base, we find implementation detail tests, and then a more significant number of integration tests, and only then a reduced number of integrated tests (ideally none).

Spotify considers the microservice as an isolated component, and a unit in itself, the reason why the term unit test is replaced by implementation detail. On the other end of the spectrum, they consider integrated tests to be any test such that the outcome of the test is reliant on the correctness of any other system.

Integration tests verify the correctness of a service in a more isolated fashion but focusing on the interaction points with other systems. Considering a service that only depends on a database and provides an API, the pattern used is to spin-up a database,

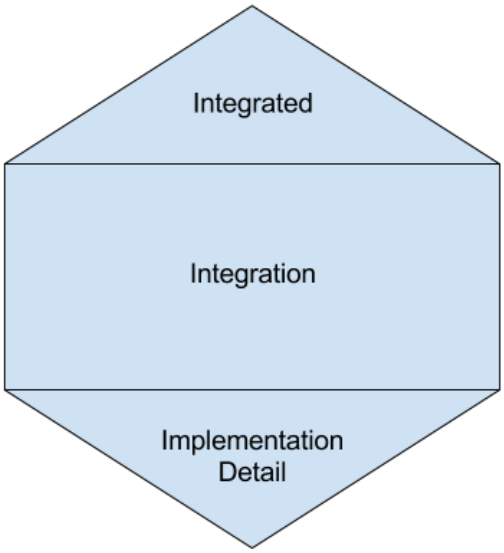


Figure 3.1: Microservices testing honeycomb [72].

populate it, start the service and then query the service as a consumer would verifying the answers are as expected, according to the request.

By not having many implementation detail tests, Spotify feel that they have more freedom to change the internal implementation while guaranteeing that the external functionality is maintained. One trade-off that this distribution presents is a loss of accuracy on a failed test, as the failure will only show the mismatch between the outputs. To fix a bug, they find themselves following the microservice stack traces to find the flaw.

Implementation detail tests focus code that is isolated and has a cognitive complexity on its own. When implementation detail tests are written, they tend to cover all possible error responses. For that reason, the integration tests that follow no longer need to re-test all the possible errors, focusing instead on understanding if the generated errors are correctly interpreted between components.

On a side-note Spotify also mentions that beyond this test distribution, they are also making a transition to use consumer-driven contract testing (detailed previously in section 2.2.2.1). Their goal is to increase the confidence that code changes in provider services do not accidentally break the contract established with consumer services.

3.3 SoundCloud Consumer-Driven Contract Testing

Alon Pe'er from SoundCloud gave a talk at microXchg 2017 [48], explaining how SoundCloud introduced consumer-driven contract testing when they moved to a microservices architecture. They introduced it as a way to reduce the flakiness and over-reliance on end-to-end testing.

They decided on using Pact [55] as their consumer-driven contract testing framework for the tests and make use of an interesting feature of Pact, the pact broker. The Pact

broker facilitates sharing pact files between different projects as well as fetching all pact files related to a service for easier verification. It also has support for versioning and tagging, and it automatically generates documentation for every pact file.

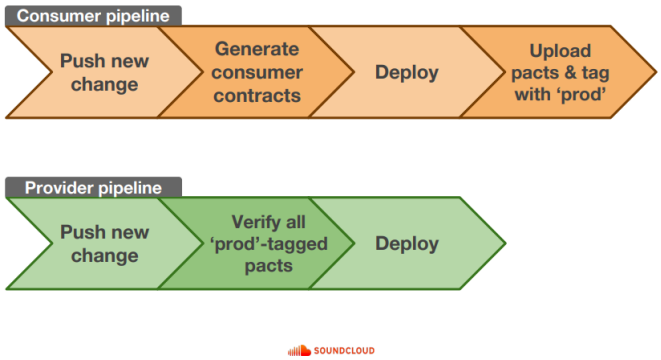


Figure 3.2: SoundCloud's continuous integration pipeline with Pact [48].

Figure 3.2 generally describes the pipeline used for deploying new services. In the consumer side, after a change, the contracts are automatically generated, and the service is deployed. The new contracts are then uploaded to the pact broker and tagged with a label that identifies the production environment.

The provider side must verify all his pact files tagged as production before deploying. Only if all pacts are verified, the new service version is deployed, guaranteeing that the changes have not broken the communication with any consumer service.

The most significant caveats highlighted in the presentation are the need for clear communication between the teams because it is essential for the strategy to work. Also, although the consumers drive the providers, the consumers should not make changes before communicating with the provider team as they need to make sure the changes requested are doable in the context of the provider team.

3.4 Microservices Validation: Mjolnirr Platform Case Study

D.I. Savchenko et al. proposed a validation methodology for generic microservices systems [60]. The strategy highlights the need for having comprehensive validation of the individual microservices. Beyond validating the microservice in an isolated manner, it also references the need to guarantee correct cooperation between microservices. Only then, it is possible to integrate microservices into a holistic system continuously.

The proposed approach was developed against a prototype, called Mjolnirr, constituted of isolated components connected using a message passing pattern. These aspects of the prototype in itself do not guarantee it follows a microservice architecture. The authors emphasise the need for the microservices to be autonomous, meaning each service can be operated independently from other services. Each microservice also has a standardised interface describing the service functions (API), and the functionality scope should be as fine-grained as possible.

The proposed methodology proposes a high-level testing division composed of unit validation, integration validation, and system validation. Unit validation refers to the individual validation of the microservice, integration validation, the validation of the interaction between services and system validation as the validation of the whole system.

System validation is equivalent to the concept of end-to-end testing we have seen before (section 2.2.2.1), and the authors argue the same techniques used in the prior monolithic architectures remain valid. Since the validation at the system level does not consider the internal structure, it follows a black box approach.

Regarding functional unit validation, the strategy defines two main stages: The first is composed of unit, integration, and system validation within the scope of the individual microservice. A single microservice has multiple components that work together to deliver functionality, and just like the composition of microservices aims to provide functionality to fulfil the business requirements, the composition of the microservice components aims to deliver the microservice functionality.

Although each service should have a narrow functionality scope, it can still be relatively complex and be composed of multiple components. That is why each microservice component should be validated (unit validation), as well as its interaction with other components (integration validation). Ultimately, these tests should be accompanied by validation that the components work according to the requirements (system level validation).

The second stage of the functional unit validation concerns the self-validation of the microservice interface. It aims to guarantee that the service complies with the specifications defined in the design phase. It translates into testing the service uniquely through its external interface and analysing the input/output pairs.

Moving on to the integration validation, it aims to validate the communication between the microservices. This phase should weed out issues concerning communication protocols and formats, message sequencing, and other related communication issues that are common in highly distributed systems. The paper highlights the need to have a clear understanding of the messaging graph. Understanding clearly how the services communicate and the message origin and destination enables us to have a clear view of the service topology.

Moreover, the authors argue that only by having an understanding of the message graph it is possible to design integration validation techniques. Additionally, we believe knowing the communication sequence, and the data structure makes it so the test design can be driven from the outside, in a black box approach, which provides a better separation from the implementation, which ties nicely with what is proposed by Spotify in the previous section 3.2.

The authors also resumed the main conditions to determine the quality of the microservice as full isolation for testing at the component level, message communication at the integration level and encapsulation for the system level. The concepts translate into having complete independence of dependencies at the component level, using test

doubles. At the integration level, the tests do not cover the behaviour of the microservice in itself but only how it communicates with others. At the system level, the microservices should be encapsulated and deployed, and the tests should be unaware of the implementation details of each microservice and only exercise the system as the end-user would.

3.5 Waze Deployments with Spinnaker

At Google Cloud' 18, Tom Feiner, Infrastructure Team Lead at Google, gave a talk explaining how Waze started using Spinnaker [64] and the advantages it brought to the company [40].

Before Spinnaker, a deploy of a service into production was a mostly manual process which involved developers creating a server instance, creating software images and then deploying them to multiple scaling groups which meant that rollbacks were extremely painful. As Waze started to grow, and the number of services increased, that process quickly started to become a bottleneck.

With the introduction of Spinnaker, Waze ended up having a production pipeline for every single service. Deployments became a single click operation for every team managed via GUI, and teams also could get feedback from the service in production with complete autonomy from other teams. Other advantage highlighted from Spinnaker was the abstraction it provided between different technology stacks and the ability to deploy to multiple cloud providers.

The change of speed in delivery brought along some new problems as they realised that most pipelines were duplicated with only a few changes, which also silo-ed teams, which meant that any good ideas for the deployment process stayed within the team.

The solution for this problem consisted in creating a pipeline template that most teams were happy to use out of the box while giving the liberty to teams to adapt it for their needs, while still being able to use the templated pipeline. The fact that every team was able to use and contribute to this template also ensured that best-practices in delivery were readily available for everyone.

The templated deployment pipeline started with a bake stage to create the service images. The image was then subject to a smoke test, to ensure the most important functionalities of the service worked correctly, and only then the service deployment to production. According to Tom, this resulted in a significant increase in the delivery speed of the teams, which ended up being a problem, as it brought more risk to the production environment. To resolve this problem, they added a canary stage, to safeguard the production environment.

Figure 3.3 shows the updated pipeline template. In the figure, each car represents a service. In the pipeline, each service's image gets baked and tested before reaching the production environment. After reaching production each service is deployed with a canary stage, therefore keeping two versions of the service live in production. After

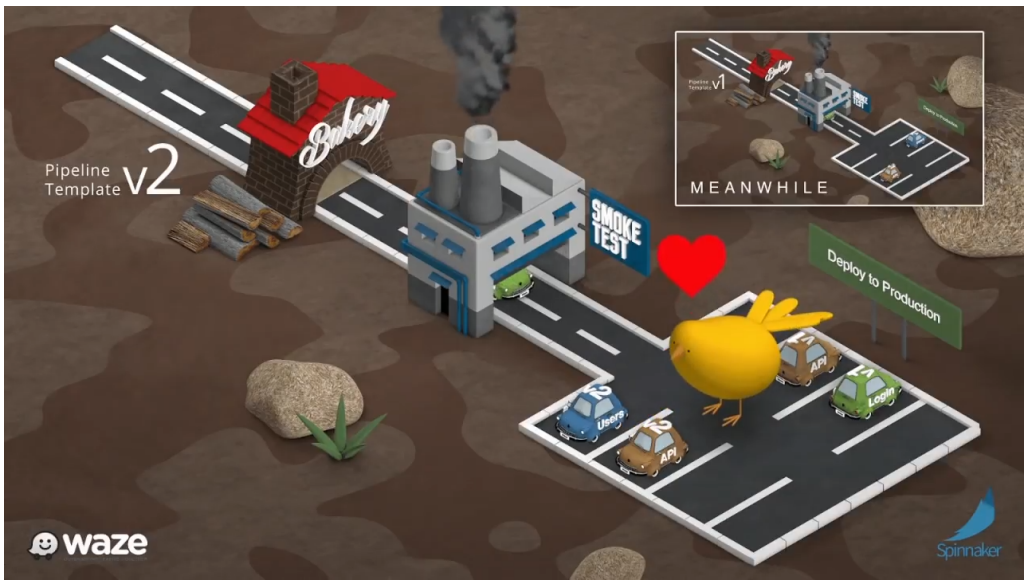


Figure 3.3: Waze pipeline template for continuous deployment [40].

performing a canary analysis, the service is either maintained in production, or destroyed if it does not meet the required criteria.

Another aspect that was alluded to was that the pipelines specification was done entirely in code, which meant that the templates could be easily reviewed and tracked, and pipeline rollbacks were also easier to perform.

3.6 Summary

The related work presented, although not entirely applicable to the problem, helped to shape the solution. Understanding the strategies used to tackle each of the identified problems enabled us to conduct our efforts better and apply some of the solutions presented to our specific use case. The related work also highlighted some lessons and pitfalls to be mindful of when designing the solution.

From the infrastructure testing section (3.1), the main takeaway was the division created in the cloud broker system. Separating the infrastructure configuration enables testing in isolation and use dedicated tools for infrastructure testing. This work also highlighted how infrastructure testing still has some traps, and made it so we were more careful when devising tests for infrastructure and the RoI problems it has.

When it comes to microservices related strategies, we encountered different methodologies. While Spotify's strategy focused mostly on integration tests to ensure microservice validity, the Mjolnirr case study proposed a much more layered approach to testing and involved many more test levels.

The difference in the strategies does not necessarily mean either one of them is better as it highly depends on the context and microservices environments. In our scenario, we drift more to following the Mjolnirr layered testing strategy. The reason we do not follow

Spotify's strategy has mostly to due with our system dependencies. Our dependencies setup and usage is exceptionally time-consuming, which means using many integration tests would produce a much bigger feedback loop. Spotify's definition of integration tests is similar to the definition of component tests, with the caveat that the dependencies are not always replaced by test doubles. If we used a large amount of integration tests with doubled dependencies, we would have a lower feedback loop than if we used the real dependencies. Nevertheless, abusing the usage of test doubles would bring issues concerning the drift against the real services and the maintenance work they would require. If we used the real dependencies we would have no concerns of that nature but they would translate into a much bigger feedback loop. In our specific case using more implementation detail tests allow shortening that feedback loop, and defining less integration and component tests mitigate the risks of using the test doubles.

For services whose dependencies are less time-consuming or in services who are going through major refactoring processes, the definition of more integration tests can be a correct approach. Stepping away from implementation detail tests and designing more integration tests would allow faster change of code without having to refactor the associated implementation detail tests while guaranteeing service behaviour is as expected.

The Mjolnirr case study helped to grasp the concept of decomposition in the microservices. Decomposing an already small service enables defining tests that are extremely focused. Defining tests at a finer-grain allows faster test execution times while also providing better regression protection.

On the other end of the spectrum, the Mjolnirr case study also maintains end-to-end tests, being that they argue that these type of tests do not change from the previous monolithic architecture. In our case, this means we can use the existing end-to-end testing strategy to a lesser degree. We can perform the tests using the same methods, but we do not need to write as many tests as previously because we have a bigger test harness of other types of tests.

From the multiple testing techniques highlighted, a common factor is the use of contract tests. Spotify is making the transition to consumer-driven contract testing, while SoundCloud already completed the transition and highlighted the advantages of the Pact tool to perform said tests. We had already realised Pact was the more mature tool in the area of consumer-driven contract testing. SoundCloud's presentation further proved it to be a complete solution to develop contract tests. It also showed how the Pact tool enabled contract validation and validation of the services independently. The pact broker feature presented also fits well in contract sharing providing an easy way to share the contracts and visualise the communication bridges.

The authors of the Mjolnirr case study do not explicitly suggest the usage of consumer-driven contract testing but instead focus that it is imperative to have a clear notion of the messaging graph. The messaging graph facilitates having a better understanding of the services topology and is essential in the design of functional integration tests. Tied with the remainder research of this dissertation, we also believe this provides a more natural

way to define tests without having to look into the implementation of the microservices. Defining tests looking at the microservices as black boxes also provide a way to be closer to the end-user and can be designed following user-stories.

Building this graph would be an implementation challenge, but the Pact tool provides a built-in feature to visualise this message graph of services bound by contracts.

Regarding CI/CD environments, Waze brought to our attention the usage of Spinnaker. Spinnaker provides an abstraction layer between different technology stacks and the ability to deploy to multiple cloud providers ties nicely with avoiding a vendor lock-in. This work also introduced us to the concept of image baking and immutable infrastructure, and its advantages propelled the usage of Spinnaker (to be discussed in further detail in section 4.1).

Chapter 4

Implementation

This chapter describes the testing strategy implemented and the components developed during this dissertation that support it. Its organisation is as follows:

- Section 4.1 - Continuous Integration and Deployment Environment: explains the architecture, tools and followed patterns, explaining their benefits;
- Section 4.2 - Chef Testing Strategy: focuses on the implemented strategy to test the chef dependency and lessons learned along the way;
- Section 4.3 - Microservices Testing Strategy: explains the defined testing strategy highlighting the different test levels created and the created pipeline.

The constructed solution in this dissertation is a blend between testing strategy definition and its integration into a CI/CD pipeline. The result is a Proof of Concept (PoC) on testing a microservices architecture with infrastructure dependencies, and its implementation showcases techniques and ways to approach the problems seen previously, in section 1.2. The necessary source code (microservices and chef recipes) was developed alongside the strategy definition so as not to constrain the applicable strategy to an already pre-determined architectural model.

Developing the code to test brought some critical contributions to the overall strategy. Creating the prototype from a blank slate made it possible to understand how the design of the system itself dramatically influences the possible testing approaches. Because the orchestrator system is soon to be undergoing an architectural change, it was also crucial for the dissertation to provide insight on how the design of the system itself affects testability. For that reason, using an already built net of services would constrain the strategy that would be applicable.

