UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

PACE: Domain-Specific Language to Enable Developers Autonomy in Dealing with Complex Build Pipelines

Nelson Ricardo Matos Fonseca

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa
Co-orientador: Prof. Doutor João Paulo Fernandes

Covilhã, Junho de 2019

Acknowledgements

At the end of another important phase of my life, I do not want to miss out on the opportunity to thank all the people who helped me to reach this point.

I would like to begin by thanking Professor Simão Melo de Sousa and Professor João Paulo Fernandes for all the coordination during this dissertation, and for helping me doing this dissertation in an industrial context. All the feedback provided and all the time spent to help me through this work was essential to get to the end with a strong and cohesive dissertation.

I also want to thank OutSystems for giving students the opportunity to do a dissertation in an industrial context. More specifically, I would like to thank Mário Pires and Miguel Costa Antunes, my coordinators from OutSystems, who helped me not only during this dissertation with their ideas and feedback but also for help me grow at different levels.

Still in the context of OutSystems, I want to thank the Productivity Group for having incorporated me as an element of the team, and for all the help and support provided during this work. They were an essential source of motivation to carry this dissertation to the end. I cannot forget the other OutSystems R&D teams that have always been available to help at any time, thank you!

Now in Portuguese, so that it can be understood by them...

Tenho que agradecer também à minha família. Um obrigado aos meus pais Maria Fonseca e Nelson Fonseca, e ao meu irmão Carlos Fonseca, por me terem apoiado não só financeiramente, mas também emocionalmente tendo sempre servido de suporte e sempre me apoiado em todas as decisões. Sem este apoio fornecido por eles não teria conseguido “cometer a loucura” de ir estudar sozinho para Lisboa.

Não me posso esquecer da minha segunda família, a família do coração. Um obrigado aos “Sanaitsabes”, “Tascoscopos”, e restante família ubiana, pela amizade, pela ajuda, por estarem sempre prontos a dar uma palavra de apoio, e por nunca me deixarem sozinho mesmo estando separados por centenas de quilómetros.

Mais uma vez, um grande obrigado a todos por estarem presentes na minha vida.

Resumo

A complexidade do software faz com que os seus processos de validação também eles sejam complexos, nomeadamente os *build pipelines*. Esta complexidade dos *build pipelines*, associada com a falta de conhecimento existente nas diferentes equipas sobre a sua manipulação, faz com que as equipas não consigam ser totalmente independentes. Esta independência faz com que exista uma equipa responsável pela manutenção de *build pipelines*.

A falta de independência das equipas leva a que elas não consigam desenvolver os seus componentes de ponta a ponta, o que poderá levar a um atraso no desenvolvimento se a equipa responsável pela manutenção de *build pipelines* não conseguir satisfazer todos os pedidos em pouco tempo.

Uma vez que o mercado de *software* é um mercado competitivo, é preciso eliminar todas as fontes de atrasos, ou de possíveis atrasos, para que se consiga entregar valor aos clientes de forma rápida e frequente. Com isto, é necessário encontrar uma solução que permita que diferentes pessoas de diferentes áreas e equipas possam manipular *build pipelines*, de uma forma simples e rápida, sem possuírem praticamente nenhum contexto sobre os conceitos, termos, e configurações de *build pipelines*.

Neste trabalho é então apresentada uma *Domain Specific Language (DSL)* com uma sintaxe de simples compreensão que abstrai alguns conceitos relacionados com *build pipelines*, que permita criar lógica de *build pipelines* e programar a lógica de automação tudo na mesma linguagem, e que também permita ter reutilização de código. Esta solução é baseada com duas soluções já implementadas na indústria e que obtiveram sucesso, o uso de uma *DSL* e a existência de bibliotecas de *build pipelines*.

O desenho, implementação e validação foram feitos no contexto industrial da OutSystems. Isto permitiu validar o protótipo num cenário real, fazendo a comparação entre o uso do protótipo e a forma de desenvolvimento de *build pipelines* atual no contexto desta empresa. Os resultados obtidos mostram evidências de que no geral a produtividade aumenta com o uso da *DSL*.

Palavras-chave

Pipeline as Code, Entrega Contínua, Integração Contínua, *Build Pipeline*, Linguagem de Domínio Específico, Melhoria da Produtividade, Redução de Complexidade, Autonomia

Resumo alargado

O mundo das novas tecnologias atualmente evolui a um passo extremamente rápido. Este facto faz com que as empresas precisem de lançar o seu software o mais rápido possível para o mercado para se manterem sempre atualizadas. Esta aceleração precisa de ser feita cuidadosamente para que não se saltem passos importantes no ciclo de desenvolvimento de software.

Toda esta aceleração precisa de vir de várias áreas dentro de uma empresa, e uma das fases no ciclo do desenvolvimento de software que esta dissertação se insere é o controlo de qualidade. Nesta fase é preciso garantir que tudo corre de uma forma suave. Para que isto aconteça as empresas estão a implementar os conceitos de *Continuous Integration and Continuous Delivery (CI/CD)* e a dar autonomia às diferentes equipas para que possam desenvolver o seu componente de ponta a ponta sem ter ninguém a bloquear o seu caminho. Desta forma, cada equipa pode desenvolver ao seu ritmo podendo cada uma ter o seu processo de *release* independente.

Um desafio existente em dar autonomia às equipas é que elas devem ser capazes de lidar com todas as fases de desenvolvimento de software, desde o design até à *release* de um componente. Atualmente nem todas as indústrias são capazes de fazer isto por vários motivos, sendo um deles a complexidade na construção de *build pipelines*. Esta complexidade de construção ligada com a complexidade do software existente na indústria, e com a falta de conhecimento existente nas diferentes equipas sobre a manipulação de *build pipelines*, faz com que as equipas não consigam ser totalmente independentes. Por este motivo, é comum existir uma equipa responsável pela manutenção de *build pipelines* dentro da engenharia, que recebe vários pedidos para fazer alterações a *build pipelines*. Isto poderá resultar na introdução de atrasos nas equipas desnecessariamente.

O objetivo desta dissertação é então encontrar uma solução que permite que diferentes pessoas de diferentes áreas e equipas possam manipular *build pipelines*, de uma forma simples e rápida, sem possuírem praticamente nenhum contexto sobre os conceitos, termos, e configurações de *build pipelines*.

Para encontrar uma solução que melhor se adequa, foi feito um levantamento do estado da arte onde foi observado que algumas soluções dentro desta área já tinham sido implementadas. Estas soluções apresentaram os seus pontos fortes e fracos, mas nenhuma delas resolvia totalmente o problema no qual se está a combater. Da análise destes pontos surgiu a ideia de juntar duas soluções já implementadas na indústria: o uso de uma *DSL*; e a existência de bibliotecas de *build pipelines*.

Nesta dissertação é proposta então uma *DSL* que permite a reutilização de código através do conceito de herança existente nas linguagens orientadas a objetos, e que permite através de uma sintaxe simples criar lógica de *build pipelines* e programar a linguagem de automação tudo no mesmo script. Assim os desenvolvedores conseguem criar ou manusear os *build pipelines* facilmente, libertando-os de toda a complexidade inerente à configuração. O objetivo com esta proposta é reduzir o número de configurações e conceitos que o desenvolvedor precisa de saber antes de começar a manipular *build pipelines*. Para além disto, esta proposta foca-se em baixar a curva de aprendizagem para que o tempo e dificuldade na aprendizagem sejam baixos.

Esta solução foi desenvolvida no contexto industrial da OutSystems, no entanto apesar de o protótipo estar pensado para este contexto, pode facilmente se adaptar a um contexto industrial mais genérico.

A solução depois de implementada, foi validada dentro do mesmo contexto. O objetivo com esta validação experimental foi mostrar evidências que com a solução proposta consegue-se fornecer aos desenvolvedores uma *DSL* onde consigam de uma forma simples e rápida manipular *build pipelines* complexos. Para atingir este objetivo foram feitas validações com dois grupos que desenvolveram duas tarefas. A diferença entre estes dois grupos é que um executou as tarefas com o protótipo da *DSL* desenvolvido no contexto desta dissertação, e o outro desenvolveu recorrendo ao *pipeline as code* JSON oferecido pelo GoCD. As métricas recolhidas durante esta validação permitiram fazer uma análise formal e informal. Na análise formal, foi feita uma análise ao questionário *System Usability Scale (SUS)* e *Net Promoter Score (NPS)*, uma análise aos tempos de desenvolvimento, e o cálculo do teste estatístico Mann-Whitney U que permite ter evidências estatísticas de que existe, ou não, diferença entre os dois grupos. Na análise informal foram retiradas conclusões através do comportamento e dificuldades sentidas pelos utilizadores testados.

O resultado da validação experimental permitiu mostrar evidências de que a produtividade no desenvolvimento de *build pipelines* aumenta, a possibilidade de erros serem cometidos é menor com a *DSL*, e que é menos custoso aprender a utilizar a *DSL* do que a corrente forma de desenvolvimento na OutSystems.

Abstract

The complexity of the product developed makes its validation processes too complex, namely build pipelines. This complexity of build pipelines, coupled with the lack of knowledge in different teams about their manipulation, means that teams cannot be fully independent. This independence makes one team responsible for maintaining build pipelines.

The lack of independence on the teams means that they can not develop their components from end to end, which can lead to a delay in development if the team responsible for maintaining pipelines cannot fulfill all requests in a short time.

Since the software market is a competitive market, it is necessary to eliminate all sources of delays, or possible delays, in order to deliver value to customers quickly and frequently. With this, it is necessary to find a solution that allows different people from different areas and teams to handle build pipelines, in a simple and fast way, with practically no context about the concepts, terms, and configurations of build pipelines.

In this work, a DSL is presented with a simple understanding syntax that abstracts some concepts related to build pipelines, which allows to create build pipelines logic and to program the automation logic in the same language, and it also allows having code reuse. This solution is based on two solutions already implemented in the industry and that has been successful: the use of a DSL; and the existence of libraries of build pipelines.

The design, implementation, and validation were done in the industrial context of OutSystems. This allowed the validation of the prototype in a real scenario, making a comparison between the use of the prototype and the form of development of current build pipelines in the context of this company. The results obtained show evidence that in overall, productivity increases with the use of DSL.