The creation of the testing strategy had plug-ability in mind. Because of the lack of maturity in the area of infrastructure testing, some of the tools used might eventually change, but the goal was that the strategy would remain even if the tools used did not. An effort was made to understand why and what we should test at each level and not only how. Understanding *how* to test at each level is closely tied with the technology and available tools at the moment, while understanding *why* and *what* to test at each level gives freedom on which tools to apply. Identifying the problems each test level should detect also contributes to avoiding test repetition between the different test levels.

Because defining the testing strategy needs to have in mind all the different pieces and dependencies of the original system, this project considers the main dependencies of the system and integrates them in the PoC, to emulate the constraints and needs of the orchestrator system. The implementation has three main facets. Firstly, there is the definition of the CI/CD environment to support the development, testing, and deployment of the built components. There is the creation and definition of a testing strategy to assert the validity of chef recipes that are a primary driver of the orchestrator system to configure machine instances. Lastly, there is the creation of a set of microservices that exercise the major dependency of the orchestrator, the AWS, alongside the definition of the testing strategy for these microservices.

A gap was created between the Chef Recipes and the microservices due to the responsibilities each of those components have. While Chef recipes are responsible for configuring machines, the microservices are at a higher level in a sense they orchestrate the infrastructure and not only machines. From an architectural standpoint, the microservices will call upon AWS and other external dependencies to orchestrate the infrastructure, but the responsibility of configuring machines to have the desired state is passed to the Chef Configuration Management Tool. The Chef recipes are an essential piece in the scope of configuration, while microservices are more on the scope of orchestration.

In a holistic view of the system, the microservices would call upon chef recipes to configure machines. With that in consideration, it is always needed to be able to test the chef recipes in isolation and ultimately test the integration between the chef recipes and the microservices. This integration was not considered in the PoC due to the overhead of setting up the Chef Configuration Tool to be able to interact with the microservices. This setup would involve, among other requirements, instantiating a Chef Server to hold the recipes, manage licenses, and the machines to apply the recipes would need certain pre-requirements. The time investment that the setup would require revealed that the RoI was too low to test that specific integration.

4.1 Continuous Integration/Deployment Environment

The creation of the continuous integration and deployment environment had in consideration the challenges that microservices often bring into the release cycle. The independent nature of the services means that any service could if need be, be deployed independently

of all other services. While this enables a faster speed regarding value delivery, mechanisms must be in place to deal with the inherent complexity of this model with many moving pieces.

With this in mind, the decision was to embrace the immutable infrastructure pattern while deploying the services, which means that every service version has an immutable image associated, creating a more linear rollback experience.

The concept of immutable infrastructure rivals the traditional mutable infrastructure. In the mutable infrastructure model, the servers are updated in-place. The upgrade and downgrade of packages and change of versions of the services occurs in the existing server that is continuously mutated.

With immutable infrastructure, the servers do not change after the first deployment. Whenever a new version of the service needs to be deployed, a new server is provisioned and once tested, it replaces the previous one [83].

This paradigm aims to provide more consistency and a more predictable deployment process. Because the configuration of the servers at any specific time is mappable to a server image kept in source control, it also provides a more straightforward rollback process.

Additionally, due to the usage of immutable images, it becomes easier to scale horizontally (creating more server replicas to distribute traffic) and also allows using the same image for testing and production purposes. Propagating the same image across the pipeline avoids configuration differences that can ultimately result in different behaviours in testing and production stages.

To embrace the immutable infrastructure paradigm, the tool that checked that particular box while being able to integrate seamlessly with continuous integration tools was Spinnaker. Spinnaker's basis of immutable infrastructure was the main reason for choosing it over other CD tools. The fact that it also provides built-in production testing techniques also meant that experimenting with this tool could also unlock some potential of future work in the area of production testing.

Jenkins was the chosen CI tool, mainly because it integrated very well with the Spinnaker workflow, and it allowed to have the pipelines managed as code alongside the code repositories. This way, a failure of a particular build could be more easily tracked and associated with the pipeline processes and not only to the source code. It also provided a better workflow for sharing similar pipeline definitions between different microservices, mapping back to the concept of pipeline templating (3.5). Pipelines as Code "allow Jenkins to discover, manage, and run jobs for multiple source repositories and branches eliminating the need for manual job creation and management [56] ".

Moreover, Jenkins is among the most distinguished CI tools with tremendous plugin support which enables working with many different languages and utilities. This pluggable architecture fits well with the microservices paradigm of being able to develop multiple services in different languages, and managing each pipeline as code to manage their differences and particularities [38].

Because the project was divided into different areas, namely the separation created between the definition and testing of chef recipes and the definition and testing of microservices, there is not a general pipeline that covers both.

Still, there are some common aspects that both share. Firstly there is the integration with team communication software, namely slack[88], meaning there is no need for the developer to continually monitor the pipeline to check on its behaviour as every step of the pipeline can automatically message the developer in case of failure or success.

On another note, source code and the pipelines themselves are stored in GitHub[25] repositories and pushing a new version of the project to the repository will immediately run the defined pipeline for that repository. This process allows the developer to check the build, test, and deployment of its code in small incremental steps while guaranteeing the quality of the code in a fully automated process.

The following section will explain the architecture of the CI/CD environment and the pre-production and production environments for the microservices.

4.1.1 Environments Architecture

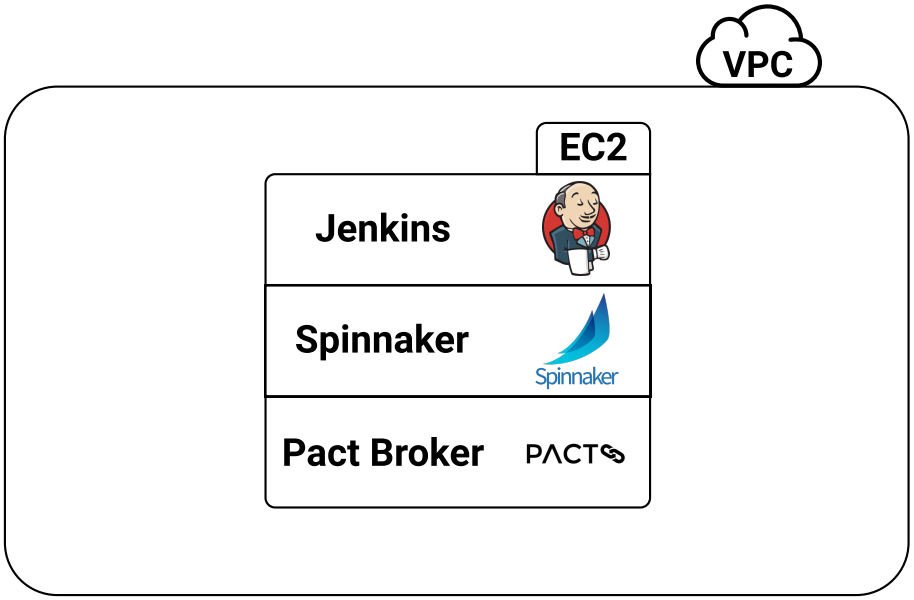


Figure 4.1: CI/CD environment architecture.

Figure 4.1 represents the isolated environment that holds the server responsible for managing the CI/CD pipelines.

It consists of an EC2 machine gated by a VPC and configured with all the necessary tools to manage the pipelines' execution. Most notably Jenkins, Spinnaker and the Pact broker are installed as these are the main drivers for defining and overseeing the pipelines.

Apart from the CI/CD environment, two other environments are essential for the microservices context.

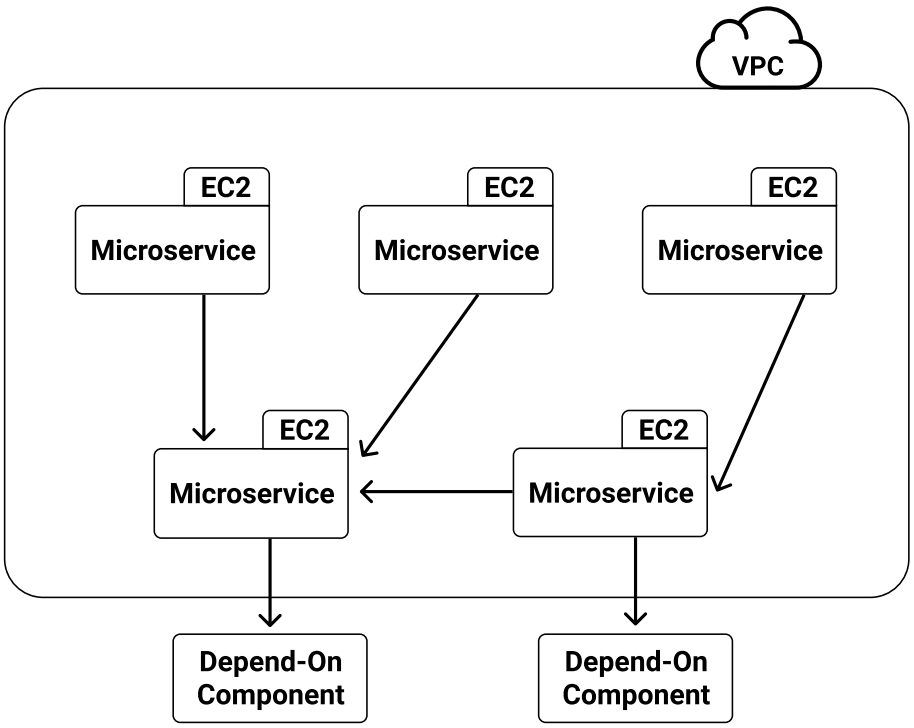


Figure 4.2: Pre-production and Production environment architecture.

Figure 4.2 represents the architecture of the pre-production and production environment. The environments are distinct and isolated, but the architecture is, mainly, the same. The goal of having a pre-production environment is to test the services in an environment that is as close as possible to the production environment but without affecting end-users.

The main differences between the environments are in the accessibility and security rules. While the pre-production environment is only accessible from the CI/CD environment to run the tests, the production environment is also available to the end-users.

4.2 Chef Testing Strategy

The approach followed to define a testing strategy for the chef recipes started by analysing the tool itself, and its usage within the OutSystems orchestrator system. The analysis highlighted the need to create a set of recipes that used composition. In the Chef context, this translates into defining recipes that use other recipes. In a simple analogy against program code, it is like defining a method to perform a particular operation and using that method in different parts of the program to avoid code repetition.

Beyond analysing the recipes themselves, we also researched the Chef testing space for testing tools and methodologies that would apply to our use case. The tools discovered can be categorised into three main testing areas: static analysis, unit testing, and integration testing. The decision to use the following tools was facilitated by the fact that in the Chef testing space, there is a small number of tools available. This fact made it so deciding

between tools was not hard at all, and there was only the need to determine if the existent tools provided any benefit and RoI.

We have seen before that Chef is a powerful tool to configure machines, but we have not delved closely on how it performs such configuration.

Before we present the strategy, we must have a good understanding of the tool itself. The main driver of the machines configuration are the Chef Recipes, which are written using Ruby. Beyond using ruby, which provides excellent flexibility, they use the Chef DSL, which provides a collection of configurable pre-built resources.

Generally, the recipes are a collection of configured resource blocks aided by ruby helper-code that helps control the execution flow. Having a full programming language supporting the creation of the recipes allows defining complex flows while the existence of the pre-built resources makes it simple to define more straightforward recipes. Furthermore, recipes also feature attributes, that work basically as inputs to the recipe.

Figures 4.3 and 4.4 map two compositional recipes into graphs because understanding the DSL and resource semantics is somewhat irrelevant to understanding the strategy. Additionally, while designing the tests, we realised that it would be easier to understand the coverage of the recipe tests by transposing the resources into graph nodes which also adds an abstraction layer and allows an easier understanding of the recipe functionality without worrying about semantics. To showcase the strategy, it also hides some complexity of the recipe definition that is irrelevant for the high-level strategy definition.

The presented recipes will work as a running example while explaining some aspects of the Chef testing strategy. The presented recipes are as simplified as possible while maintaining enough complexity to explain some of the crucial details of the strategy. The following recipes were inspired by the ones made available by Jamie Tanna [35] with some modifications to better suit our needs.

The goal of the recipes is to fetch a remote executable file and use it to start a service in a Linux based machine. The composition is achieved by defining a recipe that has the sole responsibility of downloading a file from a specific Uniform Resource Identifier (URI) and sets the location of the file as its output. The recipe that installs the service performs the operations related to defining and starting a service based on an executable but calls upon the first recipe to fetch the file.

The recipe responsible for downloading the executable (Figure 4.3) starts by verifying if there is a defined URI and if not, throws a custom exception. If the URI is correctly defined, it goes through an additional check to see if the specified file is in zip format. If it is in zip format, it requires the path for the executable inside the zip file to be specified, and a failure to do so raises a custom exception.

If the file is not zipped, meaning it is an executable file, the recipe calls a Chef resource to download the file based on the given URI and sets the absolute path output value with the download file location. If the file is zipped, there are a couple of additional steps. First, we need to make sure that the system is capable of extracting files from zipped files. In order to extract the files, the system must have installed a program capable of doing

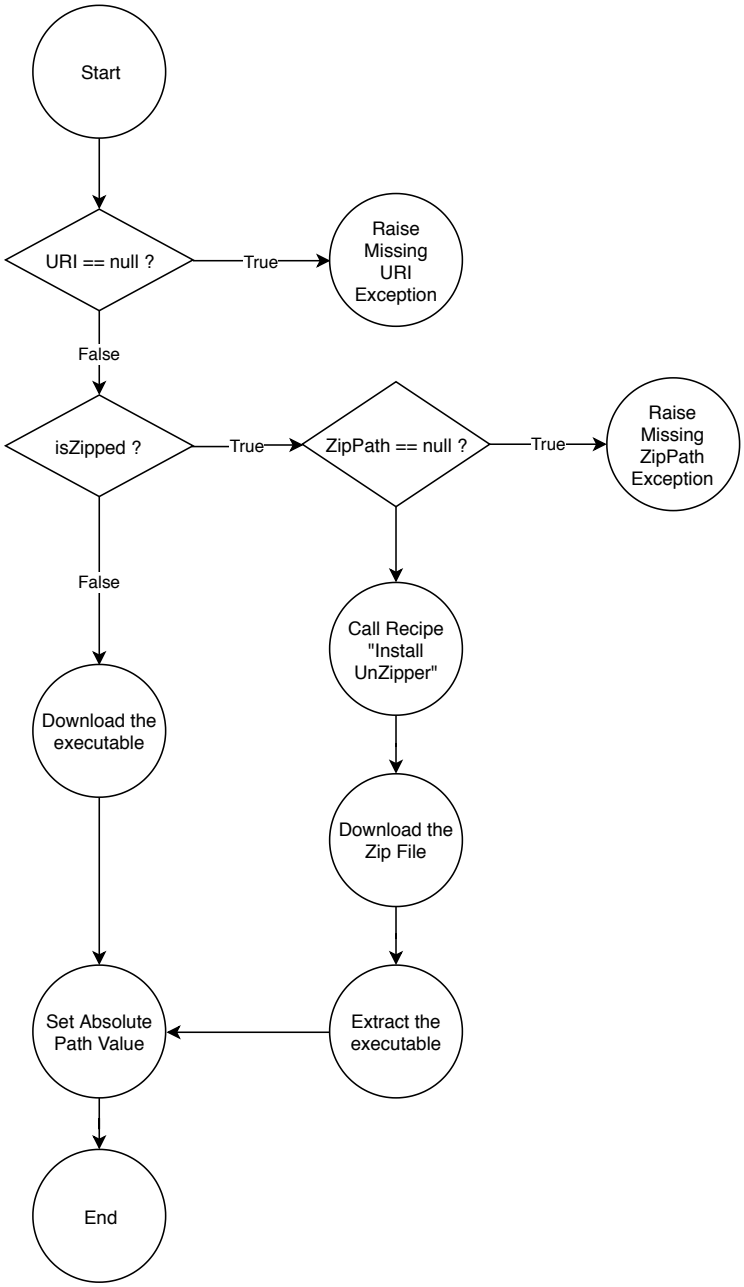


Figure 4.3: Download executable recipe mapped to control flow graph.

so. Instead of manually installing it, we call upon a recipe that does just that. This recipe is available from the Chef Supermarket [78], a community-driven hub of maintained recipes, which means we do not need to implement and test such a basic recipe. After the instalment, the recipe downloads the zip file from the specified URI and then extracts the executable file. After the download, it also sets the absolute path output value with the extracted file location.

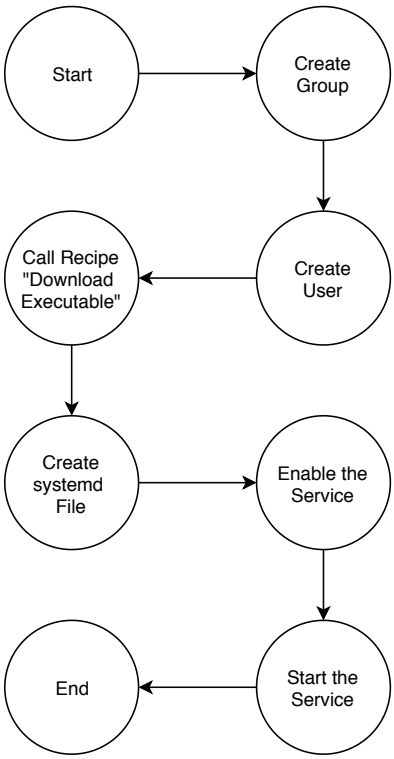


Figure 4.4: Service install recipe mapped to control flow graph.

The recipe that installs and starts the service (Figure 4.4) is very straightforward. It starts by creating a group and a user that will be tied to the service. After creating those, it calls upon the previous recipe to download the executable file. Since the recipe to download the executable outputs the file location, we have all the needed details to create the required systemd service file. This service file must contain the executable location, user, and group to run the service. After creating the file, all there is left to do is enable and start the service.

With this example in mind and knowing how the recipes work more comprehensively, we have all the necessary context to understand the strategy application.

4.2.1 Implemented Test Levels

4.2.1.1 Static Analysis

To perform static analysis we used two complementary tools: Cookstyle[20] and Foodcritic[24].

Cookstyle inspected the recipes enforcing style conventions and best-practices rules. The tool itself comes with a pre-determined set of rules, but it provides the ability to customize them, which means the teams have complete control over what styles and conventions they enforce.

This step strives to establish uniformity of the recipes, which should result in fewer errors and in code that is easier to maintain. Cookstyle's usefulness goes beyond only detecting broken rules as it is also able to correct them automatically.

For example, we defined rules to demand the usage of single-quoted strings over double-quoted whenever string interpolation was not needed, and Cookstyle was able to correct that automatically. This functionality could prove very useful when there is a need to convert existent recipes to comply with a new set of rules. Instead of manually correcting every recipe, we can just run the correction command saving developers' time.

Foodcritic is very similar to the previous tool in a sense that is also capable of enforcing many of the same rules, but it has two main differences. It is not capable of correcting issues automatically, and it is capable of detecting some new relevant issues, namely anti-patterns. It is capable of warning, for example against deprecated functions or even suggesting more readable ways of defining a particular code block.

Because we want to have the capability of automatically correcting issues and be warned against anti-patterns that might lead to execution errors, we run both tools against the recipes. The small overlap created between both tools execution is almost negligible because both sets of analysis can run in seconds, making the usage of both viable.

4.2.1.2 Unit Testing

To explain the methodology for creating unit tests, it is essential to understand the scope of unit tests in the Chef environment. The Chef recipes are designed to be applied to a machine and converge the system into a specific state. The input values provided to the recipe will usually determine the final state of the system. Just like a method in program code will have conditional statements that change the traversal of the execution graph, the same happens in chef recipes.

Looking at a recipe in this way allows us to think about testing chef recipes like we test program methods. The issue that arises is that to test the recipe we would need to converge a real machine with the recipe, which would go against the definition of unit tests, as these are meant to be fast and beyond that, converging a real system could be seen as more of an integration test, as we can observe how the system integrates with the application of the recipe.

Looking at these conditions, the only tool that allowed to test with these constraints was ChefSpec [10]. ChefSpec simulates the convergence of a recipe on a machine, meaning it runs the recipe collecting information on what resources would be called and with which attributes, but skips applying those resources to the system, resulting in much faster execution times. This simulated convergence model meant that ChefSpec was good

in asserting which resources would be applied to the system according to the input values. Consequently, it excels at testing complex logic and asserting the result of input combinations. A result of this simulated convergence model is that ChefSpec cannot determine if the machine reacts as expected to the recipe because it skips applying changes to it, and for to this reason is very coupled to the developers mental model on how the system reacts to the application of the resources.

Taking the recipe from figure 4.3 as an example. A good set of unit tests would try multiple input combinations and assert that the expected resources were configured. If we define an input combination with a missing URI, that test would only need to assert that the right exception was thrown, likewise if the input combination stated that the file was in zip format and we do not include the path inside the zip we would check if the exception for the missing zip path was thrown.

In designing the tests, the focus was always to have full node/resource coverage, but that coverage was not enough. Striving to achieve node coverage was not tricky as defining only a couple of tests would, usually, grant this coverage. This phenomenon is related to the way Chef resources can incorporate some of the logic conditions within themselves, which means that a test that covers a specific node does not necessarily cover all possible node mutations.

Taking the resource "Download the executable" as an example. This resource was mapped to a node, and its functionality seems pretty simple, it should be responsible for downloading the executable to the machine. So, we define a test that provides a specific URI and checks that the resource is called with the specified URI. This simple test would grant node coverage, but this resource has more complexity than it seems. Beyond being responsible for downloading the executable, it must define a user that owns the resource and the specific permissions. If we consider the default values for these extra fields, we can add those values to the previous test assertions and check that the file was downloaded with the correct URI, user, and permissions. Even if we forgot to check these new conditions, we would still have node coverage, but we would never check the user and file permissions that could lead to an execution error. If we make the user an input parameter, meaning we can create a user to be responsible for the file, we should make another test that checks that the resource respects the custom user input.

We can see that these tests go beyond node coverage, and they should. For the prototype, tests were defined to validate the different combinations of input values and assert that the resources were called with the expected attributes, to enhance the confidence in the recipes. A failure in the assertions would usually mean the resources did not respect the input values configuration.

Chef, and specifically ChefSpec, only has support for checking resource coverage. Because that was insufficient, we needed to define a method for creating a good set of unit tests. The fact that Chef allows the definition of default attributes for the recipes created a good starting point to define the first couple of tests. The first tests would always cover the default input values and assert they configured the resources correctly. The second

set of tests would then cover the exception scenarios.

In the next set of tests, the strategy was to override the value of the default attributes one by one. For each test in this set, the scope should be as narrow as possible. Each test should focus on asserting how the modified input value affected the execution flow and the configured resources, avoiding repeating assertions of previous tests. Following this strategy enhanced the focus of each test and made it so an error on a specific test would be easy to map to the error in the implementation, enhancing debug-ability.

Applying this strategy to recipe 4.4, we would first create a test with the default input attributes. The test asserts the correct configuration of all the defined resources in the recipe. Because the recipe does not throw any exceptions, we skip the second set of tests.

The last set of unit tests is the most extensive. We would start by changing the default group. The test would check if the create group resource had the custom group, that the user created was inserted in the custom group, if the group was correctly passed down to the called recipe and finally check that the systemd file specified the custom group. We do not need to check the enable and start service nodes because the group input parameter does not affect these. Adding assertions for these resources served no purpose and would only reduce the test focus. The remainder tests follow the same pattern but changing the focused attribute.

Even by defining unit tests with this strategy, we could not guarantee the total correctness of the recipe because of the simulated convergence model of unit tests. Because of the constraints, it is important to have a test phase that actually converges the recipes into the machine and tests this integration.

4.2.1.3 Integration Testing

To perform integration testing, the main contenders were ServerSpec[62] and InSpec[32], both created by Chef. Because InSpec was built-upon ServerSpec and it is going to be the more maintained project comparatively, it became our tool of choice.

With InSpec, we were able to define functional tests to determine how the recipes converged on actual systems. For this, we set up virtual machines and chose a small subset of input values and defined the functional/acceptance assertions in InSpec tests. Contrary to the ChefSpec tests, these take much longer to run as it is necessary to instantiate the virtual machines, converge the recipe, and only after that process, we can validate the correct configuration.

Because these tests take so much more time (in our case a ChefSpec test took about one second to run while an InSpec test took about two and a half minutes), the written tests in InSpec took a much more functional approach. They determined if, for example, the service was running in a specific port while in the unit test section, we could only check if the service resource was created with the expected configuration.

In terms of quantity of tests and because of the big time investment these tests take,

the strategy was to design only a handful of tests based on essential user journeys to determine the acceptance of the recipe, leaving most of the logic work (the input permutations testing) to ChefSpec tests.

Even though they are a significant investment in terms of the time of execution, these tests were easier to write and to maintain than ChefSpec tests and were able to detect errors that ChefSpec could not detect because of its simulated convergence model.

Taking recipe 4.4 as an example. With a ChefSpec test, we would verify that the recipe contained a resource that would create the service file with the correct attributes, including a specific port, while an InSpec test would check if the services running on the machine contained the specific service running on a particular port. An instance where ChefSpec tests would not fail and InSpec tests would is if we configured the service to run in a root reserved port. In ChefSpec, we would assert that the service file was correctly configured to use the port specified in the input parameters. If, by any chance, we specified a port under 1024 the ChefSpec tests would pass, but when trying to run the recipe with the same input values in a real machine and asserting with InSpec that the service was running, it would show that the service never started running because it could not run on those conditions. Linux services can only run services on those ports if the user has root privileges.

Some of these situations appeared while designing the tests, and while some of the failed InSpec tests were able to be mapped to unit tests, in this example, we could check that the port input value was always above 1024, some were not mappable and thus InSpec tests were essential in determining the validity of the recipe. While discovering failing situations in InSpec tests, it was essential to have an awareness and make an effort to map, if possible, the failing situation to a unit test as they were much faster than defining InSpec tests.

Lessons Learned

Beyond being able to define tests that focus on the recipes in an isolated manner, which is already a significant improvement on the current end-to-end approach, the process of creating recipes from scratch and defining tests allowed to extract some valuable lessons that contribute to the strategy.

Regarding recipe composition, we realised that promoting recipe separation and composition lead to more modular recipes and in term a better testing experience. It is simple math. Let us suppose we have a recipe with six input attributes that for the sake of simplifying the example can only represent boolean values. If we want to test all combinations of these inputs, we will have $2^6(64)$ combinations that we should test to cover all possible value combinations. If we wanted to test all 64 combinations, it would mean having 64 tests for a single recipe. If instead, we were able to refactor this recipe into two, by extracting a recipe and using composition, we would reduce the number of needed tests. If we split that recipe into two, and assume that each of these recipes now takes half of

the inputs that would translate into testing each one with a combination of three input values $2^3 * 2(16)$ combinations, drastically reducing the combinations and avoiding the previous combinatorial explosion. Promoting recipe composition translates into needing to define fewer unit tests (ChefSpec tests) to assert the recipe's logic. This separation also means that the integration between the two recipes can become a new error surface, but the definition of integration tests (InSpec tests) can help in asserting that the two recipes work well together. This does not mean that the number of integration tests has to increase when compared to the original recipe (without composition), as we can keep using the same integration tests composed for the original recipe, as these tests should be enough to guarantee that the separation does not break the original recipe functionality.

Another realisation was that because of the nature of ChefSpec tests, in recipes that lacked complex logic, meaning the recipe graph had a single or only a few paths with few attributes, the definition of the tests almost generated a one to one mapping with the recipe itself, meaning the tests were almost an exact copy of the recipe. While designing the tests, this seems like a non-issue, because even if we are applying Test Driven Development (TDD) the reflection between tests and recipe made it easy to define each other. The problem arose when there was a need to change the recipe to add functionality, for example. Because of the one-to-one mapping between tests and recipe, changes to the recipe will most likely break the tests. This means that changes must be performed in both, resulting in double the maintenance work. Also, because the tests are so coupled to the implementation, the need to change them every time the recipe is changed reduces their value, as they are no longer a guarantee of stability and correctness.

4.2.2 Pipeline

The defined pipeline for the chef recipes was fairly simple, has it had no CD section.

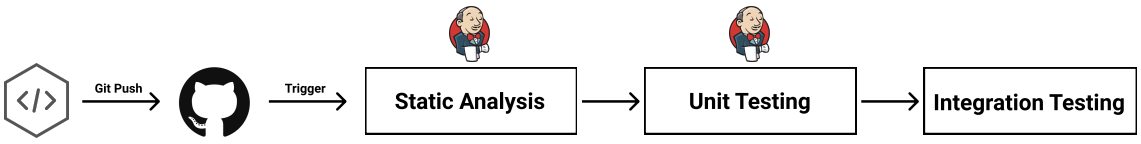


Figure 4.5: Chef pipeline.

Figure 4.5 represents the Chef pipeline. The recipes were stored in source control, and a commit triggered the tests to run. In our CI/CD environment, we only ran static analysis and unit tests.

Integration tests were not run as part of the pipeline because locally they ran by starting virtual machines, using VirtualBox[50] and Vagrant[76], coordinated by ChefKitchen[79]. The remote environment was a virtual machine hosted in AWS, which meant we would need to change the process, as we would not be able to instantiate virtual machines inside a virtual machine.

The solution would be to no longer use VirtualBox and Vagrant but instead start virtual machines in AWS, configure them and then assert their state. The benefit from this approach in the context of this dissertation was not justifiable as the recipes were not actually being used beyond the context of the tests definition and strategy mapping. For that reason, we considered that running the integration tests locally was enough for said goals.

4.3 Microservices Testing Strategy

The first step towards the definition of the microservices testing strategy was the creation of the microservices prototype. The prototype exercised the most critical dependencies of the orchestrator system. Only by defining a prototype that resembles the upcoming microservices functionality, it is possible to determine if the strategy is a good fit for our context.

The design of the prototype revolved around defining microservices that performed cloud operations commonly used in the orchestrator system. The first decision made was how to implement the microservices. Mainly, the language and framework to use. As stated before, the strategy should be able to fit multiple languages and frameworks as it should focus more on the concepts and less on the specifics of each technology.

Mainly due to familiarity with the stack, the microservices were developed using Spring Boot[65] and Java[36]. Already understanding the concepts of both language, and framework enabled a faster ramp-up time which in term increased the time available for defining the testing strategy.

Notwithstanding knowing the stack, there was still a need to understand how to pack the service, integrate it in the pipeline, make the pipeline run the tests and report the results, and deploy it in the different environments. The implementation of the first microservice focused mostly on these topics.

Because the main focus was not in the design of the microservice itself, a vital lesson emerged. The developed microservice was not adequately designed for testing in the ways we wanted. We understood the need to have multiple test levels from the performed research in the preparation phase, but when it was time to define the first tests at the highest granularity possible, namely unit tests, it simply was not possible.

The microservices' narrow scope and simple functionality ended up being a trap into agglomerating different responsibilities into a single component. It meant that, for example, the smallest class we wanted to test with unit tests had already too many integration points and hidden complexity that made defining unit tests a nightmare. Understanding proper responsibilities division, and how the design of the service influenced the definition of the tests, was a clear indicator of the need to design the microservices following a layered architecture, with clear boundaries and well-defined responsibilities.

After studying and researching the common architectures of microservices, we realised that most just had different names for the same concepts. For consistency purposes,

the concepts will be explained using usual Spring terminology.

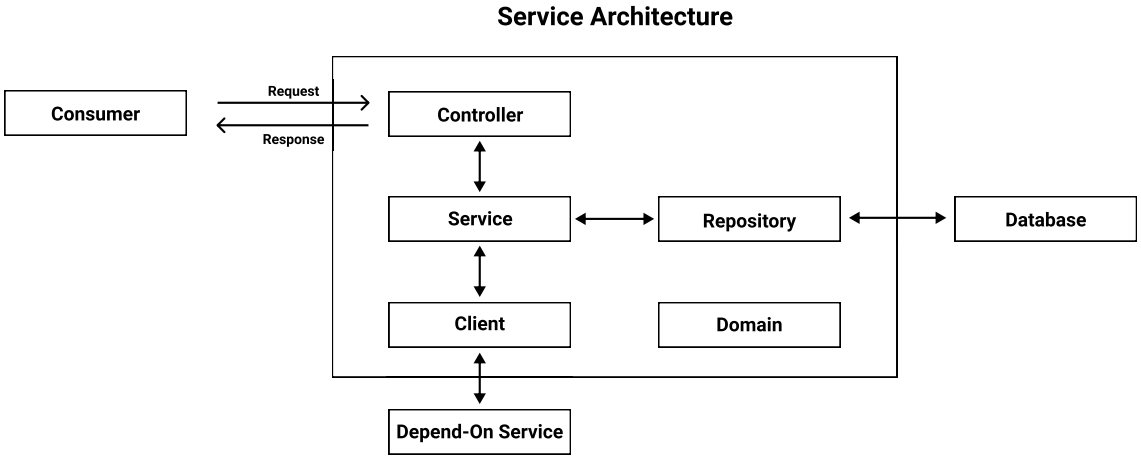


Figure 4.6: Microservice architecture schematic.