Keywords

Pipeline as Code, Continuous Delivery, Continuous Integration, Build Pipelines, Domain-specific language, Improving Productivity, Reducing Complexity, Autonomy

Contents

1 Introduction **1**
1.1 Context and Motivation . . . 3
1.2 Problem Statement and Goals . . . 4
1.3 Research Methodology . . . 7
1.4 Proposed Solution . . . 8
1.5 Main Contributions . . . 9
1.6 Document Structure Overview . . . 9

2 Industrial Context **11**
2.1 Introduction . . . 11
2.2 The Company and its main Product . . . 11
2.3 The Development Team, Methodology and Paradigms . . . 13
2.4 Platform Development . . . 14
2.5 Quality Assurance . . . 16
2.6 Conclusion . . . 18

3 State of the art **21**
3.1 Introduction . . . 21
3.2 CI/CD Practices in the Industries . . . 21
3.3 Autonomy and Elimination of Team's Dependencies . . . 23
3.3.1 Use of existing CI/CD tools . . . 24
3.3.2 Creation of Pipelines Libraries . . . 28
3.3.3 Creation and Use of DSL . . . 29
3.4 Conclusion . . . 31

4 Proposed Solution **33**
4.1 Introduction . . . 33
4.2 Modus Operandi . . . 33
4.3 Decisions . . . 35
4.3.1 DSL . . . 36
4.3.2 Internal DSL . . . 37
4.3.3 External DSL . . . 37
4.3.4 Configurations . . . 38
4.4 Conclusion . . . 42

5 Solution Implementation **45**
5.1 Introduction . . . 45
5.2 Toolset . . . 45
5.3 Implementation Decisions . . . 47
5.4 Final DSL Structure . . . 53
5.5 Implementation Details . . . 61
5.5.1 Definition of grammar rules . . . 61
5.5.2 Code Generator Programming . . . 62
5.5.3 Creating Custom Validation Rules . . . 64

5.6 Comparison of build pipelines made with DSL and JSON 64
5.7 Conclusion 67

6 Experimental Validation 69
6.1 Introduction 69
6.2 Plan 70
6.2.1 Task 1 73
6.2.2 Task 2 73
6.3 Results 74
6.4 Discussion of the Evaluated Results 77
6.4.1 Formal Analysis 77
6.4.2 Informal Analysis 81
6.5 Conclusion 82

7 Conclusion 85
7.1 Conclusion and Final Results 85
7.2 Future Work 87

References 89

A Questionnaire to calculate SUS 95

B Critical Values of the Mann-Whitney U 97

C Questionnaire Results 99

D PACE Script 105

E JSON Script 107

F PACE Grammar 109

List of Figures

2.1 OutSystems product [Out18b]. 12
2.2 The first build pipeline created to validate the OutSystems product. 17
2.3 The current build pipeline that validates the OutSystems product[1]. 17

3.1 Pipeline used to test different CI/CD tools. 26

4.1 Pipeline system in GoCD. 38
4.2 Pipeline in PACE. 39

5.1 Grammar example. 46
5.2 Example of a function written in Xtend. 46
5.3 Resource assignment to Jobs. 48
5.4 PACE telemetry. 50
5.5 Example of run. 50
5.6 First PACE design. 52
5.7 Current PACE design. 52
5.8 Current PACE design. 53
5.9 PACE import. 53
5.10 PACE include. 54
5.11 PACE telemetry. 54
5.12 PACE pipeline definition. 54
5.13 PACE repositories definition with manual trigger. 55
5.14 PACE repositories definition with schedule trigger. 55
5.15 PACE repositories definition with automatic trigger, and with a repository not being triggered. 55
5.16 PACE Stage definition. 55
5.17 PACE Stage dependencies definition. 56
5.18 PACE discardable agents definition. 57
5.19 PACE Job definition. 57
5.20 PACE Job timeout definition. 57
5.21 PACE fetch artifacts definition. 58
5.22 PACE Run definition. 58
5.23 PACE variable definition. 58
5.24 PACE save artifacts definition. 59
5.25 PACE function of Stage type definition. 59
5.26 PACE function of Job type definition. 59
5.27 PACE function of Python type definition. 60
5.28 PACE Stage using override feature. 60
5.29 PACE Stage using extends feature. 60
5.30 PACE Job using override feature. 61
5.31 PACE Job using extends feature. 61
5.32 Example of PACE grammar. 61
5.33 Example of PACE code generator. 62
5.34 Example of PACE validation rule. 64

5.35 Build pipeline implemented. 64
5.36 Build pipeline, represented in figure 5.35 implemented in PACE. 65

6.1 Curve for L =31%. 71
6.2 Pipeline to be created in both tasks. 72
6.3 Task 1 duration times. 79

A.1 Questionnaire to calculate SUS. 95

B.1 Critical Values of the Mann-Whitney U (Two-Tailed Testing) [Lau09]. 97

C.1 Responses from control group to the affirmation “I think that I would like to use this system frequently.”. 99
C.2 Responses from experimental group to the affirmation “I think that I would like to use this system frequently.”. 99
C.3 Responses from control group to the affirmation “I found the system unnecessarily complex. (The system should be simpler)”. 99
C.4 Responses from experimental group to the affirmation “I found the system unnecessarily complex. (The system should be simpler)”. 100
C.5 Responses from control group to the affirmation “I thought the system was easy to use.”. 100
C.6 Responses from experimental group to the affirmation “I thought the system was easy to use.”. 100
C.7 Responses from control group to the affirmation “I think that I would need the support of a technical person to be able to use this system.”. 100
C.8 Responses from experimental group to the affirmation “I think that I would need the support of a technical person to be able to use this system.”. 101
C.9 Responses from control group to the affirmation “I found the various functions in this system were well integrated.”. 101
C.10 Responses from experimental group to the affirmation “I found the various functions in this system were well integrated.”. 101
C.11 Responses from control group to the affirmation “I thought there was too much inconsistency in this system.”. 101
C.12 Responses from experimental group to the affirmation “I thought there was too much inconsistency in this system.”. 102
C.13 Responses from control group to the affirmation “I would imagine that most people would learn to use this system very quickly.”. 102
C.14 Responses from experimental group to the affirmation “I would imagine that most people would learn to use this system very quickly.”. 102
C.15 Responses from control group to the affirmation “I found the system very cumbersome to use. (cumbersome - slow or complicated and therefore inefficient)”. 102
C.16 Responses from experimental group to the affirmation “I found the system very cumbersome to use. (cumbersome - slow or complicated and therefore inefficient)”.103
C.17 Responses from control group to the affirmation “I felt very confident using the system.”. 103
C.18 Responses from experimental group to the affirmation “I felt very confident using the system.”. 103

C.19 Responses from control group to the affirmation “I needed to learn a lot of things before I could get going with this system.”. 103
C.20 Responses from experimental group to the affirmation “I needed to learn a lot of things before I could get going with this system.”. 104

List of Tables

3.1 Characteristics of each CI/CD tool important for the context of OutSystems.[1] . . 27

4.1 PACE to GoCD Concepts. 39

6.1 JSON questionnaire results. 74
6.2 PACE questionnaire results. 74
6.3 Table adapted from the mapping of the SUS result in percentiles to adjectives [Had18]. 75
6.4 Response grouping table. 75
6.5 Table adapted from [Sar19]. 76
6.6 Results of task 1. 76
6.7 Results of task 2. 77

Acronyms

API Application Programming Interface

CD Continuous Delivery

CI Continuous Integration

CI/CD Continuous Integration and Continuous Delivery

DE Development Environment

DSL Domain Specific Language

GPL General Purpose Language

IDE Integrated Development Environment

IS Integration Studio

LOC Lines of Code

LPS Language Server Protocol

NPS Net Promoter Score

PaaS Platform as a Service

PR Pull Request

PS Platform Server

SS Service Studio

SUS System Usability Scale

SVN Subversion

UBI University of Beira Interior

URL Uniform Resource Locator

VSM Value Stream Map

Chapter 1

Introduction

The software market is constantly changing. If a software development company does not act fast it will hardly get good results in the market. In order for this to be attainable, a number of agile development techniques emerge, where automation reigns, allowing companies to be one step ahead of the competition.

In addition to the market's needs for a given product, what leads customers to invest in it always ends up being the quality of it. A product with poor quality means that customers are not satisfied with it, making difficult for them to stay ahead of their competitors.

In order to guarantee the quality of a product, it goes through several validation phases, where several tests are performed. These tests are essential to discover errors and defects implemented during software development, so it is important to know the results of these tests quickly. One way to get the results of the tests fast is to make them automatic so that a developer is not wasting time doing manual and repetitive work, which in addition to being highly time and effort consuming tasks, can easily lead to errors.

The CI/CD practices, presented in more detail in chapter 2, allow "deliver software to a production environment with speed, safety, and reliability" [Chr19]. This allows the developed product to be tested as quickly as possible, so that it can fail as fast as possible, leading to less time between the time that the error is committed until it is corrected [SDG+16]. All changes run through build pipelines, explained in more detail in chapter 2, allowing the product to always be validated in the same way automatically.

The various validation phases of a build pipeline that software normally has to pass to ensure its quality is complex [Jen17a]. One way of removing this complexity from developers is to automate and optimize all steps through build pipelines. The use of build pipelines is one of the key points that there is a lesser effort on the part of the developer to develop software, obtaining quick feedback of the validation carried out by the build pipeline. If one of the validations fails, the developer is informed of this and must correct the software, which will then go through the same validation processes again. Validations done in this way allow tasks to be performed in a repetitive, automatic, and reliable manner, allowing a software delivery with a lower risk of bugs [Jen17a, Jen18].

In this way, with CI/CD practices, we can have advantages that allow us to validate the product in a fast way. However, turning validation processes into a build pipeline to get the most out of it, brings added difficulty to developers who have to learn and master new concepts, technologies and tools in order to create these build pipelines. Sometimes this is one of the bottlenecks in companies that prevent software development from happening more quickly because the implementation of build pipelines can be poorly done, which means that is not taking the advantages of what the Continuous Delivery (CD) practice offers. In chapter 3 the case of

Amazon, Pivotal, and many others, who have built tools to handle more easily the creation and maintenance of build pipelines, are presented. There is also the factor of developers having to learn and master new concepts, technologies and tools besides those needed to develop their product.

Software is continuously changing over time, and validation phases keep being added, removed, or changed. All this means that changes need to be made to build pipelines in order to match the new validation process. This constant development applied to a complex product leads to complex build pipelines difficult to maintain. Several companies have been dealing with this problem by having special teams enabling the practice of CI/CD by providing other teams with CI/CD platforms that they can customize. However, in some situations due to the complexity of the product, it can lead to developers teams depending on platform teams to be able to evolve and modify the build pipelines. This creates extra handoffs that compromise the autonomy of teams and even their delivery flow.

The objective of this dissertation is to simplify and increase the productivity of the developer teams in creating and maintaining their build pipelines independently of their complexity without requiring a high level of expertise on the technologies and tooling used to develop the build pipelines. In this way, abilities will be optimized to make changes to a complex build pipeline.

Of the various solutions studied in the state of the art, chapter 3, the solution of building a DSL stands out from the rest since it is a solution that other companies have already implemented and that have succeeded. These DSLs are mostly implemented in YAML and therefore are declarative DSLs, being close to a configuration file. However, in order to achieve our goals, unlike the solutions studied in the state of the art, the DSL proposed in this dissertation allows to reuse code, have logic that makes it possible to have much more control in the manipulation of build pipelines, and have the ability to have the task automation language incorporated into the DSL itself. This difference is what makes the implemented solution stand out from the other pipeline as code solutions that are currently implemented in the industry.

With the creation of this DSL, we will be able to abstract the developer from practically all the complexity related to build pipelines, thus eliminating handoffs, making the different teams have the necessary context so they can manipulate their own validation processes. This dissertation, therefore, presents a solution to increase efficiency of the different teams allowing them to more quickly evolve their component, without needing the support of other teams.

The prototype of this DSL was constructed in an industrial context addressing the real needs of several development teams and their highly developed pipelines system used to validate the actual commercial and complex product. This prototype was then validated in the same context by collecting a set of performance metrics taken from developers who have no context in manipulating build pipelines. These experimental validations are described in more detail in chapter 6.

The problem statement and goals, the research and work methodology, the proposed solution, and a summary of the main contributions and structure of the document are defined in this chapter.

1.1 Context and Motivation

The work was developed in an industrial context of a software development company, OutSystems[1], in the context of a partnership with University of Beira Interior (UBI).

OutSystems is a Portuguese company founded in 2001, which is currently the market leader in Low-Code Platforms for Application Development [Out19, Out16] and has been revolutionizing the way the web and mobile applications are developed. OutSystems sells a software development platform that aims to abstract the complexity and technologies behind the creation of software. The OutSystems platform drastically increases the speed of software development while highly reducing the required developer's level of expertise and knowledge about the underlying technologies.

The OutSystems product is a very complex piece of software, with a large and complex validation process running tens of thousands of automated tests. This process of validation has been evolving over the years made by one team that has a high level of knowledge about build pipelines and the CI/CD tool used.

More information about OutSystems, its product, and how developers work can be found at chapter 2.

With what has been described, the context of OutSystems becomes an interesting case study, where we want to enable the autonomy of the different development teams to develop, maintain and manage their complex build pipelines while enabling the OutSytems's engineering to scale and accelerate in order to satisfy the business needs.

However, OutSystems is not the only company facing these challenges, several other companies in the industry have faced similar problems.

In order for companies to accelerate, they need to reduce the time from ideation to feature delivery. Most companies are releasing product changes much more frequently in order to gather user feedback and ensure they are delivering the right features that provide the intended value to their customers. To be able to do that they need to increase development teams feature delivery flow, which means eliminating any impediments including dependencies on other teams as much as possible.

The need to eliminate dependencies is to increase development team's flow and allowing teams to have full autonomy and control on the entire software development lifecycle, from ideation to release, of the components they own. If a team does not need to depend on external teams to develop their components they can avoid wait times and hence increase their flow. Handoffs between teams are one of the main causes of delays in software development and a major blocker to acceleration. So that no team delays the release process due to the blockage that other teams may cause, the need arises for the development teams to become autonomous. This way they can pick up on the component they are responsible for and develop from start to finish, without any bottleneck delaying their development.

[1] https://www.outsystems.com/

To improve the autonomy of software development teams, several companies have developed their own solutions throughout the years and some of them even became commercial products. The different solutions studied and presented in this dissertation (chapter 3) although they are different in each case, all have a common topic, the pipeline as code. Pipeline as code is defined as a way to create build pipelines through a programming language, allowing developers to easily have similar pipelines without having to do the configuration in the CI/CD tool [Bad17]. In addition, the pipeline as code also allows having the entire pipeline versioned in a source repository [Jen18]. The pipeline as code is a generic approach of tool vendors to tackle the problem, however, the pipeline as code that the companies present is declarative, which brings limitations in the definition of build pipelines. If we join the concept of pipeline as code to a language that allows the creation of a build pipeline logic, together with a task automation language, and together with code reuse, we have already been able to increase the abstraction that will help developers to manipulate build pipelines.

Since this dissertation is in the industrial context of OutSystems, the motivation for this work is closely linked to the reality of this company. The CI/CD practices, were implemented to improve efficiency, an caused a special team to be created, the Engineering Productivity Group, dedicated to creating and maintaining the R&D CI/CD platform. This team manages a complex system of build pipelines that is able to validate all platform components and properly manages their dependencies while ensuring that developers have the fastest feedback loop possible. This type of team does not exist only in OutSystems. The industry pattern is to have a team (or teams) enabling the development teams to manage and create their build pipelines in a self-service manner. Unfortunately, this is hard to achieve and sometimes these teams become service providers instead of enablers [Hel19, Mat, Cor16]. This is considered an anti-pattern that can hurt the developer teams' flow, and that is the exact situation that OutSystems is trying to avoid. A self-service philosophy promotes autonomy, which means that the teams responsible for build pipelines do not become a bottleneck. Once a product is constantly growing it is necessary to have its validation processes follow this growth without long delays. The goal is then to create a solution so that the different teams in a company, can change from a philosophy service provider to self-service so they can be autonomous in the changes they have to make.

The work brings innovation to the way developers work not only in OutSystems R&D but also in the industry since the solution found allows moving away from the concept of pipeline as code as declarative language. The union of a language of simple comprehension that abstracts the underlying CI/CD tool, with a language of task automation, with a language that allows the creation of build pipelines logic, and with reuse of code, maintaining all the expressiveness, is a new way that was not being explored extensively. This will allow teams to have full autonomy to change complex build pipelines without the need to understand all the concepts, terms, and techniques behind build pipelines.

1.2 Problem Statement and Goals

With the context that has been discussed above, we can see that OutSystems, due to the complexity of its product, presents difficulties with validation processes. We have also seen that the industry, not just OutSystems, has validation processes that can be difficult to understand

if there is no background on build pipelines, leading to silos of knowledge between different teams. Knowledge silos are characterized as a lack of communication between different teams, or between different elements of a team, that causes knowledge about a certain area not to be spread by the company [Bri16]. This problem, already identified in the industry, means that teams are not autonomous in the development of their components, which can lead to delays in their release processes. This bottleneck, and others that may exist, cause the industry to fail to achieve the goal of delivering value to the customer ensuring the highest possible quality on a frequent basis.

The problem that this dissertation proposes to solve can be divided into three topics that are correlated between them:

- Characteristics and complexity of the product;
- Lack of team autonomy;
- Market needs.

Starting with the characteristics and complexity of the product, a problem that affects the development acceleration of several companies. In highly complex software systems it is often difficult to validate its various components and subsystems independently. This difficulty leads to the validation processes accompanying the complexity of the product, becoming something equally complex. In these realities, it is necessary that all the teams are able to effectively evolve the build pipelines systems as they evolve their product, which means that all teams require a similar and high-level of knowledge and skills to evolve their shared build pipelines.

However, not all developers and teams have this high-level knowledge, and this is a factor that prevents different teams from making small changes to the build pipeline. This frequently leads to a dependency of special teams to manage and maintain the build pipelines systems that can lead to the existence of a bottleneck. This bottleneck prevents development teams from presenting full autonomy in the construction or alteration of validation processes, delaying the development of their product.

Teams without autonomy cannot accelerate their development because they may be waiting by pending work from another team. This problem could be solved if the teams invested part of their time to learn the concepts related to build pipelines, and to use tools that allow the construction of build pipelines. Although it is possible to do, developers would have to learn to use one more tool with a high degree of complexity, in addition to that they already have to know to do their work, unnecessarily introducing an overload in learning.

The need to give teams autonomy in order to accelerate the development of the product is due to the fact that there is currently a high need for the product by the market, thus leading to the third topic that makes this definition of the problem. The evergrowing market needs require a constant acceleration in product development. This acceleration will allow the company to deliver value in an incremental way taking benefits, such as, knowing exactly the path to follow when it comes to developing their product. The acceleration in product development requires that all impediments that delay its development are eliminated.

Knowing that: The great complexity in the industries products, together with the lack of knowledge about build pipelines in different development teams, makes them unable to change build pipelines and build their own validation process, delaying the development of the product.

The problem is described as: Find a solution that allows different developers from different areas and teams to handle build pipelines, simply and quickly, with little or no context on build pipelines concepts, techniques and toolset.

The main objective of this dissertation is to solve the lack of autonomy and poor productivity of different teams on the manipulation of complex build pipelines. The solution found to address this problem should fulfill the following requirements:

- Ability to describe different build pipelines used by each R&D team, using a programming language that allows them to program their build pipelines as they do for the product they develop;
- Support "code reuse" so that developers can reuse each others pipeline constructs. In this way, duplication of code can be avoided, and developers don't have to redo something already done by other developers;
- Support complex build pipeline definitions by composing modules defined in multiple source code files, according to each team's needs. The implemented solution should not cause restrictions on the implementation of build pipelines, and it should be possible to define simple and complex build pipelines;
- Have built-in support for the most common pipelines jobs and constructs. In this way we can abstract concepts and tasks that are common in build pipelines from R&D, making it easier to use the solution;
- Support for task parallelization. Parallelization is a key technique to shorten the feedback loop in complex systems that need to run thousands or tens of thousands of tests. Being able to have this ability without having to know about all the complex details is a key requirement for companies with complex systems and large test population (as is the case of OutSystems);
- Be very simple to use. Developers should be able to use it with the minimum knowledge about build pipeline concepts and without the need to know anything about the underlying CD system being used, or details about the jobs implementation. This way we can reduce the learning curve that the developer will have to pass;
- Abstract concepts related to build pipelines so that developers can work with the implemented solution without having to go through a long process of learning;
- Built-in mechanism to manage (create and destroy) the agents needed to execute the jobs. So developers do not need knowledge about the infrastructure required to execute the build pipeline jobs.
- Should be extensible to ensure it can evolve as the product evolve and not become a blocker for future pipeline evolutions due to some missing concept that might only be relevant in the future;

- Should be able to create build pipelines for different CI/CD tools. In this way, the solution to be implemented is not strictly coupled to a single CI/CD system, as the need to have a build pipeline implemented in different CI/CD systems may arise;
- Should allow versioning of build pipelines changes, using source control version systems, so that developers can have control over the changes made in build pipelines.

The solution that encompasses these topics can offer teams the desired autonomy so that they can make their changes without any complexity involved, allowing their validations to be made in a timely manner and also allowing faster development of the product. Due to the business context in which this dissertation is inserted the solution will only be compatible with GoCD, however, it should be simple to make a global solution for the different CI/CD tools available. GoCD is an open-source tool that allows build pipelines to be created visually, or by code [GoCb].

A developed prototype that meets the stated objectives will be evaluated in the real context of OutSystems R&D, in order to see if the solution proposed in chapter 4 positively impacts the development of build pipelines when compared to the current form.

1.3 Research Methodology

In order to o reach the objectives presented previously, an approached was followed that divided the work into several distinct tasks. These tasks began with the study of the problem and the state of the art in the area of the problem, moving to a solution proposal followed by its implementation and evaluation.

Since this work was done in the OutSystems context, all the work was done following the same development model used by the R&D engineers, that is, agile methodologies. All the work was done in sprints of two weeks where at the end of each sprint it was presented the result of the work elaborated during the sprint, and what was planned to do for the next one.

The research methodology used throughout this dissertation can be divided into the following tasks:

- Understand the industrial context, analyzing the build pipelines to be used, the challenges, shortcomings, and existing requirements, in order to understand the problem domain;
- Formulation of the problem to be tackled in this dissertation;
- Study of the state of the art of pipeline as code, and DSLs, in order to find different perspectives on the problem to be solved;
- Proposal for a solution that tackles the specified problem in a general way, but that meets the requirements of the industrial context in which this dissertation was inserted;
- Implementation of the proposed solution within the context of OutSystems;
- Experimental validation of the implemented solution, in order to validate that the objectives were fulfilled;

- Analysis of the implemented solution, and the results obtained during the experimental evaluation in order to obtain future work that needs to be done to make the implemented solution stronger.

1.4 Proposed Solution

This section gives a brief overview of the proposed solution to the problem. In chapter 4 a description of this solution can be found in a more detailed description.

The complexity of the OutSystems product led to its components being difficult to validate individually, leading to the creation of build pipelines that over time have become very complex. Given this complexity, there is a difficulty for different teams to follow and understand all the terms, concepts, and techniques used to create a build pipeline that validates parts of the OutSystems product.

This context led to the need to have a solution where the different teams can handle the build pipelines autonomously, and without knowing all the underlying technologies.

We studied several solutions implemented by other companies that focused on the autonomy of different teams in the manipulation of build pipelines. Of these solutions, we highlight the use of DSLs that allow, through configurations, to specify build pipelines and the use of libraries of build pipelines where the developers were free to choose the one that they want to use. These solutions cannot solve by themselves the presented problem, so it was decided to combine both solutions, so that the developers achieve through a simple syntax that allows a great expressiveness to create or to handle build pipelines freeing them from all complexity inherent to the configuration of build pipelines.

The DSL design was done with the goal of allowing to create simple and complex build pipelines, with a correct architecture while reducing the likelihood of developer's errors.

To achieve the goals, it was necessary to have in the DSL different concepts of the ones existing in GoCD and allow the developers to create a build pipeline with the correct architecture. It was also included in the DSL design, the concept of code reuse and of set of build pipelines configurations. This set of configurations can be divided into three points:

- Configurations that the user can define through the DSL syntax, and that are mapped directly to the CI/CD tool;
- Configurations that the DSL abstracts through an expression, which at compile time generates much of the code needed to be valid in the CI/CD tool;
- Configurations that are pre-defined that developers cannot change. These configurations belong to GoCD, and this does not prevent using the same syntax to generate code for other CI/CD tools since the configurations are assigned by the compiler engine and not at the syntax level.