Figure 4.6 represents the general architecture of the implemented microservices. The controller classes are the entry point of the service and are responsible for validating and routing the incoming requests. The service layer encompasses the business logic specific to the microservice and communicates with the clients and existing repositories. The client classes create an abstraction layer for all the communications with other services of which we are dependent, while the repositories make the bridge to a persistence solution. The domain encapsulates the entities used across the service.

This architecture allows us to have a better separation of concerns and is a better fit for our testing needs. Most implemented services follow this architecture model, but not all are required to have all the presented components. Stateless services do not need repositories and therefore databases, while some services contain trivial business logic that makes the service layer just boilerplate. When that is the case, the controller can communicate directly with the client class, for example.

The following section (4.3.1) will explain in detail the implemented test levels, and the architecture will play a big part in understanding which tests are better suited for each component.

After understanding what each test level focuses on and how the whole enables validating the entire system, section 4.3.2 will demonstrate how the testing strategy was integrated into a deployment pipeline and the guarantees it provides.

4.3.1 Implemented Test Levels

4.3.1.1 Unit Testing

As we have seen in section 2.2.2.1, unit tests aim to test at the lowest possible level, and they need to be as fast as possible so we can quickly detect bugs and regressions. Because we cannot have fast tests while instantiating real infrastructure, the approach was to completely isolate the units under test from its dependencies by using mocking.

Most of the unit tests followed the solitary unit testing approach, so, we relied heavily on the usage of mock objects. Instead of implementing mocks from scratch, we used the Mockito[45] tool, which allowed us to define expectations on the downstream dependencies while also verifying they were called as expected.

Unit tests focused on classes that contained business logic and were more easily decoupled. Mapping back to the architecture, the classes better fit to unit tests were the controller, service, and domain classes.

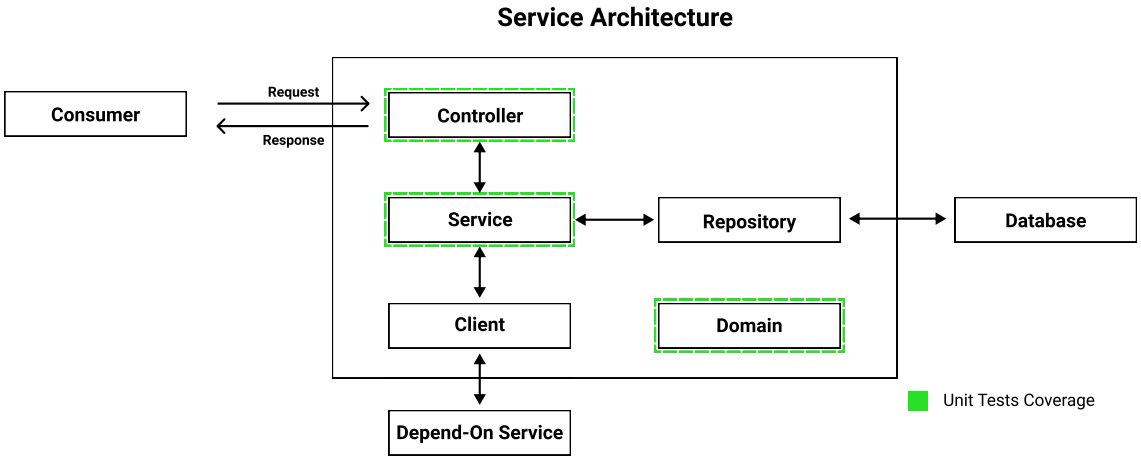


Figure 4.7: Unit tests coverage.

Figure 4.7 portrays the coverage attained with the definition of unit tests.

The controller tests focused mostly on ensuring the controller properly handled and mapped incoming requests. It meant testing that the requests were adequately validated and that any invalid requests threw the expected errors according to the service API. These tests mocked the service dependencies, which meant we were not exercising the service. Because the controller tests are isolated from the service, it is vital to have an understanding of the expectations the controller defined for the service class and use these expectations to define the service tests.

In the service tests, given that it is highly dependent on the client and repository classes, we used the same mocking mechanism to these dependencies. The service tests aimed at testing the service-specific business logic and making sure that the expectations the controller had for the service were met. If we do not make sure the expectations of the controller are met, both sets of unit tests will pass, and we will only detect the error in a later stage of the test cycle.

Beyond testing the controller expectations, the usage of mocks also enabled to verify that the downstream classes were called with the expected values. For example, the usage of mocks enables verifying that the service calls the repository with the right arguments without having to set up a database.

Unit tests in the domain classes are highly dependent on the complexity of the domain objects. If the objects are simple data holders, with basic getters and setters, there is no need to test as they are most likely getting tested in other tests as domain classes are

used across the entire service. If, on the other hand, the domain classes contain complex logic, like object validation or transformation, it is advised to create some tests to protect against regressions.

Because unit tests are the faster, it also means that we should aim to design, whenever possible, unit tests over other types of tests. Whenever a bug is found, and there is the need to define a test to prevent it from reappearing in our codebase, we should always aim at defining a unit test, if possible, and only if it is not possible we should go one level up the pyramid.

The client and repository classes do not usually have unit tests because of the nature of their dependencies. While the controller and service classes have dependencies on components that are totally under our scope, the client depends on a different service to perform its operations, and the same can be said of the repository class that is dependent on a database to perform its operations.

The complexity of these classes is usually not in the class itself but in how it integrates with the dependencies. For that reason, these classes are best tested with integration tests.

4.3.1.2 Integration Testing

Like we have seen previously in section 2.2.2.1, integration tests can be split into two main categories: persistence and gateway. In our architecture, the persistence integration tests will focus the repository class, and its integration with the database and the gateway integration tests will focus on the client and its integration with other services.

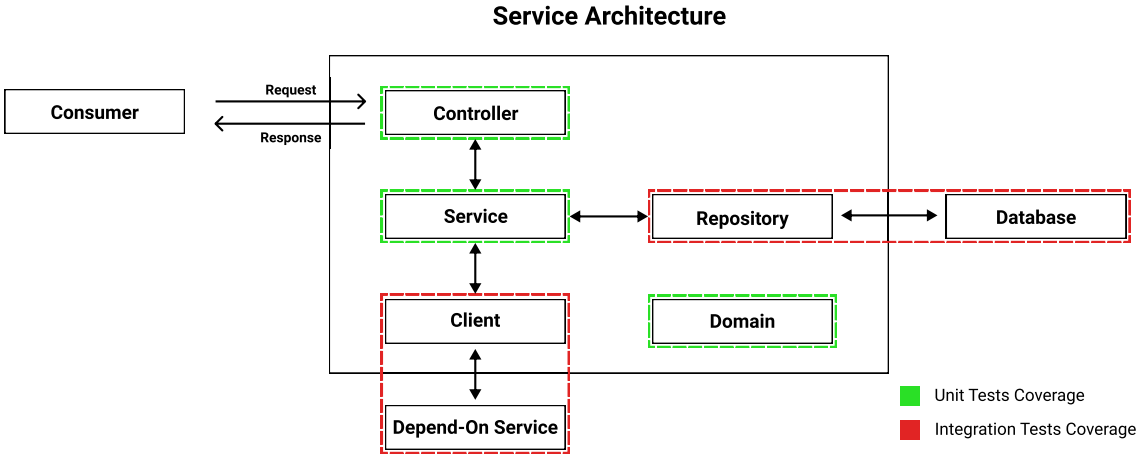


Figure 4.8: Integration tests coverage.

Figure 4.8 portrays the coverage attained with the definition of integration tests.

Persistence Integration Testing

When testing the integration between our repository classes and database (or other persistence solution), we must have in mind that what we want to test is the integration and not the underlying frameworks. We should not design tests that check if the database works

or if the framework used for communicating with the database is correctly implemented. What we want to make sure is that the code we write, mostly data queries, is fetching the correct data.

In our use case, because the database usage was fairly simple, using H2[27] as a replacement in-memory database for testing was pretty good because it sped up the testing. However, in some scenarios, the generic approach of using H2 might not be the most recommended. For example, in cases where the service uses features specific to a particular database, not supported by H2, a possible solution is to start a docker container with the specific database implementation and start the container for integration tests.

Our tests focused on testing the written queries by pre-populating the database with data and then performing the queries and asserting the correct data was retrieved. Beyond testing the success cases, we should also assert if our repository classes correctly handle and map database errors.

Gateway Integration Testing

With gateway integration tests, it is essential to understand that there is a considerable difference between testing against services we do not control and ones we do control. With services developed within our organisation we can always be aware of changes made. Furthermore, the way the service operates can also be analysed by looking at its implementation. Moreover, contract tests (to be discussed in detail in the following section, 4.3.1.3) also provide an extra layer of validation in the integration points.

External services are black boxes to which we only have access to the external API and documentation. These differences will have a significant impact on the way we design the tests.

When testing the communication with other services, we are dependent on those downstream services. This dependency can make our tests fail if the other service is unavailable or with problems. In order to remedy the dependency problem, the solution was to use virtualisation tools that advertise themselves as the real service dependencies and allow to test the integration between the HTTP client of our service, the remote communication, and the other services.

To allow testing with fake services, we defined profiles in our services that were activated by environment variables. According to the environment, the service would always understand the location of the dependency. The profiles use dependency injection, and so the code did not change between the different environments guaranteeing that the same code that is tested is the one used in production. This way, we push the test complexity into the test harness and outside of the service code.

Using virtualisation also allowed us to take control of the dependency and easily inject certain types of errors that would be extremely hard or even impossible to recreate on-demand. Virtualisation also proved to be very effective to reduce the test execution times, mainly when testing against the infrastructure provisioning dependencies. Because we

fully control the virtualised service we were able to inject latency into the communications and even make the virtualised service respond with bad data in order to test how our service handled this type of errors that can occur in distributed systems.

The integration tests take a more functional approach focusing on the client class. In these tests, we are trying to assert that we are correctly mapping the external concepts into our data structures and validating the integration logic. Once again, we also want to detect error cases and how our client class handles them, and that is one of the most significant advantages of using stubs and virtualisation.

Because we are defining the interactions we want to test, it is imperative to make sure that they are on par with the actual service. The parity with the real service is the biggest concern when using this type of mechanism.

External Services

For external services, we make sure the stubs are on par with the external service by recording the interactions with the real dependency. With the recordings, we have to make sure they are updated from time to time, to accompany the development of the external service. As it is out of our control, we cannot know for sure when to update the recordings, so depending on the stability of the external dependency, we should define an expiration time on the recordings. Surpassed the expiration date, new stubs should be recorded.

We chose hoverfly[82] to test against external services as it allowed to record the traffic and it allowed a better test separation, meaning each test could have its recording and this way avoid mismatches between tests that could result in unpredictable and hard to debug failures. Furthermore, studies show hoverfly as one of the fastest tools for this type of testing [63].

The method consisted in setting up hoverfly as a proxy between the real dependency and our service and allow it to record the interactions. If the test result is green we save the interactions and the next time the test runs, the service will, once again, ping the proxy but as it already has the saved recordings, it will answer straight away without forwarding the requests to the real dependency. Using recordings mainly works in protecting against regressions and allows speeding up the tests immensely. Most times, our services were creating or mutating infrastructure on AWS. For example, a simple request to create a machine can take minutes, and the way the service works is by requesting a machine to AWS that then answers with the machine details. The machine creation is not immediate, so, we need to poll AWS for the state of the machine and only when created and running, we can continue.

By recording the interactions, we will have saved all those polling requests. Most of them are repeated interactions in which we are asking if the machine is running and we are getting a response that it is still not ready. If we use the recordings without modifications, we can save some time because we can run the service locally, but we are

still making too many requests. To speed up the tests what we do is modify the recordings.

We check the recording file and change it to skip to the end request that says the machine is ready. Instead of requesting a machine and having dozens of requests saying it is not ready, we can have only a couple to test the polling mechanism and then reply with the final original interaction. This mechanism decreased the time of execution of some of the tests by up to 90%, when compared with the usage of the real dependency.

Internal Services

To test the integration points between internal services we used wiremock[91], as a way to set up the virtualised services, and to guarantee parity with the real service, the stubs defined in the tests were the same used for contract testing.

Using the stubs from the contract test allows us to use the same mechanism of communication with the provider team. We can be sure that, syntactically, the communication is always correct because that is guaranteed by contract testing. Furthermore, it brings the advantage of sharing our functional views of the service with the provider team, allowing them to approve and check that our view of the system behaviour is coherent with their implementation. Whenever the contract changes, the stubs for integration testing also change, and for that reason they are always on par with the provider service, because contract testing assures that parity.

4.3.1.3 Consumer-Driven Contract Testing

We decided to include contract testing in the strategy because in a microservices architecture, the value is mostly in the interactions between the microservices. Misunderstandings between microservices is an error surface we wanted to tackle directly. Contracts help to ensure that the services talk in the same *language* and that they understand how each of them communicates.

We opted for using consumer-driven contract testing mainly because we believe the primary goal of a service is to offer functionality to its consumers. This relationship translates into allowing the consumers to actively participate in the provider API definition.

Introducing this level of testing helps the provider services to make changes without breaking their consumers. Since the provider tests must always validate against the consumers' expected contract, we guarantee that the contract the consumers are expecting to be valid does not suddenly change and break the service.

It also guarantees communication between provider and consumer teams, as they always have to share the contracts between them. To implement consumer-driven contract testing the tool we used was Pact[55]. The usage of the tool in the related work (section 3.2) along with the helpful features it brings (Pact Broker), the maturity of the tool and the fact that it provides a way to be language agnostic while maintaining language-specific bindings were determining factors in its usage.

With consumer-driven contract tests, the strategy aims to test the communication between the developed internal services. These tests do not functionally test the services, so they work at the frontier of the service, targeting the controller and client classes.

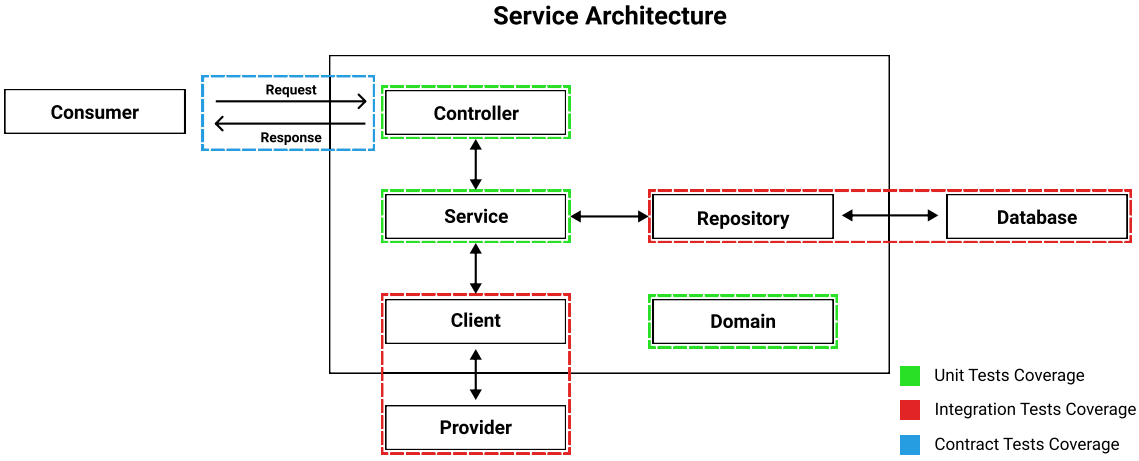


Figure 4.9: Contract tests coverage.

Figure 4.8 portrays the coverage attained with the definition of contract tests.

For the consumer service, because the only class that communicates with the provider is the client class, that is the only one that needs to participate in the test, alongside the supporting domain classes. The consumer tests define the expectations the consumer has for the provider service, and then the Pact tool creates a fake service with all the defined interactions.

For each interaction, we must validate that the client class can correctly process the defined expectations. If all tests are green, the Pact tool generates a contract that is a collection of the request/response pairs previously defined.

For some interactions, there is a need to define a state for the provider service. The state allows the provider service to prepare for the incoming request and prepare the adequate response. For example, if the provider service performs a search in a database for a person and we are defining the expectation, we can say that whenever we query the service for a specific user, it must have the field name that is a string. However, during the tests of the provider, it might not have the user in the database and returns an error that would result in a red test as if the provider was breaking the contract.

In this situation, we define a state that the provider must be in to validate the expectations. As these states are embedded in the contract, it allows the provider service to prepare for the requests, and in this specific case, have a user ready to be returned.