The design allowed to increase the abstraction of concepts, and reduce the number of configurations, and lines of code that a developer needs to write in order to create new build pipeline.

In this way, it is possible to reduce the complexity associated with the manipulation of build pipelines by the developers who do not have the necessary context.

With this proposed solution, we were able to offer the teams autonomy to make their changes with practically no complexity involved, allowing their validations to be made in a timely manner, accelerating product development.

1.5 Main Contributions

The work described throughout this dissertation can make a relevant contribution to the industry. Its main contribution is to propose a new way of using the pipeline as code, which consists of creating a DSL that abstracts the concepts of the underlying CI/CD tool, and that allows to create logic of build pipelines, and the logic of the automation tasks. All of this in the same code base with a simple syntax that allows code reuse. This DSL has the name PACE (Pipeline As CodE), described in more detail in chapter 4.

To achieve this main contribution, research was done on the possibility of different teams being able to manipulate their own build pipelines, thus eliminating dependencies between teams, which has concluded that different companies use different solutions. However, the different solutions of different companies can be divided into the following three topics, and it was in the last two that the solution was based:

- The use of existing CI/CD tools which allow or not pipeline as code, without any layer of abstraction;
- Creating libraries of build pipelines, where developers through configuration files can have the desired pipelines;
- Creation or use of DSLs that rely on the existing CI/CD tools, removing complexity in the creation of build pipelines.

The solution was based on the creation of a new DSL based on GoCD, a CI/CD tool, which removes complexity related to build pipelines and their configurations, and allows code reuse. This code reuse allows parts of other build pipelines to be reused and can create pipelines libraries, thereby eliminating some of the complexity.

1.6 Document Structure Overview

This document is organized into 7 chapters that present the work that was developed in the context of this dissertation. Each of these 7 chapters can be summarized as follows:

- Chapter 1 - Introduction: Responsible for introducing the context of the problem that gives rise to this dissertation and the motivations to solve it. It is also presented in more detail the problem that is intended to solve and the objectives of this work. Following is a brief overview of the proposed solution and the main contributions made with the result of this dissertation. Finally, an overview is given on the structure of this document.

- Chapter 2 - Industrial Context: The industrial context in which the work developed in the context of this dissertation is inserted, is presented. We can also find here all the details necessary to better understand the problem in question and the need for this problem to be solved.
- Chapter 3 - State of the Art: State of the art is presented in the areas of CI/CD, different CI/CD tools, and solutions implemented by other companies for an autonomous manipulation of build pipelines. From this collected materials, an analysis is done regarding the different advantages and disadvantages existing in the different approaches found in the research.
- Chapter 4 - Proposed Solution: Contains the proposed solution to the problem of the lack of autonomy in the manipulation of build pipelines, which is the creation of a DSL that simplifies several configurations from the point of view of the developer. This chapter will also explain the several decisions taken over time that led to the design of the syntax of the proposed DSL.
- Chapter 5 - Solution Implementation: A more detailed description of how the solution proposed in chapter 4 has been implemented. The most important steps taken during implementation are specified, and the technologies, tools, and languages used to create the DSL. In this chapter, we can see all the implementation details that are important for understanding how the solution works.
- Chapter 6 - Experimental Validation: The experimental validations are presented through a set of metrics that show that the proposed solution achieves the objectives presented in this chapter.
- Chapter 7 - Conclusion: Summarizes the work presented in this dissertation, adding final conclusions. Finally some directions on possible future work are presented.

Chapter 2

Industrial Context

In the previous chapter the structure of the document, the problem, and the motivations to solve it, was presented. The industrial context in which this problem will be solved is presented in this chapter, sections 2.2 and 2.3. In sections 2.4 and 2.5 we can also find an explanation of the details needed to better understand the problem, and the needs to solve it.

2.1 Introduction

The fast technological growth that the world is currently experiencing means that companies need to be constantly evolving to not be outdone by their competitors, and more important, to be able to meet the market and customer needs. This leads to a challenge in developing and making software releases continuously, so that value is delivered to the customer faster and more often than it was a few years ago.

These needs are something that OutSystems currently also faces. In the case of this company, it led to a reorganization of R&D to meet these needs and to improve the productivity of the developers. The Engineering Productivity Group, discussed in more detail in section 2.4, is an example of a R&D group of teams that focuses on improving software development processes and tools, and increasing developer productivity.

In this chapter is presented the industrial context of OutSystems where this dissertation is inserted. The main goal is to present in more depth the company OutSystems, its platform, and the development techniques used by R&D. In addition, some details are presented about the OutSystems product to show its complexity. Although the context presented in this chapter is specific to OutSystems, some concepts and problems are common in companies that do software development through agile methodologies, as we will see in chapter 3.

2.2 The Company and its main Product

OutSystems is an international company, founded in 2001, and which since 2016 has remained the market leader in Low-Code Platforms for Application Development [Out19, Out16]. OutSystems's goal is to make life easier for developers who in their daily lives have to develop various types of software, allowing them to create web and mobile applications through a visual language.

The visual language lets the developer work on an abstraction that allows to model and design four different layers of applications: Data, Process, Logic and User Interface. An application, once compiled using a 1-Click Publish® approach, which generates all the application code and all database assets, can be automatically deployed into execution environment across multiple

devices and over different stack combinations.

The OutSystems product has been developed in Portugal by several teams using mostly C#, following a CI/CD software development approach. This platform is represented with all its components in the figure 2.1, being a software that can be supplied to the client as a solution on premises, or in the cloud as Platform as a Service (PaaS). The platform can be divided into three main components:

- Development Environment (DE);
- Platform Server (PS);
- LifeTime.

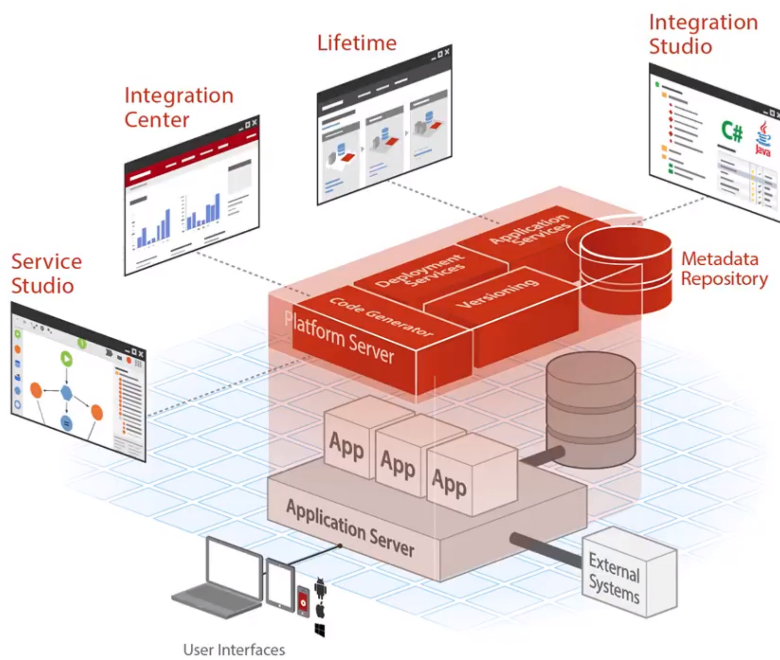


Figure 2.1: OutSystems product [Out18b].

DE is composed of Service Studio (SS) and Integration Studio (IS) [Out18b]. SS is the Integrated Development Environment (IDE) that allows developers to develop web or mobile applications using visual models [Out18b]. IS allows developers to integrate with external systems (microservices, databases) also allowing the creation of components that extend the functionality of the OutSystems product [Out18b].

PS is composed of a set of components needed to generate, optimize, compile, and deploy mobile or web applications through three specialized services [Out18a]:

- Code Generator - This process is triggered when the developer publishes their application or module, and is responsible for generating applications with code that is produced in the IDE;
- Deployment Services - Deploys the application generated for each front-end;
- Application Services - They manage the execution of scheduled tasks and is also responsible for managing error logs, auditing, and performance metrics.

The Service Center is also part of the PS and allows customers to manage, monitor and troubleshoot their OutSystems applications [Out18b].

LifeTime allows management of different environments, define security policies, and manage the entire lifecycle of OutSystems applications from development to production [Out18b].

We can see in this brief description that the OutSystems product is composed by multiple services that are developed using multiple technologies. The next sections addresses its development and validation process.

2.3 The Development Team, Methodology and Paradigms

OutSystems currently has over 1000 employees, and the R&D department, responsible for the evolution of the OutSystems product, currently has around 150 employees who are divided by different teams. These teams can be separated into three main groups: those working within the core OutSystems product, those working on extending services around the OutSystems product, and support teams.

The teams working on the core product, described in section 2.2, work actively on the same code base, making several changes to the same code. They work on the same repository (Subversion (SVN)) using a variation of trunk based development, single branch development. Trunk based development is a type of development where all changes are made to the main branch, and a new feature is developed in the main branch and if a release is made, an unfinished feature shouldn't cause problems that affect the release [Dav18, Kam18]. This type of development is recommended to be able to implement and practice CI/CD [Mug17, Vis15, NS13]. The business model that OutSystems currently presents requires that each major version has to be maintained for at least 2 years, which is why R&D opts for single branch development, one for each supported major version, a practice that is common in the software development industry [GFFLB13]. These teams validate their development using the same validation process represented and explained in section 2.4.

The teams responsible for extending the services around the OutSystems product, like mobile stack fit team which build rich and reliable mobile apps, due to the fact that they do not work on the same OutSystems product code base, may have a different development model. These kinds of teams have more autonomy on their way of working, not only on their releases but also on their development process, therefore they may have different CI/CD practices implemented. These teams usually have their own repository and their own validation process, which may or may not be automated.

Finally, there is the group of support teams that help to increase the developers' productivity by supporting the tools and processes that enable and promote the practice of CI/CD.

All these teams develop new code and create new tests continuously. The development at R&D is done using agile methodologies that allow incremental deliverables, and continuous learning and planning, and there are always a collaboration between the team [Aar17]. This methodol-

ogy is based on the acceleration of the product so that a CD with quality and fast feedback is obtained on the tasks performed [TTV+17, KS08, Eri18].

The CI/CD practices are what helps a company to be agile when it comes to software development [Tun18b]. By using these practices, OutSystems can accelerate the development of its product to continuously deliver value to customers. This dissertation aims to help in this effort.

2.4 Platform Development

The OutSystems product code is mostly written using C#, but other programming languages like Java, Typescript, JavaScript and the OutSystems language itself are also used.

The agile software development methodologies and various CI/CD techniques used by OutSystems help achieve its goal of staying in the lead. However, due to having a product with a market that demands that it grows, they need to accelerate it's development, so as not to be outdone by their competitors. For this to happen, it is necessary that this company can maximize its process of development of the platform. OutSystems, during its first 17 years of life, it was able to have 1000 clients, and thanks to the growth of the market it is expected that it can have more 1000 customers, however in a time frame of 17 months. This hyper-growth and the need for this company to remain a leader in low-code makes it necessary for the product to continue to evolve at a higher speed to meet the needs of the market/customers, and not be overtaken by competitors.

The way the company wants to accelerate product development is not to make teams work more, but to invest in research and to better understand the customer so that a product evolves based on the feedback they provide. To obtain feedback to guide the development of the product, it is necessary to launch new features, experiments, and proof of concepts to the market as soon and as fast as possible. For this to be possible, they need to eliminate impediments that may exist, such as the handovers of dependencies between different teams that affect the lead time.

The OutSystems product has been developed using the practices of CI/CD which are being implemented incrementally since 2015. For them to be fully adopted there is a set of necessary guidelines that must be followed. An example of practices that need to be adopted to be able to say that we have the CI/CD implemented in the company are the following ones [HF10, DMDG07]:

- Single Development Branch Per Major Version - To share code continuously. Code integration is validated earlier and faster;
- Commit code frequently - Should commit code frequently (several times a day) so that the integration does not take up too much developer time;
- Do not commit broken code - Should test the modifications made before doing commits, to not commit code that does not work;
- Fix broken builds immediately - Errors must be corrected immediately to prevent other developers from being stuck without being able to commit;

- Run private builds - Developers should run the code on their machine to avoid broken builds;
- Avoid getting broken code - When the build is broken developers should not checkout the repository;
- Write automated developer tests - To have a CI/CD system, build and tests should be automated;
- All tests and inspections must pass - All tests must pass. If one fails, developers must not ignore it and understand what is happening;
- Create a repeatable, reliable process for releasing software - This way developers can release software in a simple way since each part of the process has already been tested hundreds of times;
- Automate almost everything - Although there are things that are impossible to automate, such as exploratory testing, automation is a prerequisite for a build pipeline because it is only through automation that people will have what they need when they click on a button;
- If it hurts, do it more frequently, and bring the pain forward - In doing a painful process more often, developers will react to pain by invoking an action;
- Build quality in - The tests should not be a phase, but something that is done continuously throughout development;
- Done means released - We can only say that a story or feature is made when developers deliver features to customers;
- Everybody is responsible for the delivery process - All people, from the developers to the tester, must be responsible for the delivery process, and the communication between the different parties must happen on an ongoing basis;
- Continuous Improvement - The delivery process must evolve following the evolution of the applications.

The first 9 guidelines refer to Continuous Integration (CI) practices that are already implemented in the R&D development culture, however, the guidelines for CD currently (2019) are not yet fully implemented. This incremental implementation will increase the speed of development, thus meeting the needs that hyper-growth can bring to the company. The hyper-growth expected by the company puts great pressure on the adoption of these practices to avoid problems arising from the inefficient operation of R&D teams. This dissertation aims to help to improve the implementation of CI/CD practices by the R&D teams.

The practice of CI is characterized as a software development practice, where developers integrate their work several times a day [Mar06, DMDG07]. This integration must go through automated processes, such as a test build, to detect integration errors as early as possible [Mar06]. This leads to faster feedback, making it easier to detect and fix because the developer still has context about where the error was introduced [DMDG07]. The major focus of CI practice is therefore to obtain rapid feedback on the changes made by developers, making the product available at any time, so that software integration is considered a “nonevent” [DMDG07].

The CD practice is characterized as a practice where the process of delivering incremental changes to customers is optimized, making the software available for release at any time. The goal of this practice is to find ways to reduce cycle time, which is the time from when a new feature is thought until it reaches the user. This practice also wants to accelerate the delivery because the quick feedback from the customer, allows knowing exactly the path the product should follow [HF10].

To achieve CI/CD practice, there are build pipelines that make the software delivery process automated [HF10]. Build pipelines have the role of ensuring that all changes go through the same validation process on their way to the release, in an automatic way. Automation offers the benefit of failing fast because if the build pipeline is well defined the faster tests run first and those with long run-time later. A correct definition of the build pipelines causes the bugs to be detected earlier and consequently corrected in a faster period of time [Sam17]. All this combined allows the company to adapt quickly to changes in business requirements, an idea on which the OutSystems product is based, thus raising the importance of build pipelines in the life of engineers. So, this practice should be implemented because of the benefits it provides, like, making processes visible to all, and improving feedback, removing manual steps that are intensive and error-prone among others [HF10, DPvH18, Son15, Mug17].

If CI/CD practices are not implemented, several problems can occur, such as [DMDG07]:

- Lack of Deployable Software - Not being able to have deployable software at any time;
- Late Discovery of Defects - Not be able to discover defects as early as possible, and when they are discovered the developer may no longer have the necessary context to correct them;
- Lack of Visibility - If frequent builds doesn't run, the current status of the project remains unknown;
- Low-Quality Software - Failing to run tests often, defects can be introduced into the code and discovered very late;
- Low feedback loop - It can lead to problems to be identified and resolved at a late stage.

Thus, to avoid above-mentioned problems, several validations are performed, ensuring the quality of software that we are building.

2.5 Quality Assurance

To ensure product quality, OutSystems uses a set of build pipelines that contain several Stages and run whenever a commit is made to ensure everything runs as expected. The fact that the OutSystems product is extremely complex, leads to an extremely complex and long validation process, which has been constantly evolving over time, following the needs of the OutSystems product. This validation process is built using build pipelines, which must be configured correctly to be able to validate quickly the platform components that are on the responsibility of the different development teams.

From the beginning of the implementation of CI/CD practices at OutSystems, the R&D engineers created their build pipeline to validate the entire OutSystems product. Initially, this process was something simple that validated the entire OutSystems product as one, as represented by the figure 2.2. However, due to the specific needs of different teams, and to changes in infrastructure, among other needs that have been appearing over time, the build pipeline evolved. These changes, over time, made something that was initially simple, into something highly complex, and at this point the process of validation of the product is represented by the figure 2.3. Due to the constant evolution of the OutSystems product and consequent evolution of its build pipelines, this architecture will rapidly become outdated.

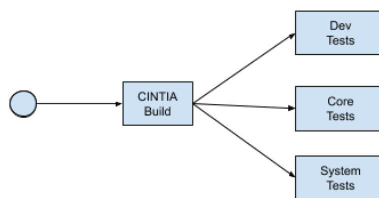


Figure 2.2: The first build pipeline created to validate the OutSystems product.

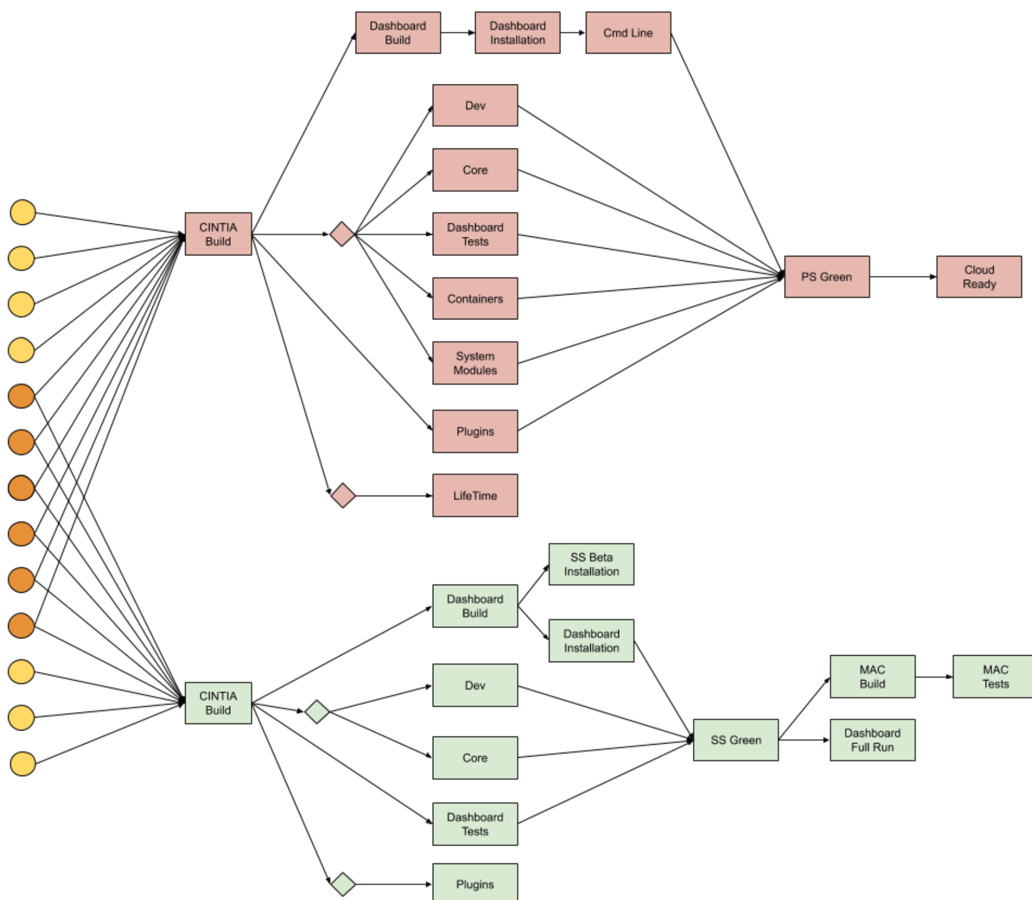


Figure 2.3: The current build pipeline that validates the OutSystems product[1].

In the figure 2.3, the build pipeline is represented by two colors because they represent dif-

[1]This architecture will rapidly become outdated.

ferent components. The green process represents the build pipeline in which the SS will have to pass, while the red represents the PS build pipeline. This separation of the build pipelines for the SS and PS was one of the evolutions that happened due to the needs of the teams to have different release cadence for SS and PS, allowing SS teams to have a faster feedback loop because the release and distribution process of SS is much simpler for the customers. However, in the figure we can see that both build pipelines start from a set of circles. The circles that appear in orange represents the SVN repository folders that activate both build pipelines, and the yellow circles represents SVN repository folders that activate a particular build pipeline (SS or PS). Orange circles are common materials to both build pipelines due to the complexity and monolithic architecture of the OutSystems platform, which means that, for now, it is not possible to separate these two systems completely, so that they can be in separate repositories.

Unfortunately, given the complexity of pipelines, different R&D teams find it difficult to follow and understand all the terms, concepts and techniques used to build a validation process of OutSystems product. This led the evolution of build pipelines, their maintenance, and the creation of new build pipelines specific to each team to be made by the Engineering Productivity Team. This team is the only one that can support teams that want to create their own build pipelines, and is also the only one able to change this complex build pipeline represented in the figure 2.3. The fact that there is only a small team (7 people) that has the expertise needed to handle build pipelines, coupled with the need to accelerate product development to meet market needs, it is expected that if everything remains as it is now, this team will become a bottleneck. This workload led the team to separate in 3 different teams, based on different technical areas, to be able to follow the autonomy and acceleration existing in the development of the product.

As already mentioned, the OutSystems product will be constantly evolving, so the build pipelines will follow this evolution. The fact that teams currently do not have enough knowledge to understand the current complexity that builds pipelines present, prevents them from making changes to the build pipeline. This leads to the teams not having the autonomy to accelerate their development. One way to give autonomy to the teams would be to make them spend some of their time learning the concepts and tools used in the construction of build pipelines. However, because developers already have to know how to work with various tools and technologies in their daily lives, forcing them to know about another complex tool introduces an unnecessary overload in the lives of these engineers.

2.6 Conclusion

In this chapter, we have an insight into the industrial context where this dissertation is inserted, focusing on the OutSystems product, how it is developed, and especially on how it is validated, being this the focus of the dissertation.

Understanding the company context is essential for understanding the context where the problem is present. With this context, we can see that OutSystems, due to the complexity of its product, has a very complex validation process. The problem in OutSystems, and where this dissertation is inserted, is that the great complexity of the product together with the lack of knowledge about build pipelines in the different R&D teams, make them unable to change the

build pipelines and build their own validation process. This, coupled with the fact that R&D is constantly growing, can cause Engineering Productivity Group to become a bottleneck and can become a source of delay rather than acceleration. Although so far this is still not a problem, there are already signs that this problem exists, but due to its scale are still easily manageable.

The main objective of this dissertation is to allow the different teams to make changes, evolve the build pipeline independently without needing the help of other teams, and without knowing all the technology behind the build pipelines. This will give teams autonomy, and focus on their changes without any complexity involved allowing their validations to be made in a timely manner and also allowing acceleration of product development.