In the provider side of the validation, the only layer that needs to be instantiated is the controller alongside the supporting domain classes. All the downstream dependencies can be mocked as we only intend to validate the correct communication protocol and data structures. The mocking mechanism is the same we used when defining unit tests for the controller class.

The contract tests help in exposing errors on how the consumer creates requests and interprets responses and also exposes misunderstandings between the consumer and provider. The tests should be focused solely on the syntactical validation as the integration tests do the functional validation.

The generated contracts also double as a piece of documentation between the services that evolves with the contract.

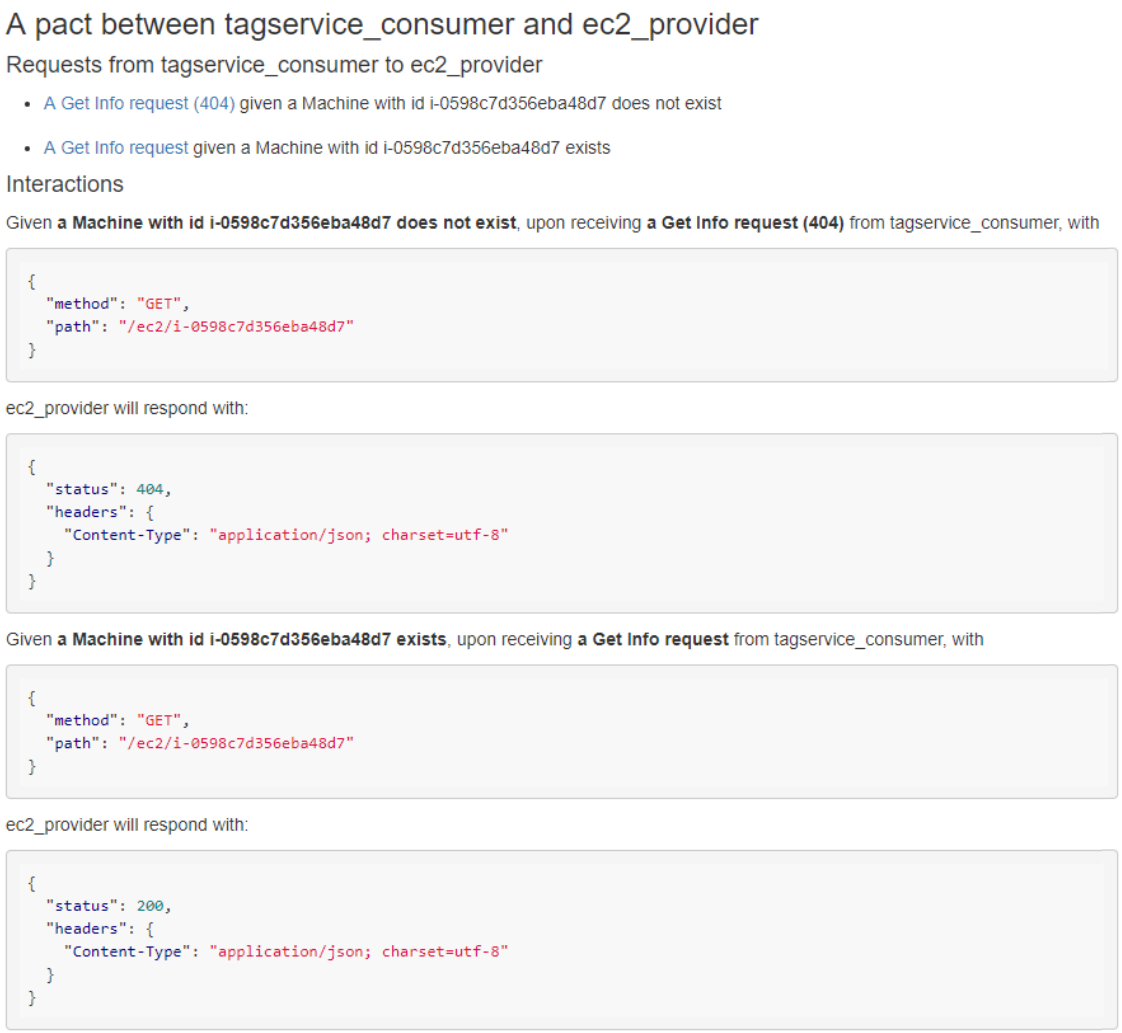


Figure 4.10: Pact file documentation.

In Figure 4.10, we can see the representation of an established contract between two services working as a piece of documentation. The Pact broker automatically generates this documentation. We can understand what requests the consumer performs and how it expects the provider to respond. In this simple example, the only information the consumer needs to understand is if a specific machine exists. It needs to understand how the provider responds when the machine exists and also when it does not exist. Because the request is the same, we used the concept of states for the provider to be able to differentiate the tests.

The provider will respond with more information and not only the status code, but because our consumer does not require that information, it only demands the status code to be present. Defining only what the consumer needs in the contract allows the provider service to change the other fields and evolve without breaking this contract.

4.3.1.4 Component Testing

The component tests' focus is to validate complete use cases, focusing a single microservice. Because this test level is the first in which all the microservice components are working together, validating the correct collaboration between the components is the primary goal. Component tests design should avoid repeating tests from the previous levels and to that effect they focus mostly on covering happy-paths that guarantee all the components can communicate adequately.

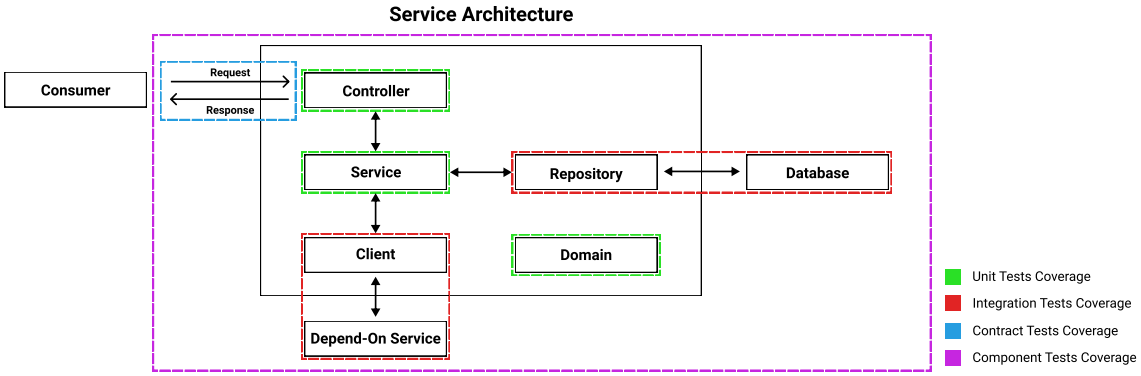


Figure 4.11: Component tests coverage.

Figure 4.11 portrays the coverage attained with the definition of component tests. Before defining the component tests the coverage we have almost covers the entire service. The definition of component tests aims to fill the gaps left by the previous test levels.

This test level is also the first one that can be completely agnostic to the implementation language and frameworks used. Because component tests are entirely driven through the service API, we can deploy the service and the needed dependencies(doubled) and perform requests like a consumer would.

The component tests defined can be classified as out-of-process and use the same virtualisation techniques used in the integration tests, namely recording and defining stubs for the dependencies.

These tests require more setup and have more moving parts, and so they are more expensive to run. For that reason, the component tests attempted to mimic the essential operations a user would perform and avoided defining edge cases.

Because the tests resemble real consumer requests, they can also serve the purpose of documenting the microservice behaviour.

4.3.1.5 End-to-End Testing

At this stage, we have already validated the microservices functionality in isolation. The goal of the end-to-end tests is to understand if the system delivers business-critical functionality. The system is the composition of the microservices with all the required dependencies.

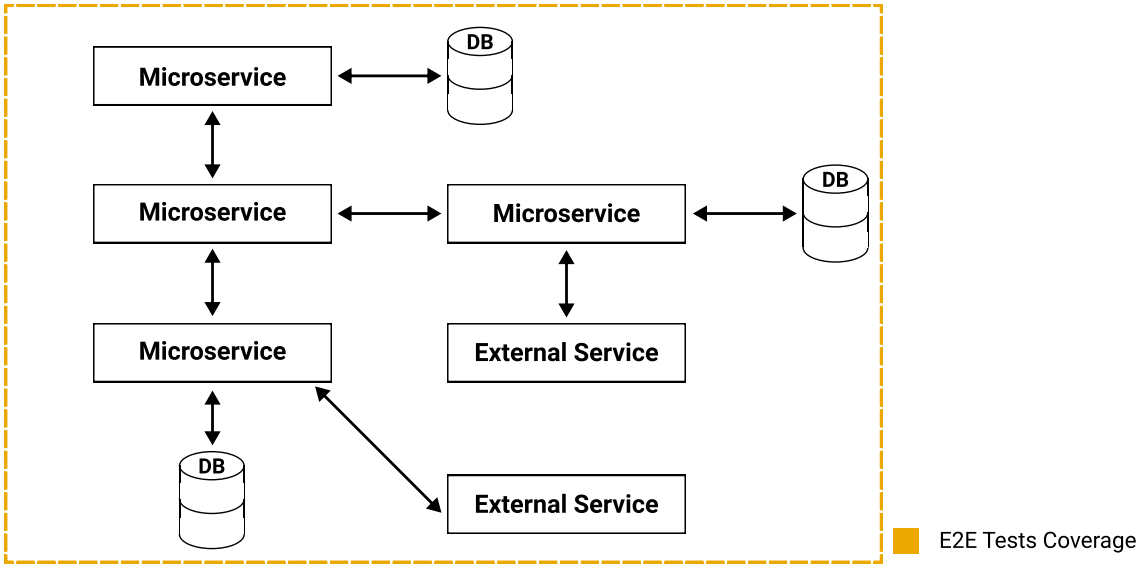


Figure 4.12: End-to-End tests coverage.

Figure 4.12 portrays the coverage attained with the definition of the end-to-end tests.

As we have seen in chapter 2.2.2.1, end-to-end tests are usually flaky due to their complex nature as they have many moving parts. Beyond their flaky nature, the provided feedback by this type of tests is usually not very accurate or precise, which means a failure in this type of tests demands a much bigger debugging effort.

Despite their flaws, this type of tests are beneficial to validate that the system as a whole can deliver the expected value. With this in mind, the definition of end-to-end tests aimed to minimise the flakiness while ensuring the critical business paths worked as expected.

This translated into grouping microservices together into a specific domain which means reducing the amount of moving parts and reducing the possible points of failure that are usually present in highly distributed systems.

The other high value of end-to-end tests is that they are much more expressive in the sense that they can be shown to managers and decision-makers as a representation of critical user journeys.

Because this is a significant advantage of end-to-end tests, the solution defined the critical user journeys and tested the commonly denominated happy-paths. The definition of the tests ensures that the different pieces can work together to deliver the needed functionality. Just like with component tests, the remainder non-happy paths were already

tested in the lower levels of the pyramid with much narrower focus. Covering those cases in the lower levels instead means the coverage of those cases is much cheaper.

By grouping multiple microservices to define end-to-end tests and covering only the most critical paths, we reduce the number of tests of this type and also address the considerable time cost usually associated with this type of testing.

4.3.2 Pipeline

This section will describe the implemented pipeline that made it possible applying the testing strategy presented in the previous section. Firstly we will introduce the pipeline at a lower granularity level and then dive into the specifics of each step.

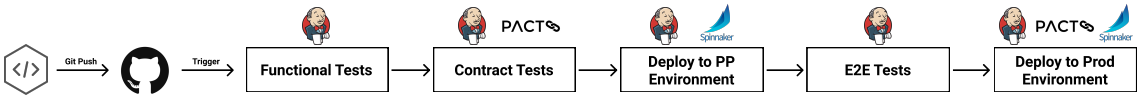


Figure 4.13: Microservices pipeline.

Figure 4.13 represents the pipeline for the microservices. It starts with the developer pushing his code to a source control system, in our specific use case, GitHub. The new commit triggers Jenkins to run the first job in which we run a first stage of functional tests. If the stage succeeds, we run a stage of contract tests, and only then we deploy the microservice to the pre-production environment. In the pre-production environment, the end-to-end tests run, and if successful, the new version of the service is deployed to the production environment where it is available to the end-users.

4.3.2.1 Functional Tests

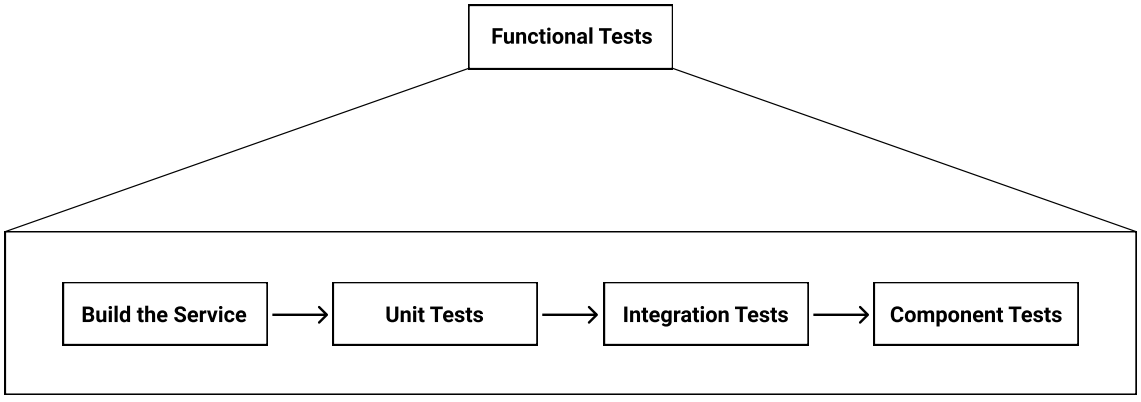


Figure 4.14: Functional tests in detail.

The functional tests step is straightforward and entirely managed by Jenkins. In this step, we guarantee that the service builds without errors and that it passes all the defined unit, integration and component tests.

All these tests were defined using JUnit 5 [39] and the usage of the virtualisation tools was simplified because the tools used had bindings specific for JUnit which managed the entire lifecycle of the virtualisation tools. The usage of the language-specific bindings was the more convenient way to use the tools and also the fastest.

Still, the testing strategy should be independent of frameworks and languages. The virtualisation tools chosen also provide stand-alone executables that allow configuration using command-line tools and HTTP protocols and therefore provide ways of being language agnostic.

Even though this method was not used in the final pipeline, it was implemented and proven to work, although it took considerably more time than the alternative. In the case of component tests, it meant deploying the microservice to an isolated environment, deploy a server containing the virtualisation tool and configure the interactions specific to each test. The extra deployment steps increased the duration of the component tests and served merely as a PoC.

4.3.2.2 Contract Tests

To understand the contract tests portion of the pipeline, it is crucial to understand the role of the pact broker and the pact matrix in the validation of the contracts.

The pact broker is the application that holds the contracts and the verification results. The consumer publishes the generated pacts to the broker, and the provider services use the pact broker to have access to the pacts their consumers published in order to validate them.

The Matrix

Consumer	Consumer Version	Pact Published	Provider	Provider Version	Pact verified
windows_consumer	d947284 prod master	6 days ago (revision 1)	ec2_provider	3f366db prod	6 days ago (number 219)
windows_consumer	9b2d715 master	6 days ago (revision 1)	ec2_provider	3f366db prod	6 days ago (number 212)

Figure 4.15: Pact Matrix.

Figure 4.15 represents a small example of the pact matrix. The pact matrix combines the published pacts and the versions of the consumers and providers that have published and validated each pact.

In Figure 4.15, we see the relationship between a consumer and a provider. We can see that the consumer published the first version of the pact with the commit *9b2d715* and that the provider with version *3f366db* was able to validate it. After that, a new version of the consumer was created, and we can see that the same version of the provider was also capable of verifying the pact. Looking at the matrix, we can know which versions are compatible with each other to guarantee safe deployments.

If a contract does not change between incremental versions of the consumer, the matrix assumes that the provider version that validated the pact before is still capable of validating it and in those cases, the provider does not need to run the tests to guarantee the consumer can safely deploy.

Because consumers and providers have different responsibilities in the contract tests, they also have differences in the pipeline.