Chapter 3

State of the art

In this chapter, in section 3.2, presents the state of the art in the area of CI/CD practices, and in section 3.3 are addressed several cases of companies attempting to implement these methodologies and faced with several problems, some of them similar to what this dissertation proposes to solve. With this industrial and academic context, it is possible to see how others solved their problems related to the construction of build pipelines, to find a suitable approach for this case study.

3.1 Introduction

CI/CD development practices have gained wide adoption in Software Development industry, having already demonstrated how effective these techniques are in software development [LMP+15, Son15, LSS18, Swa18, CS16, Fer18]. These implementation practices appear in a natural way for companies that are bootstrapping their teams, as was the case of Spotify [Tay18, Hen14], but on companies that have legacy code and internal habits its much harder to achieve, as presented in an email exchange with Mario Fernandez Lead Developer at ThoughtWorks [Fer18], but this may be possible as shown in section 3.2.

This process of transition to CI/CD usually comes with some associated challenges. Realizing these challenges and how they relate to the creation of bottlenecks that delay development in the different teams helps to understand how these companies solved their problems of lack of autonomy of their teams. Although CI/CD practices does not have well-defined tools and steps, companies have a shared mindset with other companies, and a trial/error practice so they can know how to implement these practices in the company, being what can result in one company, may not make sense to another [Cor]. The conclusions got from the contact with the industry will then be used as inspiration for a solution to the problem presented in chapter 1.

Although the research has been done globally for CI/CD practices, it is also directed to when the teams depend on another in the creation of build pipelines. The challenges were also analyzed in this area, and how they were overcome.

With this, several companies use cases will be analysed in order to learn what their approaches were, and the lessons they learned from the implementation that they did.

3.2 CI/CD Practices in the Industries

Nowadays, many companies have already realized the benefits that the implementation of CI/CD practices can bring. By using these practices, the industry manages to soften the existing pains

in software development, such as the difficulty of integration, making it easier for a developer to work.

As described in chapter 2, the CI/CD practices can be divided into two parts, CI and CD.

CI has the main goal is to make software integration a "nonevent" [DMDG07]. Duvall et al. wrote a book about CI [DMDG07], where they describe that the tensest moments in which a development team passes, is precisely the moment where it is necessary to make the integration of what has been developed over the months, being even known as "integration hell". Companies that implement CI practices can then make the integration moment something simple, but for this, it is necessary for developers to integrate their work several times a day. This way, errors will be found sooner and faster, and the developer will still have the necessary context to fix them faster. By doing this, the developer has more time to focus on developing new features, rather than looking for mistakes and bugs.

CD practice encompasses all CI practices and a few more, and the goal is to deliver high-quality software in an efficient, fast, and reliable manner [HF10]. Humble and Farley in a book about CD [HF10], show that another tense day in a development team, is the release day since it is a process where a lot can go wrong if the steps are not perfectly executed. Similar to what has been said about CI, it is also wanted here that the release process is something that is no longer stressful and does not occupy the weekend of operators and developers [NS13]. For this to be achieved it is necessary to create repetitive, reliable, and automated processes that can make the software release process simpler.

With these two practices together, we can make the delivery of the software to the customer at any time. Build pipelines are a common technique for implementing these practices more easily. Several companies then began implementing CI/CD practices to address problems related to the speed of software development and delivery software to the customer. Therefore, in the following paragraphs are described cases where these practices were implemented.

Soni published an article about the implementation of these practices, and about a build pipeline for improving the speed and quality of its software delivery [Son15]. He shows a use case of an insurance industry that had various challenges in the software development process. These challenges include continuously changing customer expectations, slow customer feedback integration processes, slow and error-prone release processes, and others. To address these issues, they decided to implement a build pipeline that automated the end to end process, allowing faster deliveries, thereby reducing feedback cycles with the customer.

Neely and Stolt presented the case of the company Rally Software that made the transition to CD [NS13]. With this change, they have been able to move from an 8-week release period to a CD model where they can make product releases whenever they want. They felt the need to switch to CD development methodologies because they were only able to launch new features and fix low priority defects every 8 weeks. To address this, the company was able to make the transition to CD through incremental steps, through the creation of documentation, and the use of techniques that allowed the detection of problems before customers were affected. By applying these practices, they have achieved the expected benefits mentioned above. Neely, when contacted via email with the question of whether there was any team responsible for cre-

ating and managing build pipelines, or if each team had its own build pipelines, replied that he had already worked on these two approaches [Nee18]. He also mentioned that in cases where the companies are smaller the teams have their own build pipelines, and in the case of larger companies, there is a team responsible for creating tools and standards for the other teams to use.

Fernandez lead developer at ThoughtWorks wrote an online article where he talks about making CI with the right tools [Mar18]. When contacted about having already worked with a company that after a few years of existence decided to implement the practices of CI/CD [Fer18], he mentioned that he worked with a company with 10 years of legacy code, where the releases happened every 3 weeks. He also mentioned that the change to CD had several problems, such as unreliable and flaky tests, few unit tests and many end to end tests, little automation, and non-existent collaboration between Dev teams and Ops teams. There were so many problems that, according to him, this transition was never fully made. Following this example, we can see that this process of change and adoption of CI/CD practices is not something that can be achieved only with the right tools, but also with the change of mindset of the people involved in the software development and delivery process.

Taylor presented the case of how the Spotify CD process works, at Code-Conf in Copenhagen [Tay18]. From the beginning, this company has focused on having autonomous teams with a well-defined goal. This autonomy makes the teams responsible for their systems, for the technologies they use, and for the build pipelines, not needing a team for operations. He also explains that because the practice of CD was born with the company, made it less likely to encounter problems that appear when the CD process is implemented incrementally in a company with several years of work. As Spotify managed to have a CD process well implemented from the start, it made the teams fully autonomous, without problems of dependencies between teams.

With the implementation of these CI/CD techniques by these companies, they were able to solve the problems of lack of speed and reliability, both in the integration of the software and in the delivery of software to the client, however it gave rise to other problems, such as the case of lack autonomy of the various teams. This will be the subject of the next section.

However, here we have also seen that if teams have full autonomy to do their job and achieve their goals, they can develop software quickly, delivering fast value to the customer. This is something that OutSystems is aiming for, and this dissertation helps to make this happen by building a solution that will give the teams autonomy to develop their build pipelines.

3.3 Autonomy and Elimination of Team's Dependencies

The dependence of teams on a particular team for a task to execute is a problem that CI/CD practices can give rise to if they are not implemented from the outset, and this is a problem that OutSystems is already experiencing. We will then analyse how companies that have implemented CI/CD practices have avoided or solved this problem.

A case that prevented bottlenecks from appearing with the implementation of CD practices was

the case of Spotify already presented in section 3.2. In this case, we can see that if the CI/CD practices were implemented in a company from the outset, where the teams have a well-defined objective and the total autonomy to do everything to achieve this goal, it would have been able to avoid bottlenecks that can arise with the need for acceleration in product development.

As OutSystems is a company with 18 years of existence and with legacy code, this example presented by Spotify is something that does not make sense as a viable approach. Therefore, this research was directed to case studies of companies that started to implement the practices of CI/CD, and build pipelines, after a few years of existence.

The approaches found to address the existing dependencies between teams are divided into three topics:

- The use of existing CI/CD tools without any layer of abstraction;
- Creation of build pipelines libraries, where developers through configuration files can have the desired build pipelines;
- Creation or use of a DSL based on existing CI/CD tools, removing complexity in creating build pipelines. These DSLs are declarative, being specialized in being configuration files.

These three topics will now be covered in more detail, to realize their advantages and disadvantages, so that we can then propose a solution to the concrete problem of OutSystems.

3.3.1 Use of existing CI/CD tools

Tune from Navico, gave a talk at Øredev 2018 under the title Sociotechnical Architecture: Aligning Teams and Software for Continuous Delivery [Tun18b], where he mentioned how they wanted to deliver software and value to customers more frequent giving developers total autonomy to understand how to get there. Since this is a subject that encompasses the practices of CI/CD and build pipelines, he was contacted about who should be responsible for the creation and management of build pipelines, and was discovered that he has seen three different approaches that worked on different scenarios [Tun18a]:

- In most companies, teams have their own build pipelines;
- In some companies, the platform team builds standardized build pipelines for all teams;
- In some companies, the build pipelines are built by a single team for efficiency and compliance, which turns out to be a bottleneck slowing down the other teams.

From these approaches, the approach of teams to have their own build pipelines is the one that matters most in the context of OutSystems. In this case, the teams use a Jenkins DSL in Groovy[1] that allow them to create the build pipelines of their choice with total freedom [Tun18a].

An analysis of this DSL, which Tune mentioned, was made to see if the problem presented could be solved with the requirements specified in chapter 1. Jenkins provides the user with the possibility to program build pipelines graphically through plugins, and through pipeline as code in

[1] https://github.com/hmrc/jenkins-jobs/blob/master/jobs/ci_open_jenkins/build/auth.groovy

a scripted or declarative way.

Pipeline as code scripted

The pipeline as code scripted is built with Groovy, which allows creating build pipelines using the features provided by this language, making this DSL quite expressive, following a more imperative programming model [Jen17b].

This DSL is named "Jenkins job-dsl-plugin", and code reuse cannot be done at the level of the structure of the build pipeline, and this is one of the capabilities that OutSystems needs a tool to have since this company needs to support different major versions of the product. Practicing CI/CD makes them have build pipelines for each major version which have several similarities, so it is important to have reuse of pipeline architecture in the solution to be implemented.

With "Jenkins job-dsl-plugin" is possible to create complex build pipelines, however, given its structure, the concepts used are closely linked to Jenkins which makes it necessary to know some concepts to work with this DSL.

Then there is the fact that developers have to know Groovy, and this being a language not used in the industrial context of this dissertation, it represents a high learning curve. The fact that Groovy is not used by some companies, and because its learning curve is accentuated, has made Jenkins opt for a pipeline as code most declarative [Jen17c].

Pipeline as code declarative

As already mentioned, Jenkins when realizing that the pipeline as code scripted has a very sharp learning curve decided to create a declarative solution to make it easier to create build pipelines [Jen17c].

This pipeline as code is a declarative language that makes necessary to know a large set of configurations in order to create complex build pipelines. Here concepts are not abstracted, so it is necessary for a developer to know a lot of the concepts related to build pipelines in order to create one. Linking the concepts of this pipeline to Jenkins makes it difficult to compile this syntax for another CI/CD tool.

In relation to code reuse, similar to the pipeline as code scripted, developers can only reuse code at the level of the tasks and not at the level of the structure of the build pipeline.

There are several CI/CD tools that allow the construction of build pipelines that validate the software developed, and part of them have been studied as part of this research.

One way to give autonomy and remove the dependence between teams in build pipeline construction and management is to give teams the freedom to use the tools they think fit their use case. To determine if any of the currently known CI/CD tools solve the problem presented, an analysis was made of the features that OutSystems considers most important to validate their product. The CI/CD tools tested were:

- GoCD[2];
- Concourse[3];
- Azure DevOps[4];

[2] https://www.gocd.org/

[3] https://concourse-ci.org/

[4] https://azure.microsoft.com/en-us/services/devops/

- Bamboo[5];
- TravisCI[6];
- Jenkins[7].

The build pipeline created on these different tools is represented in the figure 3.1. This build pipeline is triggered when it detects a Pull Request (PR) in GitHub, as represented in figure 3.1. After the PR is detected, the build pipeline starts by checking whether a JSON file has the correct syntax, and if so, unit and integration tests are done in parallel against that JSON file. After these validations the automatic merge of the PR in GitHub is done.