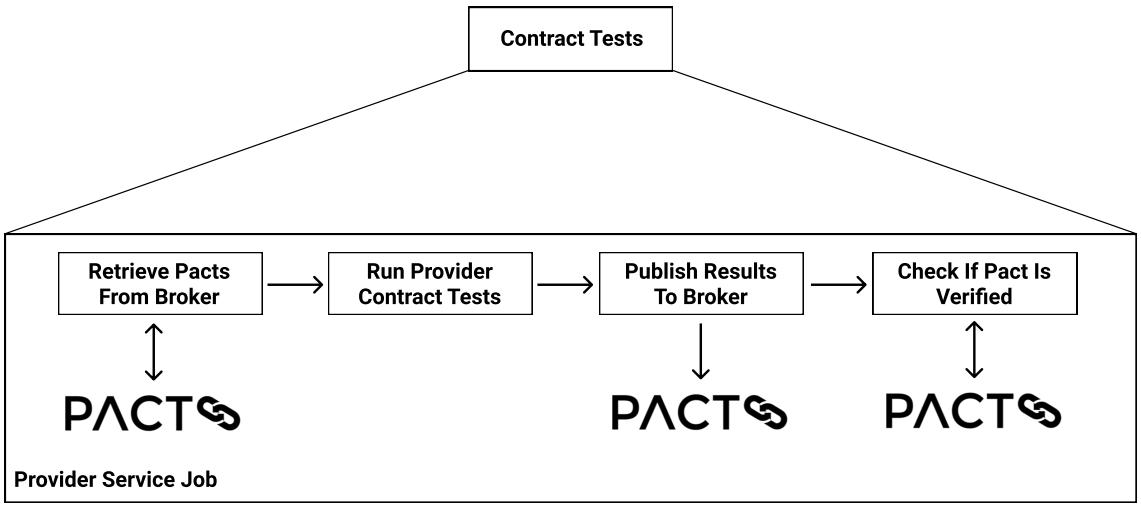


Figure 4.16: Provider contract tests in detail.

Figure 4.16 shows the steps taken by a provider service concerning contract tests. Firstly, it starts by retrieving all the pacts from its consumers marked as being currently in production. We only want to verify the pacts created by consumer services that are currently in production because that is the environment we are aiming to deploy in, and we only want to make sure the services are compatible in that specific environment.

After that, we run the tests that check if the service can validate the retrieved pacts and publish the results to the broker. We then check if any pacts failed to be verified. If any failed, it means that our provider service version would not be compatible with the consumer versions in production and deploying is not safe. Otherwise, if all pacts were verified, we can deploy it to the pre-production environment to run the end-to-end tests. Because the service versions in pre-production and production are the same if the matrix says we are safe to deploy to production, we can also safely deploy to the pre-production environment.

Figure 4.17 portrays the steps taken by the consumer service. It starts by running the contract tests and generating the pact files. It then publishes them to the broker.

After publishing the pact, we would usually be blocked from deploying, because the provider would have not yet verified the pact. Unless the pact remained the same, we would always be blocked and would need to wait for the provider to verify the new version.

To avoid being blocked, we configured a webhook that runs a custom job whenever

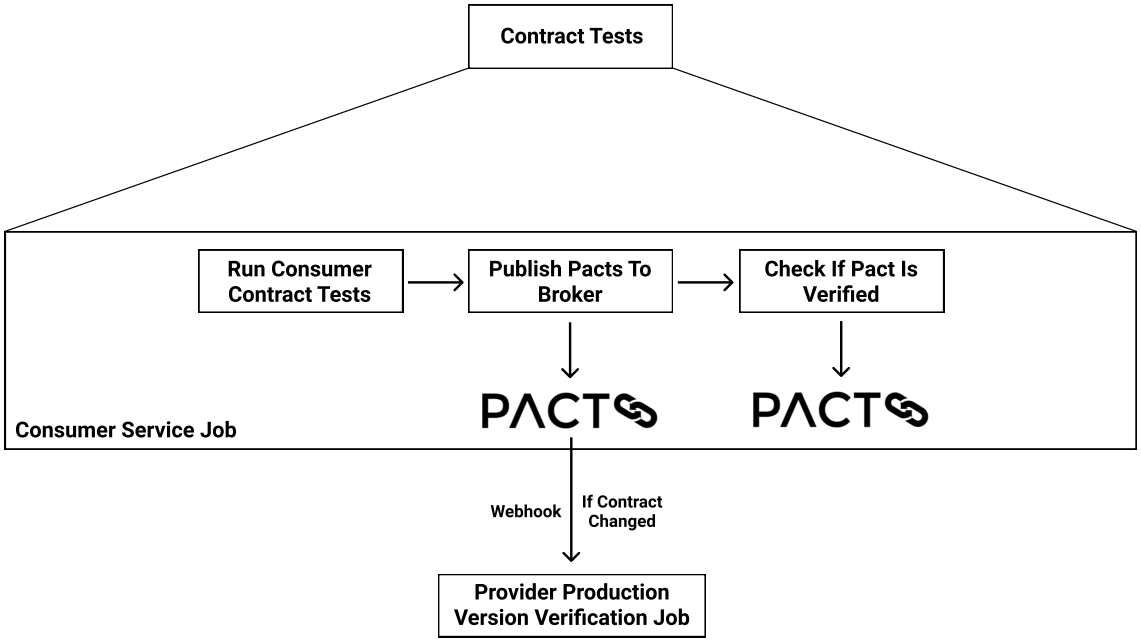


Figure 4.17: Consumer contract tests in detail.

the contract changes to verify if the provider version in production is compatible with the new consumer version.

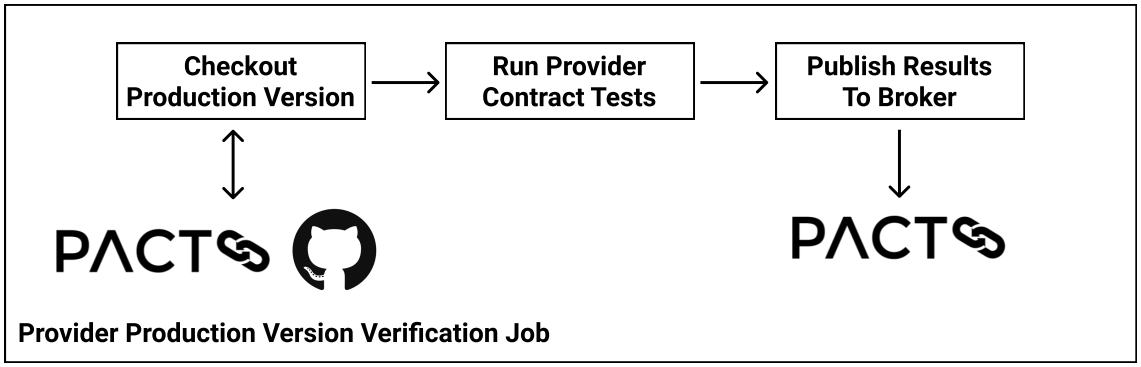


Figure 4.18: Provider Production verification job in detail.

Figure 4.18 represents the provider production version verification job. It starts by querying the pact broker for which version of the provider is in production, and that version is checked-out of source-control. After checking-out that service version, we run the contract tests against the pact and publish the results to the pact broker. This result will contain the information that determines if the new consumer version is compatible with the provider version in production.

After the result is published, the consumer job can query the pact broker in order to understand if the pact is verified. If the provider verifies the contract, the consumer can deploy the new version of the service.

If a service is, at the same time, a consumer and a provider, it runs both pipelines. If

both pipelines are successful, the service is deployed.

With the implemented pipelines we guarantee that a consumer is not deployed if its contract was not successfully verified by the provider version in production and that a provider version is not deployed if it would break the contract with its consumers' versions currently in production. We also made sure that deployments were not blocked unnecessarily by implementing the provider production version verification job.

4.3.2.3 Deploy to Pre-Production Environment

Before deploying the service to any environment and because we follow the immutable infrastructure pattern, we need to build an immutable image to serve as the basis for our deployment.

We chose to deploy the service on Linux based servers, and so we needed to be able to install the service on those servers. To solve this problem, we used nebula[49] to package the spring boot services into Debian packages that were capable of being installed in the Linux servers.

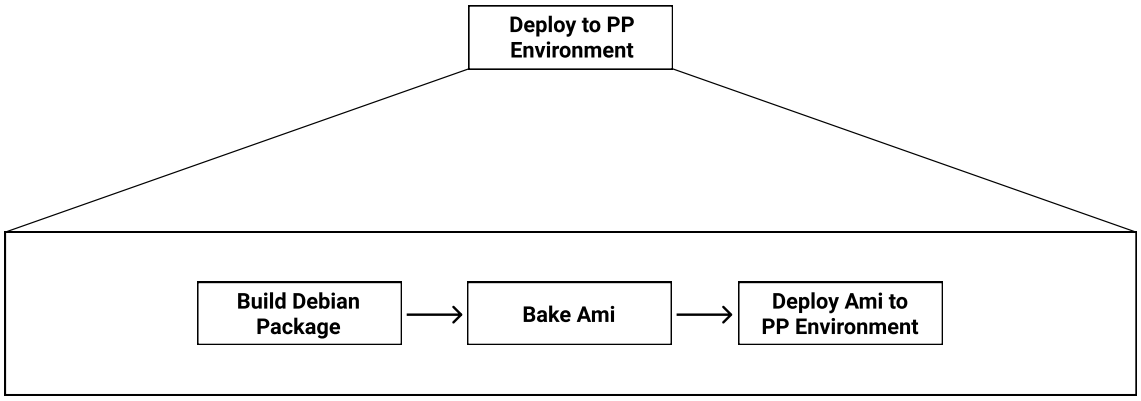


Figure 4.19: Deploy to Pre-Production environment in detail.

In Figure 4.19 we can observe that the first step is the creation of the Debian package and only then we enter the bake stage. Bake is the term Spinnaker uses to denominate the creation of an immutable image. In this process, we provide a base image; in our case, a new Linux machine on which we install our Debian package alongside any needed tools for the correct operation of the service. After successfully creating the machine, it is packed into an Amazon Machine Image (AMI) and that is the final product of the bake stage.

With this AMI, we can deploy new server instances that will always have the same configuration, and we use this image to replace the existing service version that is currently in the pre-production environment.

4.3.2.4 End-to-End tests

The end-to-end test stage uses Postman[57] to perform the requests defined in the tests and assert the responses obtained from the services, in the pre-production environment, match the expected outcome.

4.3.2.5 Deploy to Production Environment

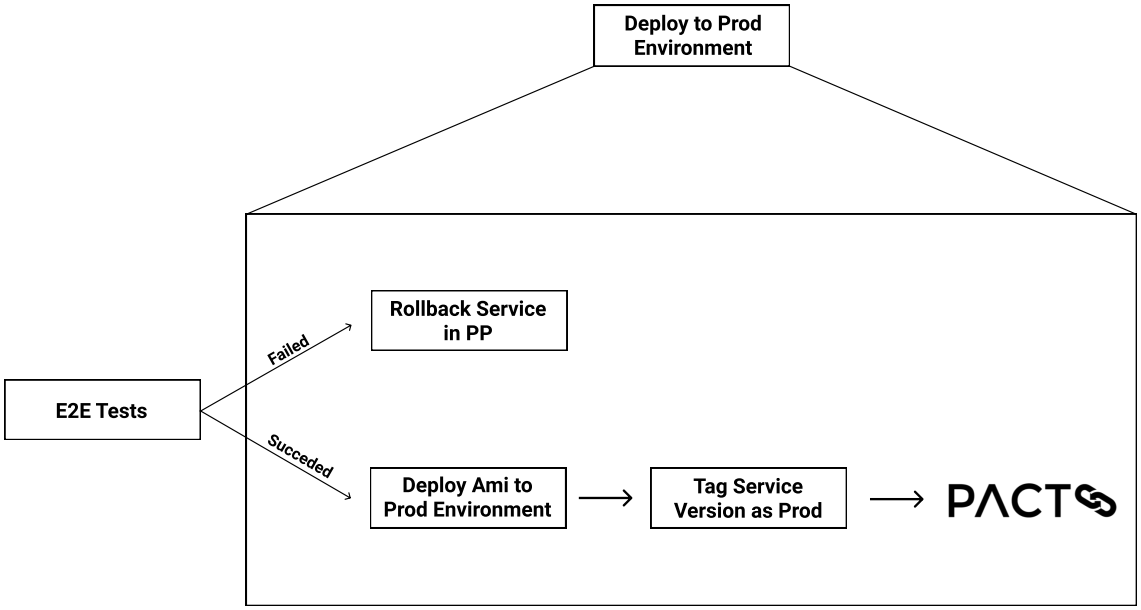


Figure 4.20: Deploy to Production environment in detail.

In Figure 4.20 we can observe that when the end-to-end tests stage ends there are two possible outcomes: a success or a failure.

In case the end-to-end tests are successful we can deploy the same AMI to the production environment and replace the current service version. As a last step we must update the information on the pact broker of which version is in production, so the contract validation stage looks at the correct service versions.

If, on the other hand, the end-to-end tests fail we certainly do not deploy to the production environment, and because we want to maintain the pre-production environment on par with the production environment, we rollback the service previously deployed to the version that was in pre-production when the pipeline started.

This stage marks the end of the pipeline. For a complete view of the entire pipeline, figure A.1 contains the complete pipeline in detail.

Chapter 5

Evaluation

In order to evaluate the developed strategy, it was necessary to determine its effectiveness against the developed prototype. Even though there is currently a testing strategy for the orchestrator system, its design targets the current monolithic architecture of the system. Because the strategies are targeting different system architectures at different scales, comparisons between the two can be faulty.

It is essential to understand what are the actual benefits of the strategy and not confuse them with the architectural changes benefits. For example, comparing the build times of the prototype microservices against the monolithic system components would undoubtedly show a massive difference. The difference would favour the microservices, but the architecture of the components justifies it, and the pipeline implementation would have almost no influence.

For that reason, this chapter will avoid direct comparisons between the strategies in terms of metrics and will instead highlight the capabilities of the developed strategy and problems it solves.

Just like the implementation stage separated the Chef and microservices context, the evaluation will also follow the same approach. Nevertheless, it is still essential to understand that the sole concept of separating the microservices and the Chef recipes is a contribution of this dissertation. In the current testing strategy, the definition of tests encompasses both contexts and lacks proper isolation. Besides that, it would not be reasonable to combine the test data as the chef recipes are not used in conjunction with the microservices and so there is no real use case that would effectively test the two contexts working together.

	Number of Tests	Test Representation (%)	Execution Time per Test (seconds)	Execution Time Sum (seconds)
Unit	47	92.16	1.49	70
Integration	4	7.84	150.5	602

Table 5.1: Detailed Chef test distribution.

5.1 Chef Evaluation

The main goal of creating a prototype specific to the Chef context was to determine if it was possible to create tests at a higher granularity level than the existent end-to-end tests and also understand the kind of errors they could detect, if any.

As evidenced in the implementation section 4.2, it is possible to create unit and integration tests for Chef. These are both at a higher granularity level than end-to-end tests.

Table 5.1 represents the data gathered from the execution of the unit and integration tests in the created recipes, averaged across ten consecutive executions. The time metrics presented were gathered from a local machine and not from the execution in the CI/CD environment. The integration tests did not run in that environment so to be able to compare the times between the execution of the unit and integration tests we needed to have the same baseline. Still, from the execution of the static analysis and unit tests between the local machine and the CI/CD environment, there is a calculated 37% time decrease when running in the CI/CD environment.

From the execution times comparison between unit and integration tests, it is possible to understand there is an enormous discrepancy. An integration test takes approximately 100 times longer than a unit test with each integration test occupying on average two and a half minutes. From this data, we can understand the need to obtain as much coverage as possible with unit tests, and leave only to integration tests what is not possible to assert with unit tests.

For that reason, the integration tests only asserted the happy-path scenarios, leaving for the unit tests most of the work to obtain test coverage. This fact translated into having over 92% of the tests being unit tests and the rest integration tests. If end-to-end tests integrated the test suite, the representation of these tests would naturally decrease, but end-to-end tests should still be the least represented test type.

Naturally, the created test distribution is tied to the usage patterns of the created recipes. The Chef recipes functionality can be extensive, and even though there was an attempt to approximate as much as possible the usage patterns to the usage in the orchestrator system, the scale factor will always influence the test distribution. Still, the tests representation should remain in the same order even if each test level percentage changes by a small margin.

More important than having a rigorous test distribution at this stage is the fact that

the implemented tests were detecting real errors at a higher granularity level. This fact enables detecting errors sooner with a diminished feedback loop. Unit tests proved capable to detect errors mostly related to specific recipe logic and to assert the recipe execution flow. Integration tests demonstrated that they could fill some of the gaps left by the unit tests simulated convergence model and reassure recipe validity. Demonstrating value in unit and integration tests was the most valuable contribution in the context of Chef testing.

Regarding the static analysis tools usage, the conjoint usage of tools proved very effective at ensuring all recipes followed the same patterns, guaranteeing uniformity, and avoiding anti-patterns. The automatic correction capabilities of the CookStyle tool were also very useful and can reduce the time it takes to update recipes to follow new guidelines and patterns.

5.2 Microservices Evaluation

To objectively evaluate the developed testing strategy and inherent pipeline integration, we should first recapitulate the objectives traced. The testing strategy aimed to be as efficient as possible regarding the time taken to write and run the created tests. Still on the topic of efficiency, the developed tests should strive to provide a quick feedback loop while still being able to test conditions and errors usually associated with distributed systems. It should also concentrate on the research of techniques that allow the definition of tests that are more focused and more reliable while providing a balanced test suite.

5.2.1 Test Distribution

Table 5.2 contains the recorded data on the execution of the unit, integration, component, and end-to-end tests. The table does not include data from the contract tests because there is no precise mapping of what is the scope of a single test in the contract tests. Differences exist on the consumer and provider side that challenge the scope of the tests and mixing the execution data from both sides would create a false average as it would be very constrained with the proportion of provider and consumer services.

Even if we split the contract tests between providers and consumers, we do not believe these should forcefully obey the pyramid distribution as they are very dependent on the kind of relationships each service has. For example, a consumer service that only requires one endpoint from a provider will have much less contract tests representation than a service that depends on multiple endpoints from multiple provider services to perform its operations.

Literature that includes the contract tests in the pyramid also have different views on their location. Some argue they belong above component tests [67] while others argue it belongs directly above unit tests [29]. Different methods to perform contract tests can drastically change the execution times. In our case, because we do not involve the entire

	Number of Tests	Test Representation (%)	Execution Time per Test (seconds)	Execution Time Sum (seconds)
Unit	118	65.92	0.54	64
Integration	39	21.79	1.77	69
Component	17	9.50	3.35	57
End-to-End	5	2.79	194	970

Table 5.2: Detailed Microservices test distribution.

service and dependencies during the contract tests, our tests will execute faster than if we needed the entire service and dependencies to execute the tests. This difference in methodology can justify the different locations in the pyramid.

Moreover, we believe contract tests should not appear in the pyramid as its representation is highly dependent on the service dependencies, and this skews the contract tests representation in a way that we cannot foresee, let alone enforce.

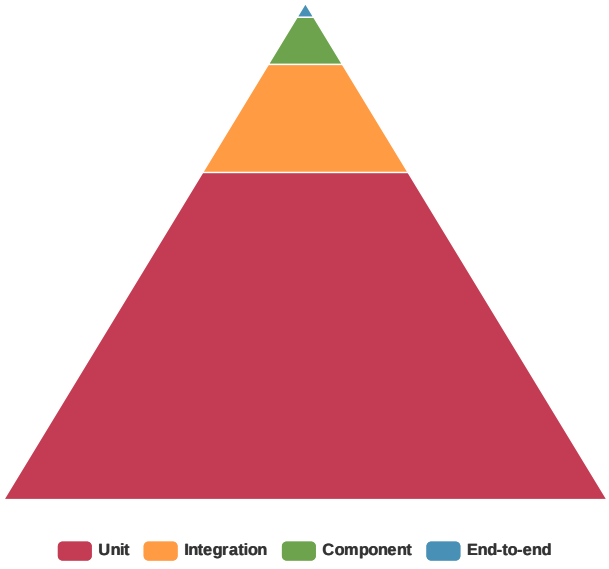


Figure 5.1: Prototype test distribution.

If we take a look at figure 5.1, we can see that the application of the strategy to the prototype successfully followed the test pyramid distribution. The figure maps the test representation information of table 5.2.

The data was taken directly from the pipeline executions in the CI/CD environment which means the times include some overhead related to the setup of the tests and tools used, the selection process of each test type that is done in runtime and also any concurrent processes running in the environment. Most importantly, all tests are subject to the same constraints, which results in the same baseline. The test data is also the result of averaging ten consecutive pipeline executions of the same source code and tests.

Noteworthy is that while defining the tests, we were not actively thinking about

respecting the pyramid. While defining the tests, the focus was instead in following the delineated strategy. Fact is that the strategy had in mind respecting the pyramid but respecting the strategy does not require actively worrying about the test distribution. The created guidelines and error mappings for each test level along with the standardised architecture allowed that, organically, the intended test distribution was respected.

This fact is, on what concerns usability and test creation efficiency, a measure of success. Following the strategy and ending up following the distribution is more natural to the developer/tester than after creating all the tests having to revise their definition so that they follow the intended distribution.

The prototype was composed of five different microservices, and its functionality concentrated on the creation and configuration of EC2 machines to exercise the AWS dependency. Some services were stateless while others managed their state using databases. Similarly to the test distribution in the Chef context, the test levels representation can have slight variations depending on the usage. For example, a system consisting of only stateless microservices should result in a reduced representation of integration tests. Once again, what is vital is that the distribution maintains the pyramid shape.

If we look at the execution times per test from table 5.2, we can see there is an increase in test duration as the granularity of the tests decreases. This increase in test duration is expected but, usually, the integration and component tests take much longer; however, in our case, the difference is attenuated by the usage of virtualisation tools.

5.2.2 Virtualisation Usage

The testing strategy resorted to utilise virtualisation in order to tackle the slow dependencies of the orchestration. Operations responsible for the creation and configuration of EC2 machines in the prototype showed the benefits of this approach.

Some of these operations, just like in the orchestrator system, demonstrated to be hard to test without the usage of virtualisation tools. For example, the simple creation of a machine in AWS that is a reasonably basic operation in the context of orchestration could take more than a minute to perform. Performing tests for this operation alone represents a challenge, because of the time it would take, but with this operation being such a building block for the rest of the configuration operations we can understand that this problem propagates along the chain because of a dependency problem.

If we need to test a different operation that performs some configuration on a machine, we need to have the machine as a pre-condition. For that, we would naturally call upon the operation that creates the machine which would also increase the test of the configuration operation along the same order of magnitude of the first operation.

This dependency problem can scale uncontrollably, and virtualisation usage helps mitigate it because we are no longer dependent on creating or configuring real machines but instead in understanding how the response would look like, using the recordings or creating the stubs by hand helped by the pacts.

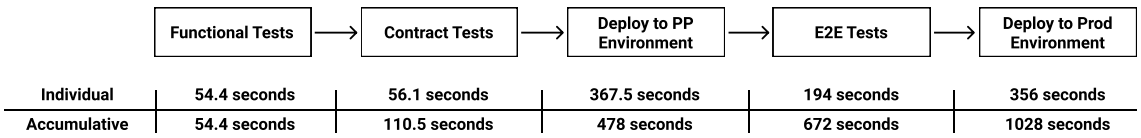
The usage of virtualisation was the most significant factor for the reduction of the integration and component tests execution time. Comparing the execution times without the usage of virtualisation against the execution times using recorded interactions showed, on average, an 85% time reduction when using virtualisation. For example, for the microservice responsible for creating the EC2 machines, the time reduction in the component tests equated, on average, to close to a minute (61 seconds, to be exact).

Beyond being able to run tests faster, the virtualisation usage also brought some other advantages, namely, a monetary cost reduction as the tests no longer needed to instantiate real infrastructure as soon as the recordings or contract stubs were in place. Moreover, the fact that the stubs remain static for consequent runs also reduces flakiness that can originate from unstable dependencies or other distributed systems related errors.

On that topic, the usage of virtualisation tools also helped achieving one of the goals for the strategy of being able to test against errors commonly found in highly distributed systems. We were able to configure the tools to simulate latency and even unresponsiveness and assert that our services were capable of handling that type of situations.

Having fast tests is important but it is crucial that they translate into a fast feedback loop.

5.2.3 Feedback Loop



Individual	54.4 seconds	56.1 seconds	367.5 seconds	194 seconds	356 seconds
Accumulative	54.4 seconds	110.5 seconds	478 seconds	672 seconds	1028 seconds

Figure 5.2: Microservices pipeline with time statistics.

Figure 5.2 shows the attained feedback loop for the different pipeline stages. The feedback loop is mostly a result of an effective application of the test pyramid distribution. By obtaining more coverage with the low-level tests, we are also guaranteeing a faster feedback loop.

In figure 5.2 the functional tests encompasses the entirety of unit, integration and component tests. We have seen that the usage of virtualisation drastically reduces the time of execution of these type of tests. This fact raises the question if the test distribution for these tests should be maintained or if we could use instead, the honeycomb distribution discussed in Spotify's related work section, 3.2.

Arguably, the feedback loop obtained with the honeycomb distribution would still be reasonable, but it has two main drawbacks. The first is related to the focus of the tests. While having a fast feedback loop is very important, it is also imperative that the feedback allows to fix the error promptly, and that translates into getting feedback that is accurate and precise. For that reason, the creation of a more extensive set of unit tests is fundamental and also has a role in protecting against regressions.

The second reason has to do with the nature of our dependencies. Because of the usage of virtualisation instead of the real dependencies we obtain faster feedback, but the setup and time taken to create this type of tests also increases as it requires creating or recording stubs for later usage. The slower creation process is justified when we only need to define a small set of tests, but as the number of tests increases, the RoI certainly goes down. For this reason, these tests (integration and component) are used with care and to tackle very specific gaps.

Those reasons justify the application of the test pyramid. In order to justify the shallow percentage representation of the end-to-end tests, we can analyse the feedback loop obtained if we only had end-to-end tests. If we had no functional tests or even contract tests in the pipeline, there would still be a need to deploy the new service version into a pre-production environment to perform the end-to-end tests. Observing figure 5.2 it is possible to understand what feedback loop this would represent as we would remove the first two stages of the pipeline. We would obtain the first feedback from testing in approximatelly 561 seconds (More than 9 minutes). If we compare this feedback against the entire functional tests stage that provided feedback in under a minute, we can see the difference justifies the distribution.

If no lower-level tests were defined, it would also mean that the end-to-end test would need to be much more exhaustive and would take much more time. In fact, because of the strategy to define end-to-end tests was to check mostly the happy-path scenarios, this comparison is as favoured as it can be for the end-to-end tests and still their usage as the first source of feedback shows to be very inadequate.

5.2.4 Contract Testing

Microservices architectures concentrate much of their value on the interactions between the microservices. Using contract testing allows to validate the interactions between the developed microservices sooner in the pipeline. If contract testing is not present, the validation of the interactions between the different microservices only occurs in the end-to-end testing stage.

Therefore, contract testing provides the ability to shift-left the interaction validation and therefore reduce the feedback loop. Beyond reducing the feedback loop, contract testing brings advantages concerning the evolution of the system as consumers participate directly in the providers' API definitions.

They are crucial in blocking deployments that are known to go wrong at the interaction level and in our use case protect the pre-production environment by failing the pipelines. Whenever contract testing detects a failure, it is also a very accurate type of feedback. We know exactly which pair of provider/consumer is failing and what interaction is breaking. That feedback allows the teams that own the services to communicate and work together on the evolution of the contract.

Once again, we can prove the sorter feedback loop provided by contract testing by

looking at figure 5.2. The implemented pipeline allows getting contract feedback in about 110 seconds while the feedback loop provided by only having end-to-end tests like we have seen before would be in the order of 561 seconds, which increases by 400% the feedback loop.

Chapter 6

Conclusions

Software testing is an activity that is continuously subject to change. Changes concerning system architectures are one of the main drivers of the testing methods mutation.

The widely accepted monolithic architecture has, in the last years, been contested by the microservices pattern. This new architectural pattern has changed the way software is developed, deployed and tested.

Testing microservices architectures has been subject to much research in the industry since the pattern became popular. Still, there is no golden bullet solution for testing microservices-based systems. There are many views on the subject, and like any other system, microservices lifecycle is highly dependent on its dependencies and constraints.

In this dissertation, there was a considerable effort in understanding the constraints of the OutSystems orchestration context in order to devise an appropriate and well-fitted testing strategy. Beyond understanding the context, the efforts put into researching microservices testing approaches used in the industry greatly helped shaping the devised solution.

This dissertation goals were mainly related to the creation of the testing strategy. The testing strategy should allow writing and running tests efficiently while allowing to detect errors sooner to shorten the feedback loop. It also aimed to provide support for testing commonly found errors in distributed systems in a deterministic way and be integrated in a CI/CD pipeline.

All of the objectives mentioned above were met, as the developed strategy alongside the created prototypes in which it was applied yielded very positive results.

The work developed in this dissertation is, therefore, an essential step into the orchestrator system architectural change. The work produced by this dissertation is the first step towards the migration, but there is still work to do in order to provide the best possible workflow efficiency and validation for the new system. Time constraints

of the dissertation, coupled with the broad scope of software testing, created the need to prioritise some fields of work. For that reason, this dissertation focused on the pre-production testing phase and more specifically, on the functional aspect of validation. The research done on the subject of software testing identified more areas of work that could further improve the developed testing strategy, and for that reason, section 6.2 will propose research topics with that intent.

6.1 Contributions

This dissertation produced the following contributions:

- Definition of a testing strategy applicable to the OutSystems orchestrator context: describes methodologies for both the Chef and microservices ecosystems. It proposes an adequate test distribution and focuses on mapping what type of errors should be tackled at each level along with techniques to reduce the infrastructure dependencies problems;
- PoC prototype implementation: the creation of a highly-contextualised prototype proves the applicability of the testing strategy. The prototype follows the defined strategy and brings the concepts to a practical level, enabling proper validation of the strategy;
- PoC pipeline implementation: the pipeline provides the support for the prototype and proposes techniques to enable more reliable and faster deployments;
- The implementation and validation process of the dissertation proved that the solution is promising for the OutSystems context and towards the architectural change of the orchestrator system.

6.2 Future Work

The following topics raise research topics that can improve the efficiency of the developed strategy and associated workflow, as well as propose new topics to enhance the security and safety of the deployments.

- Static analysis testing: static analysis in the Chef context demonstrated to be a very useful technique, even though it mostly covered style issues and checked code design properties. This type of static analysis should also be implemented in the microservices context. Moreover, static analysis can also play a part in other areas, like predicting how the code will behave at run time or, for example, check the code for the possibility of using concurrency mechanisms increasing the speed of execution. Thus, there is a research opportunity for static analysis that can improve the strategy even further by validating code quality earlier in the pipeline;

- Persistence integration testing with containerised databases: the developed prototype used databases but their usage was reasonably basic. Enterprise-grade services are sometimes coupled with particular features of database management systems. In those situations replacing the databases with a generic in-memory solution might not work. Deploying the specific database solution in a container would allow performing the necessary integration and component tests. This solution should theoretically be a little slower than the in-memory solution, but it also provides more compatibility guarantees;

- Build and deployment of services based on containers: the created services were deployed using Linux machine images as it provided the fastest ramp-up time to this dissertation. Still, it is widely accepted that deployment based on containers is considerably faster and more lightweight. Using containers would allow reducing the time taken by the deployment stages and even allow sharing the containerised services for testing purposes across the teams;

- Production testing techniques: the dissertation focused on pre-production testing techniques as there should always be a first stage of pre-production testing before production testing. However, production testing would allow not only to have another layer of protection for the end-users, but it would enable testing against real users and data, where the test cases would no longer be pre-determined;

- Monitoring: in order to accurately evaluate the quality of the developed services, it is imperative to have proper monitoring mechanisms. These are an essential precondition to employ production testing techniques and are also crucial for creating fast recovery strategies;

- Chaos Testing: according to Murphy's law: "Anything that can go wrong will go wrong", and that is the premise for chaos testing. As failure is unavoidable, chaos testing techniques deliberately introduce random failures in the system to ensure it can correctly deal with the failures and recover quickly. This type of testing allows testing against the worst-case scenarios and make sure the system is highly reliable, even in the worst possible conditions;

- Performance Testing: performance testing techniques focus mainly on the speed, scalability and stability of the systems. Therefore, they ensure that the systems respond quickly and determine how much load can the system be subject to and whether that load causes instability or not;

- Security Testing: this type of testing intends to discover the vulnerabilities of the system and determine if the underlying data and resources are protected against possible malicious attacks from unknown sources.

Bibliography

[1] *A better approach for testing microservices: introducing test kits in practice*. URL: https://xpdays.com.ua/programs/a-better-approach-for-testing-microservices-introducing-test-kits-in-practice/ (visited on 02/09/2019).

[2] *About Recipes — Chef Docs*. URL: https://docs.chef.io/recipes.html (visited on 01/30/2019).

[3] *An Introduction to Load Testing | DigitalOcean*. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-load-testing (visited on 02/14/2019).

[4] *AntiPatterns*. URL: https://sourcemaking.com/antipatterns (visited on 07/25/2019).