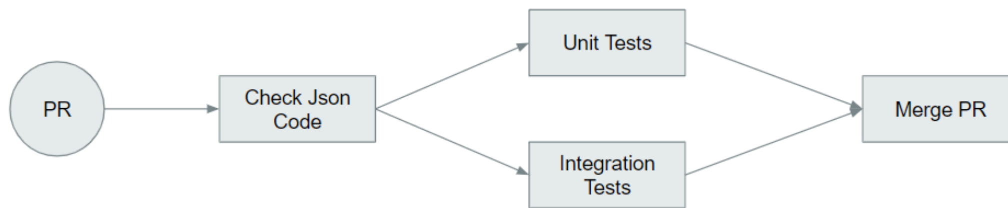


Figure 3.1: Pipeline used to test different CI/CD tools.

The goal is then to apply this build pipeline in different CI/CD tools to understand all the capabilities that the tool offers.

In the table 3.1 we can see all the characteristics taken out when testing the tools with the build pipeline shown in figure 3.1.

OutSystems has defined the characteristics represented in the table 3.1 as the most important capabilities that each CI/CD tool must have to validate their product. The "Rerun a Specific Stage", "PR and Sends Notification", "Containers", "Agents", and "Parallelism" capabilities represent existing needs for validation of OutSystems product to run automatically and quickly, and in case of infrastructural errors to recover easily. The other capabilities, they are important because they allow having an easy mental model of the constructed build pipeline, also facilitating the construction of either simple or complex build pipelines. The build pipeline represented in figure 3.1, which served as the basis for this analysis, encompassed all the practices that a CI/CD tool needs to validate the OutSystems product, so it is important that the tool being tested can implement the described build pipeline. In addition, some of these metrics refer to the topics presented in chapter 1, which represent the characteristics that we want the solution to the presented problem to have.

From this table we realized that we were able to achieve our goal, that is, we were able to implement the build pipeline presented in 4 of the 6 tools. The reason for not being able to implement in TravisCI was conflicts with the automatic merge, whereas in Bamboo it was due to the fact that the build pipeline cannot be triggered by detecting PR from GitHub because this

[5]https://www.atlassian.com/software/bamboo
[6]https://travis-ci.org/
[7]https://jenkins.io/

Table 3.1: Characteristics of each CI/CD tool important for the context of OutSystems.[1]

	GoCD	Concourse	Azure DevOps	Bamboo	TravisCI	Jenkins
Goal Achieved	X	X	X			X
Rerun a Specific Stage	X	X			X	
Detects PR and Sends Notification	X	X	X		X	X
Containers	X	X	X	X	X	X
Agents	X		X	X	X	X
Pipeline as Code	X	X	X		X	X
Graphic Interface	X		X	X		X
Value Stream Map	X	X		X		X
Parallelism	X	X	X	X	X[2]	X

[1] The analysis was done during the week 10/10/2018, so what was valid at this time may no longer be.
[2] Not tested, it only can be tested in the enterprise version.

tool does not have this feature implemented.

In relation to doing rerun to a specific Stage, it was only possible to do with half of the tested tools, and this is a necessary feature for the context of build pipeline that validates the OutSystems Product.

The topic of detecting PR from GitHub and sending a notification to the GitHub of PR validation state, is a feature that unfortunately is not available in Bamboo, but is being considered for being added in the future [Kry18].

All these tested tools let run the build pipelines in containers, and only the Concourse does not allow the build pipelines to run on agents, which may be a limitation in the case of OutSystems.

The fact that build pipelines can be created through code, only Bamboo does not allow this to happen, forcing to create build pipelines with a graphical interface. The fact that we can create build pipelines with a graphical interface, only the Concourse and TravisCI do not allow, leaving only the possibility for the user to create their build pipelines through code.

Regarding Value Stream Map (VSM), which is a visualization model that shows the actual execution of a build pipeline, is a very powerful tool to visualize and analyze the actual validation process. This feature is not available for Azure DevOps and TravisCI, and is important because it is through VSM that the developer in creating the build pipeline can see if they are following the architecture originally planned or if he made a mistake and are already moving in another direction. It is also a very useful tool for troubleshooting in highly complex build pipelines.

Finally, we have the parallelism, where all these tools allow to run the Stages in parallel. However, it should be noted that although TravisCI has this feature, it was not possible to test because it was only available in the enterprise version.

From the carried out analysis of CI/CD tools, we conclude that:

- If a tool only allows the creation of build pipelines through code, good documentation should be provided to guide the user about the capabilities that this code offers.
- If the tool only allows the construction of build pipelines through a graphical interface, there must be a simple interface that guides the user in the right direction.

A combination of the construction of build pipelines with code, with the help of a graphical interface seems to be a solution that helps the user make more correct use of the tools and leads to the creation of build pipelines more easily. However, through the analysis of the table 3.1, it is concluded that the GoCD fulfills all the requirements analyzed, which is why the main tool being used to validate the OutSystems product is the GoCD. Looking at Jenkins, since it just does not meet the "Rerun a Specific Stage" requirement, this is a tool that is just being used by some teams at OutSystems R&D, the ones that can have simple and independent build pipelines. Despite these two tools meet the requirements that OutSystems finds important, they can not solve the problem presented. All of these tools require that developer have a deep knowledge of the concepts of the tools and high expertise to master all the techniques and best practices to use these tools in complex scenarios. This represents a learning curve associated with the concepts of build pipelines, and with the tool. This learning curve turns out to be quite steep in some of these tools, being Jenkins the one that stood out the most.

3.3.2 Creation of Pipelines Libraries

Coraboeuf, at the All Day DevOps conference [Cor16], presented an approach referring to a case of a financial industry company where the practices of CI/CD have been implemented, and which has had the same problem that one wants to avoid happening in OutSystems with the solution proposed in this dissertation. This company, due to the needs of having products with different versions, created a team that was responsible for the creation and maintenance of build pipelines that validate their product. Over time, product needs led to build pipelines becoming complex and long-running. To prevent this team from becoming a bottleneck, since it was the only one responsible for the build pipelines, they decided to implement a self-service approach, safe, simple and extensible. The approach was the implementation of build pipeline libraries, where developers can have the build pipeline they need by simply resorting to a configuration file where they define the properties they need. In contact with Coraboeuf [Cor], he says that this approach was effective in this company because all the projects were very similar, and with that, he got new projects created with code and build pipelines pre-configured. However, the team that was fully responsible for the build pipelines remained responsible for maintaining, testing, and documenting the build pipeline library. Coraboeuf also said that in the company where he currently works, this implementation of pipelines libraries does not make sense, and what is currently in operation is pipeline as code, pointing out the fact that what can work for a company may no longer make sense to another. This shows that this solution can be easily applied to companies that have very similar build pipelines between different teams and that

in companies where this does not happen this solution is no longer valid.

3.3.3 Creation and Use of DSL

Fernandez, lead developer at ThoughtWorks, wrote an online article where he talks about making CI with the right tools [Mar18]. This article discusses a project in which he worked on and where there was the problem of making fast and reliable deliveries and also speaks of a DSL that they used internally to build pipelines through pipeline as code. Questioned about this DSL [Fer18], he said that they used to build everything on a centrally maintained Jenkins instance. In order to cover all the steps of build, test, and deploy, and as Jenkins pipelines concept did not exist at the time, they built a DSL that allowed build pipelines to cover these steps, and allowed the deploy of the applications simpler to do. Despite these benefits, the DSL grew over time and eventually became a black box where everyone was afraid to touch, since making a simple and small changes in it was something that become very challenging and time consuming. This solution has solved the problems, however, if we don't be careful creating and evolving this DSL, can lead to the creation of a solution that quickly becomes unusable.

Yu and Hender from Pivotal, gave a talk entitled "What if you treat your CI pipeline as a product?" in the pipeline conference 2018 [DM18], where they talked about how they made the transition from a large and single team to several small teams. This separation in several teams meant that none of them wanted to own ownership of the build pipelines, causing that over time it became painful to make the build pipelines evolve. Thus, it was decided to build an approach that gave build pipelines ownership to all teams. In the process of designing the solution they realized that in addition to being able to give ownership of the build pipelines to the teams, they could still create a product that followed by default the best practices, and that it was simple to work in order to waste less time in configuration build pipelines, and more in developing new features for customers. Despite having embraced this approach in DSLs, in contact with Denise Yu [Yu18] for more information on this approach, she states that she does not see this as a DSL, but rather as a YAML-based interface. This approach, which Pivotal provided, consists of a Ruby script that receives a small block of YAML that the various teams easily write, and gives rise to a larger YAML block that is the definition of a Concourse pipeline. This solution worked for their use case, which despite being restricted was shared by several teams in the company, and has been undergoing several evolutions to support cases of specific uses of various teams, use cases that are hidden from others through feature flags. Yu in this direct contact, also said that in hindsight they learned that if it were today they would have invested in a more composable tool, instead of building a large build pipeline that supports several extreme cases. This solution worked for Pivotal's use case because the different teams needed very similar build pipelines, and whenever there was one that was different, they built a solution to support that edge case. However, with the evolutions turned out to be a difficult solution to evolve, similar to the case presented previously.

Jacob et al. present an approach to the creation of a DSL for the Home Depot company [JKSS17]. The company's website receives more than 8 million hits per day, which led to the need to make improvements to the search platform. When changes are made to have a positive impact on one side, it can lead to a negative impact on another side, and to prevent this from happening, they applied machine learning techniques to make tests that assess the impact of change.

Because this process involves intensive computation with bigdata, the build pipeline they had implemented in Unix scripting exposed limitations that motivated them to design a DSL. This DSL allowed developers to design a build pipeline through a sequence of actions. They have created a framework that converts DSL to a Concourse pipeline, which can easily be integrated with other build pipelines if needed. This DSL seemed promising to solve the problem presented, but no more information about it was made available because it was confidential. This made it impossible to proceed with the DSL analysis.

Stewart engineering manager at Riot Games, gave a talk on dockercon [Ste16]. Although the area presents a problem that is different from what we are investigating, the need to change is similar to what is expected to happen in OutSystems. The problem they face is in the area of infrastructure and not in the area of build pipelines. This problem consisted of a team that at one point began to receive many tickets asking for several configurations for different machines, which began to create a great overload in the team making it a bottleneck. This team did not want to delay those who were dependent on them, so they thought of an approach to allow teams to create their own environments in an independent and autonomous way. They implemented configuration as code, which allowed teams to create and define their own environments, disappearing a build team, for all teams to become build teams. The implementation of a configuration as a code solution was sufficient to solve the problems related to this company because by specifying a series of variables it is possible to generate the necessary environments for a team. However, this solution may present the problem of having a large set of configurations to be specified, causing the developer to have prior knowledge about concepts related to creating environments.

Exner, director of AWS Dev Resources, at the AWS Summit gave a talk that shows how amazon has evolved into a DevOps culture, and the tools they have created to support the new processes [Exn15]. Amazon.com in 2001 began with a monolithic architecture that began to create bottlenecks within the development teams. To eliminate them, they decided to make several changes at the architectural, organizational and cultural levels. One can highlight the organizational changes where the teams became small and autonomous and began to adopt agile models, getting end to end ownership. When they applied these changes, they realized that they needed new tools that, among other things, were self-serviced and supported agile processes. Of the various tools that emerged, there was an internal tool named “pipelines” that allowed to produce more efficient processes, where the teams began to model their own release process. This tool eventually gave birth to a new AWS product, the AWS CodePipeline[8] which is a CI/CD tool similar to those tested in subsection 3.3.1. This tool allows the specification of build pipelines visually or through pipeline as code written in YAML. The visual manipulation of build pipelines is simple to do, however, it is only ideal for creating simple build pipelines. For the creation of more complex build pipelines, it is better to use the pipeline as code, which once again, similar to the tools tested, does not allow the concepts of build pipelines to be abstracted, nor does it allow the same syntax to generate build pipelines for different CI/CD tools.