[5] *AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning*. URL: https://aws.amazon.com/cloudformation/ (visited on 02/04/2019).

[6] *AWS SAM CLI*. URL: https://github.com/awslabs/aws-sam-cli (visited on 02/18/2019).

[7] *Black Box Testing - Software Testing Fundamentals*. URL: http://softwaretestingfundamentals.com/black-box-testing/ (visited on 02/07/2019).

[8] J. N. Buxton and B Randell. *SOFTWARE ENGINEERING TECHNIQUES*. Tech. rep. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[9] *Chef - Automate IT Infrastructure | Chef*. URL: https://www.chef.io/chef/ (visited on 01/30/2019).

[10] *ChefSpec — Chef Docs*. URL: https://docs.chef.io/chefspec.html (visited on 02/18/2019).

[11] *CI/CD for microservices | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd (visited on 02/18/2019).

[12] *Cloud Deployment Manager - Simplified Cloud Management | Cloud Deployment Manager | Google Cloud*. URL: https://cloud.google.com/deployment-manager/ (visited on 02/04/2019).

[13] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Signature Series (Cohn). Pearson Education, 2009. ISBN: 9780321660565. URL: https://books.google.pt/books?id=8IglA6i_JwAC.

[14] *ComponentTest*. URL: https://martinfowler.com/bliki/ComponentTest.html (visited on 02/08/2019).

[15] *Configure your OutSystems environment - OutSystems*. URL: https://success.outsystems.com/Documentation/11/Setting_Up_OutSystems/Configure_your_OutSystems_environment (visited on 09/03/2019).

[16] *Consumer-Driven Contracts: A Service Evolution Pattern*. URL: https://martinfowler.com/articles/consumerDrivenContracts.html (visited on 02/06/2019).

[17] *Continuous Integration and Delivery - CircleCI*. URL: https://circleci.com/ (visited on 02/18/2019).

[18] *Continuous integration, explained | Atlassian*. URL: https://www.atlassian.com/continuous-delivery/continuous-integration (visited on 02/18/2019).

[19] *Continuous integration vs. continuous delivery vs. continuous deployment*. URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (visited on 02/18/2019).

[20] *Cookstyle — Chef Docs*. URL: https://docs.chef.io/cookstyle.html (visited on 07/19/2019).

[21] *Cucumber*. URL: https://cucumber.io/ (visited on 02/18/2019).

[22] *Deploy != Release – Turbine Labs*. URL: https://blog.turbinelabs.io/deploy-not-equal-release-part-one-4724bc1e726b (visited on 02/14/2019).

[23] *End-to-end vs. contract-based testing: How to choose | TechBeacon*. URL: https://techbeacon.com/app-dev-testing/end-end-vs-contract-based-testing-how-choose (visited on 01/29/2019).

[24] *Foodcritic - A lint tool for your Chef cookbooks*. URL: http://www.foodcritic.io/ (visited on 07/19/2019).

[25] *GitHub*. URL: https://github.com/ (visited on 07/25/2019).

[26] *Google Testing Blog: Just Say No to More End-to-End Tests*. URL: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (visited on 02/12/2019).

[27] *H2 Database Engine*. URL: https://www.h2database.com/html/main.html (visited on 07/25/2019).

[28] *How Contract Tests Improve the Quality of Your Distributed Systems*. URL: https://www.infoq.com/articles/contract-testing-spring-cloud-contract (visited on 02/05/2019).

[29] *How Have Microservices Orchestrated a Difference in System Architecture? - DZone Microservices*. URL: https://dzone.com/articles/how-have-microservices-orchestrated-difference-in-1 (visited on 09/10/2019).

[30] *How Pact works - Pact*. URL: https://docs.pact.io/how_pact_works (visited on 02/06/2019).

[31] *How to move from CI to CD with Jenkins Workflow - JAXenter*. URL: https://jaxenter.com/how-to-move-from-ci-to-cd-with-jenkins-workflow-128135.html (visited on 02/18/2019).

[32] *InSpec*. URL: https://www.inspec.io/docs/ (visited on 02/18/2019).

[33] *Introduction - Pact*. URL: https://docs.pact.io/ (visited on 02/06/2019).

[34] *Is it Possible to Test Programmable Infrastructure? Matt Long at QCon London Made the Case for Yes*. URL: https://www.infoq.com/news/2017/03/testing-infrastructure (visited on 02/18/2019).

[35] *Jamie Tanna / jar-deploy-cookbook · GitLab*. URL: https://gitlab.com/jamietanna/jar-deploy-cookbook (visited on 08/01/2019).

[36] *Java | Oracle*. URL: https://www.java.com/en/ (visited on 08/06/2019).

[37] *Jenkins*. URL: https://jenkins.io/ (visited on 02/18/2019).

[38] *Jenkins Plugins*. URL: https://plugins.jenkins.io/ (visited on 07/19/2019).

[39] *JUnit 5*. URL: https://junit.org/junit5/ (visited on 09/05/2019).

[40] *Large-Scale Continuous Delivery at Netflix and Waze Using Spinnaker*. URL: https://cloud.withgoogle.com/next18/sf/sessions/session/155951 (visited on 02/17/2019).

[41] *LocalStack*. URL: https://localstack.cloud/ (visited on 02/18/2019).

[42] *Managing the Applications Lifecycle - OutSystems*. URL: https://success.outsystems.com/Documentation/10/Managing_the_Applications_Lifecycle (visited on 01/28/2019).

[43] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2007. ISBN: 9780132797467. URL: https://books.google.pt/books?id=-izOiCEIABQC.

[44] *Microservices Architecture – An Experience – experience@imaginea*. URL: https://blog.imaginea.com/microservices-architecture-an-experience/ (visited on 01/28/2019).

[45] *Mockito framework site*. URL: https://site.mockito.org/ (visited on 08/08/2019).

[46] *Mocks Aren't Stubs*. URL: https://martinfowler.com/articles/mocksArentStubs.html (visited on 02/07/2019).

[47] *Mountebank - over the wire test doubles*. URL: http://www.mbtest.org/ (visited on 02/09/2019).

[48] *Move Fast and Consumer Driven Contract Test Things - Speaker Deck*. URL: https://speakerdeck.com/alonpeer/move-fast-and-consumer-driven-contract-test-things?slide=81 (visited on 02/17/2019).

[49] *Nebula: A collection of Gradle plugins, built by Netflix*. URL: http://nebula-plugins.github.io/ (visited on 09/05/2019).

[50] *Oracle VM VirtualBox*. URL: https://www.virtualbox.org/ (visited on 08/13/2019).

[51] *OutbyNumbers-DataSheet @ www.outsystems.com*. URL: http://www.outsystems.com/res/OutbyNumbers-DataSheet.

[52] *OutSystems Again Named a Leader in Gartner's 2018 Magic Quadrant for Enterprise High-Productivity Application Platform as a Service | OutSystems*. URL: https://www.outsystems.com/News/high-productivity-apaas-gartner-leader/ (visited on 01/24/2019).

[53] *OutSystems Cloud architecture - OutSystems*. URL: https://success.outsystems.com/Evaluation/Architecture/OutSystems_Public_Cloud_Architecture (visited on 01/28/2019).

[54] *OutSystems Enterprise aPaaS | Core Products | OutSystems*. URL: https://www.outsystems.com/enterprise-apaas/ (visited on 01/28/2019).

[55] *Pact-Ruby*. URL: https://github.com/pact-foundation/pact-ruby (visited on 02/07/2019).

[56] *Pipeline as Code*. URL: https://jenkins.io/doc/book/pipeline-as-code/ (visited on 07/19/2019).

[57] *Postman | API Development Environment*. URL: https://www.getpostman.com/ (visited on 09/05/2019).

[58] *Presentations and Videos - Spinnaker*. URL: https://www.spinnaker.io/publications/presentations/#spinnaker-continuous-delivery-from-first-principles-to-production (visited on 02/17/2019).

[59] *Programmable Infrastructure Needs Testing Too - OpenCredo*. URL: https://opencredo.com/blogs/programmable-infrastructure-needs-testing/ (visited on 01/31/2019).

[60] D. Savchenko, G. Radchenko, and O. Taipale. "Microservices validation: Mjolnirr platform case study." In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2015, pp. 235–240. ISBN: 978-9-5323-3082-3. DOI: 10.1109/MIPRO.2015.7160271. URL: http://ieeexplore.ieee.org/document/7160271/.

[61] *Selenium - Web Browser Automation*. URL: https://www.seleniumhq.org/ (visited on 02/18/2019).

[62] *Serverspec - Home*. URL: https://serverspec.org/ (visited on 02/18/2019).

[63] J. P. Sotomayor, S. C. Allala, P. Alt, J. Phillips, T. M. King, and P. J. Clarke. "Comparison of Runtime Testing Tools for Microservices." In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, July 2019, pp. 356–361. ISBN: 978-1-7281-2607-4. DOI: 10.1109/COMPSAC.2019.10232. URL: https://ieeexplore.ieee.org/document/8754337/.

[64] *Spinnaker*. URL: https://www.spinnaker.io/ (visited on 02/18/2019).

[65] *Spring Boot*. URL: https://spring.io/projects/spring-boot (visited on 08/06/2019).

[66] *Static Analysis vs Dynamic Analysis in Software Testing*. URL: https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/ (visited on 07/19/2019).

[67] *Test Strategy for Microservices | GoCD Blog*. URL: https://www.gocd.org/2018/05/08/continuous-delivery-microservices-test-strategy/ (visited on 02/07/2019).

[68] *Testing and monitoring in production - your QA is incomplete without it : Assertible*. URL: https://assertible.com/blog/testing-and-monitoring-in-production-your-qa-is-incomplete-without-it (visited on 02/14/2019).

[69] *Testing apps locally with the emulator | Cloud Pub/Sub Documentation | Google Cloud*. URL: https://cloud.google.com/pubsub/docs/emulator (visited on 02/18/2019).

[70] *Testing in Production, the safe way – Cindy Sridharan – Medium*. URL: https://medium.com/@copyconstruct/testing-in-production-the-safe-way-18ca102d0ef1 (visited on 02/13/2019).

[71] *Testing Microservices, the sane way – Cindy Sridharan – Medium*. URL: https://medium.com/@copyconstruct/testing-microservices-the-sane-way-9bb31d158c16 (visited on 02/12/2019).

[72] *Testing of Microservices*. URL: https://labs.spotify.com/2018/01/11/testing-of-microservices/ (visited on 02/04/2019).

[73] *Testing Strategies in a Microservice Architecture*. URL: https://martinfowler.com/articles/microservice-testing/ (visited on 02/07/2019).

[74] *The Test Pyramid In Practice (1/5) | OCTO Talks !* URL: https://blog.octo.com/en/the-test-pyramid-in-practice-1-5/ (visited on 07/24/2019).

[75] *UnitTest*. URL: https://martinfowler.com/bliki/UnitTest.html (visited on 02/18/2019).

[76] *Vagrant by HashiCorp*. URL: https://www.vagrantup.com/ (visited on 08/13/2019).

[77] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. "Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services." In: *OSDI*. 2016, pp. 635–651.

[78] *Welcome - The resource for Chef cookbooks - Chef Supermarket*. URL: https://supermarket.chef.io/ (visited on 08/05/2019).

[79] *Welcome to Kitchen - KitchenCI*. URL: https://kitchen.ci/ (visited on 08/13/2019).

[80] *What is cloud orchestration (cloud orchestrator)? - Definition from WhatIs.com*. URL: https://searchitoperations.techtarget.com/definition/cloud-orchestrator (visited on 01/31/2019).

[81] *What is DevOps? | Atlassian*. URL: https://www.atlassian.com/devops (visited on 02/04/2019).

[82] *What is Hoverfly? — Hoverfly v1.0.0-rc.1 documentation*. URL: https://hoverfly.readthedocs.io/en/latest/index.html (visited on 02/06/2019).

[83] *What Is Immutable Infrastructure? | DigitalOcean*. URL: https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure#advantages-of-immutable-infrastructure (visited on 07/19/2019).

[84] *What is Infrastructure as Code? - Azure DevOps | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code (visited on 01/30/2019).

[85] *What is infrastructure (IT infrastructure)? - Definition from WhatIs.com*. URL: https://searchdatacenter.techtarget.com/definition/infrastructure (visited on 01/30/2019).

[86] *What Is Software Testing - Definition, Types, Methods, Approaches*. URL: https://www.softwaretestingmaterial.com/software-testing/ (visited on 07/24/2019).

[87] *What is Software Testing? Introduction, Definition, Basics & Types*. URL: https://www.guru99.com/software-testing-introduction-importance.html (visited on 07/24/2019).

[88] *Where work happens | Slack*. URL: https://slack.com/ (visited on 07/25/2019).

[89] *Why environment provisioning is a key part of DevOps? – Clarive*. URL: https://clarive.com/why-environment-provisioning/ (visited on 01/30/2019).

[90] *Why You Should Be Testing in Production | Sauce Labs*. URL: https://saucelabs.com/blog/why-you-should-be-testing-in-production (visited on 02/07/2019).

[91] *WireMock - WireMock*. URL: http://wiremock.org/ (visited on 08/14/2019).

APPENDIX

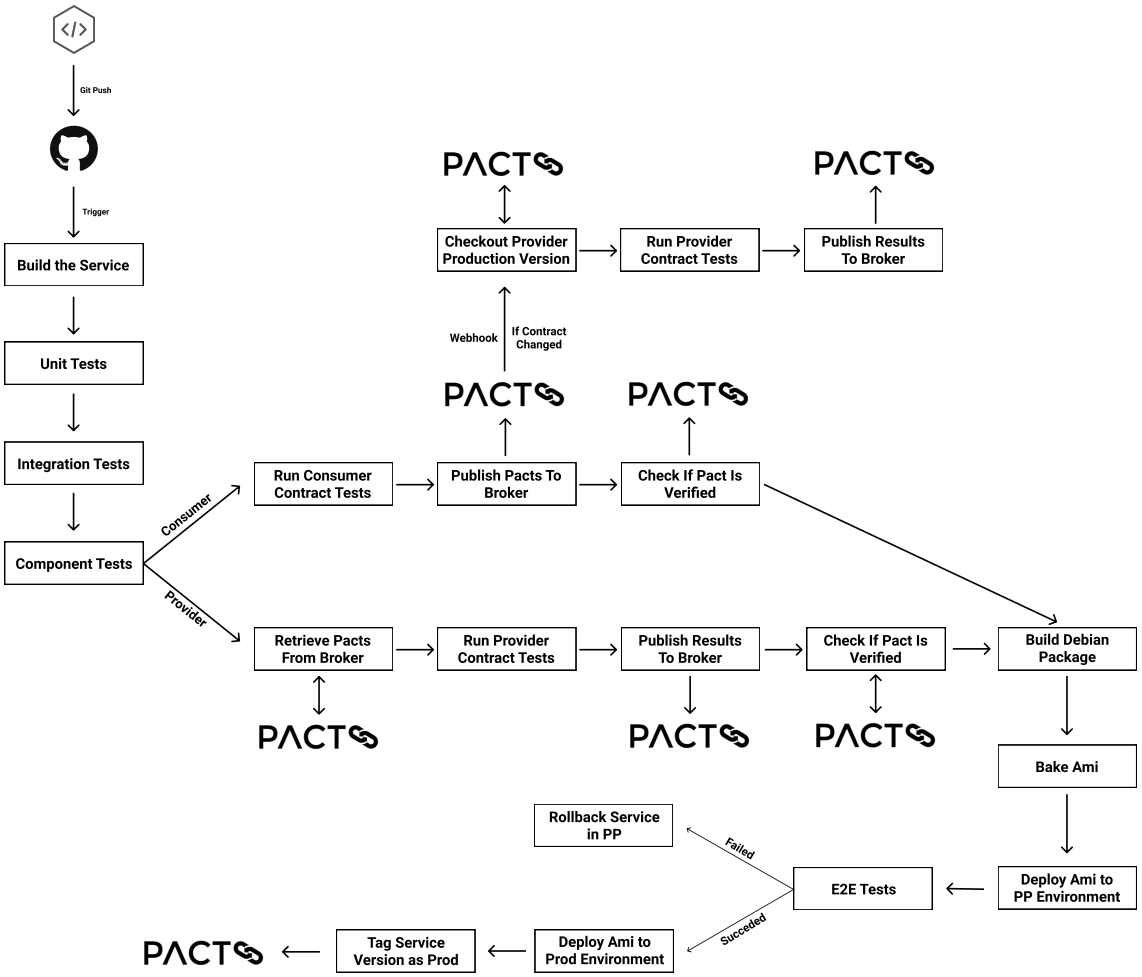


Figure A.1: Microservices detailed pipeline.