As mentioned in subsection 3.3.3 Damien Coraboeuf currently works in a company where different teams use pipeline as code to develop build pipelines. In the contact with Damien [Cor], when asked how they work inside the company with the use of pipeline as code, he mentioned

[8] https://aws.amazon.com/pt/codepipeline/

that there are 3 ways to work. There are teams that can be 100% autonomous and do not need any help from other teams. Others make use of libraries that Damien and other teams carefully create and document. Finally, for the more complex build pipelines is Damien himself who defines the build pipelines entirely. Here, we can see that even with the use of pipeline as code, the manipulation of build pipelines is still a complex process, therefore always requires the creation of some kind of support to facilitate this manipulation.

3.4 Conclusion

The research was undertaken in the CI/CD area to cope with the elimination of dependencies that the teams may have in the creation of build pipelines, and cases where this happens both at the industrial and academic levels, was described in this chapter.

It all starts with the implementation of CI/CD practices. If the implementation of these practices is done from the beginning of a company, there will be no problems related to the use of these practices and the lack of autonomy of the teams [Tay18, Hen14]. If these practices are implemented in a company with several years of life, problems can be found in the implementation, such as the existence of dependencies between teams that prevent them from moving at a rapid pace [Fer18]. In the latter case, some solutions were found to prevent this from happening.

There are several cases study where the solution that the companies implement is the use of CI/CD tools, similar to those studied and presented in subsection 3.3.1 [Tun18b, Tun18a]. The advantages of implementing this approach are mostly autonomy and freedom to build pipelines of their choice without the need to have a dedicated team. The disadvantages are, lack of abstraction that the teams have about the concepts of build pipelines, being a process of complex creating build pipelines, prone to errors, and with several implementation patterns that if they are misapplied can lead to a poorly implemented architecture, there being an increase of feedback loop.

With the approach of creating a library of build pipelines, unlike the previous one, we can already free the developers from the complexity and concepts related to build pipelines [Cor16, Cor]. In this way, the developers continue with certain freedom for the use of build pipelines, since there is always the restriction of only being able to use what is available in the library. In addition to these advantages, it is also possible to prevent duplication of code and to have a build pipelines implementation pattern. However, there is the disadvantage of this approach just working in a scenario where all build pipelines are similar and standardized.

Last but not least, there is the approach of creating DSLs to abstract the complexity from existing CI/CD tools [Fer18, Yu18, Mar18, DM18, JKSS17, Ste16, Exn15, Cor]. As an advantage of this approach, developers can focus on delivering value to the business, rather than focusing on problems with the low-level settings in the CI/CD tools, thereby increasing their productivity [Yu18]. As expected, this approach also has disadvantages, such as a major initial effort to create a DSL, development, and maintenance that may be difficult to do, and the fact that DSL might not be flexible enough to adopt new features.

Several approaches have been presented with their advantages and disadvantages. They will be addressed in more detail in the next chapter, which explains in detail why they are approaches that may or may not be effective in the context in which this dissertation is inserted.

Chapter 4

Proposed Solution

The previous chapter introduced the solutions implemented by several companies in the industry. In this chapter, in section 4.2, it is presented the way in which the teams responsible for maintaining build pipelines work, and the proposed solution to solve the problem of build pipelines manipulation complexity, which is defined in chapter 1, which leads to a lack of autonomy. In section 4.3, an analysis of these solutions is made in order to find a solution that solves the presented problem.

4.1 Introduction

The solution presented in this chapter focuses on the pipeline as code. However, due to the issues presented in chapter 1, we cannot focus only on this solution, since it does not fulfill the requirement that developers with low build pipelines knowledge can create or handle existing build pipelines. Due to this, it was necessary to find a solution that covered the whole problem presented.

As also mentioned in the previous chapter, there is no ideal solution. Each case is a case, so in the context of the company OutSystems and of the problem presented, the proposed solution is not one used by other companies, but a combination of two solutions described in chapter 3:

- **Creation or use of a DSL** - A solution where we create or use a DSL that allows to create build pipelines through code.
- **Creation of build pipelines libraries** - A solution where we create a series of libraries with build pipelines that are often used by developers, promoting reuse of code and pipeline definitions which is particularly relevant when having the multiple teams that work on the same product, and hence need to share and work on the same build pipelines.

The proposed solution is the creation of a DSL that allows the reuse of code, thus combining the solutions of creation libraries of build pipelines and the possibility of having pipeline as code. These two measures allow developers to create or manipulate build pipelines through a simple syntax by freeing them from all the complexity inherent in build pipeline configuration.

The purpose of this chapter is to present in more detail the solution to the problem that exists in dealing with complex pipelines by developers with low knowledge in this area.

4.2 Modus Operandi

During the course of this dissertation, the teams responsible for the manipulation of build pipelines changed the way they worked to tackle the issues they were facing.

The initial way to manipulate existing build pipelines in OutSystems R&D was to program them visually.

As already mentioned, the CI/CD system that OutSystems uses to validate its product is GoCD. This tool allows developers to create build pipelines visually in a simple way, but, as build pipelines complexity increases, creating or manipulating pipelines becomes a more complicated task. The fact that the OutSystems product needs complex build pipelines to validate it, makes this solution unfeasible for the teams that deal daily with build pipelines. Thanks to this, the need to find a new solution arose.

The solution found was to manipulate a configuration file that has all build pipelines visually defined in XML code. This file was manipulated in order to separate the various build pipelines into different files. Thus it was possible to handle complex build pipelines more easily when compared to visual manipulation. At the end of the modifications, the build pipelines separated into different files are re-manipulated to join them all together in a configuration file, applying it to the GoCD, where the changes would be applied immediately.

The workaround to program the build pipelines through code caused the problem of not allowing concurrent work in the same configuration file. Whenever different developers started working on their machine with the same version of the configuration file, the last one to add their version to the GoCD server would generate a conflict. Trying to solve this conflict sometimes led to the work being superimposed.

Due to this problem, other solutions began to be thought through. During this time of designing a new solution, the GoCD has released a new version with pipeline as code support developed in JSON or YAML.

With this new feature present in GoCD, the teams responsible for maintaining the build pipelines changed their way of working to use pipeline as code with JSON. Now the developers who manipulate the build pipelines have to develop their code in JSON, and then commit that code in a code repository that GoCD will interpret and apply the changes made. This new feature was able to solve the problems that the teams had in the development. However, although this is a good solution to be used by OutSystems, as we saw in the section 3.3.1, it is not enough, since the problem of complexity is not solved. Using GoCD, the handling of these build pipelines continues a complex process for a developer who does not have the knowledge of GoCD and the concepts of build pipelines.

The change of programming build pipelines in XML and visually for programming through pipeline as code, made that the possibility of reusing code disappeared. Visually, or through manipulation of the XML file, GoCD allows reusing code through templates. The templates allow reusing the entire structure of a pipeline together with the tasks. However, currently in pipeline as code this is not possible. This way of code reuse was not ideal because it did not have great granularity in the reuse, but it allowed reuse of the build pipelines structure, which now does not happen with the pipeline as code.

Therefore, from the moment it was decided that the team would begin to manipulate build

pipelines with the GoCD pipeline as code, it became known that the solution to be implemented in this dissertation would have to produce JSON GoCD pipelines as code.

4.3 Decisions

Analyzing the state of the art we can observe that the various solutions implemented in the industry can be organized in the following three topics:

- Creation of build pipeline libraries;
- The use of existing CI/CD tools;
- Creating or using a DSL.

Of these three topics, the advantages and disadvantages of each were individually analyzed.

Regarding the topic that leads to the creation of build pipelines libraries that developers can use as configuration as code, we have the benefits of releasing complexity, preventing duplicate code, and achieving a build pipelines implementation pattern. However, this solution is limited because it can only be applied in scenarios where there is a great standardization of build pipelines allowing for a good library to be enough for teams to be able to create their one build pipelines. This limitation prevents it from being a solution that can be applied to OutSystems, since more and more each team will need to have its build pipelines adjusted to their realities.

The other solution presented in the state of the art is to allow different teams to use any CI/CD tool that best suits them. This solution allows these teams to have the freedom and the desired autonomy. However, we would end up without any abstraction, with complex error-prone processes, with various implementation patterns, and possibly with poorly designed architecture that could give rise to a larger feedback loop. As already mentioned, a larger feedback loop can lead developers to always think twice before committing, contrary to our goal of accelerating customer value delivery. This solution would eventually leave the team responsible for maintaining the build pipelines without the danger of becoming a bottleneck, but we would be without concept abstraction, and with a sharp curve in the learning of these concepts. In addition, if this solution is applied in companies where multiple teams work on the same core product, such as OutSystems, this would not be possible, since teams that share the same assets would not be able to have validation processes in different CI/CD tools.

Finally, we have the topic of creating and using a DSL. As we saw in the state of the art, this is a solution already used by several companies, such as Pivotal, Amazon, Home Depot and Riot Games [DM18, Exn15, JKSS17, Ste16]. This solution brings the benefits of allowing to focus on delivering value to the company, rather than focusing on build pipeline configurations. In this way, we can increase productivity, the quality of build pipelines construction, and achieve a high level of abstraction. Despite these advantages, care must be taken with the implementation of a DSL so as not to make its evolution and maintenance complicated. We must also be careful about the expressiveness of the language itself in order to be able to be used to create different kinds of build pipelines, finding the right balance between level of abstraction and increase productivity and expressiveness.

With the analysis of these three topics, at first glance, the third topic is what most stands out as the topic that fulfills all the requirements and objectives for a possible solution, since the problems that it brings can be tackled from the beginning. However, having declarative DSL makes it difficult to reuse code at various levels of the build pipeline architecture. So the decision fell on a DSL that allows code reuse with granularity, to be able to reuse work made by other developers. This solution joins the last two topics, eliminating the problem of libraries being created by only one team, which could lead, again, to a possible bottleneck.

The combination of solutions leads to an exploration of an area that has not been explored so far: the term “pipeline as code” has been viewed as configurations made in JSON and YAML, and this proposed solution attempts to approximate the term “pipeline as code” to a more common programming language. With this approach, we can derive the benefits of reusing code and abstracting concepts, terms, and build pipelines configurations.

Since the proposed solution is a DSL, the decision was made to move forward with a visual or textual DSL. If the solution presented was a visually-defined DSL, the developers within OutSystems would make the adoption of this language more easily, since the OutSystems product is a visual language. Despite this, by adopting a visual language, it would make the implementation process more complex. As the objective of this dissertation is to make a proof of concept, it was decided to choose the simplest implementation, and if there is evidence that the productivity of the developer’s increases, in a future work move to a visual DSL in order to study the existing performance impact with development of build pipelines visually. Moreover, there is no expectation that there is a great difference in the complexity of development between a textual or visual language since the abstraction of concepts is present in both types of language, the only difference is the way that the developer program.

As mentioned in chapter 2, OutSystems R&D develops the core product by practicing single branch development. Since OutSystems needs to support different major versions of the product, they practice single branch development by major version, which causes them to have build pipelines for each major version. This makes the build pipelines for different major versions have several similarities, so it is important to have code reuse in the solution to be implemented.

A prototype of these two solutions used in the industry is expected to bring a simple and quick way to manipulate build pipelines across all teams, without the need to have a background on build pipelines. So, the learning curve of new team members to handle build pipelines is expected to be low.

4.3.1 DSL

A DSL, according to Fowler [Fow10], consists of a “programming language that contains limited expressiveness and focuses on a particular domain”. The opposite of a DSL is therefore a General Purpose Language (GPL) that has several capabilities able to solve several levels of problems of different domains [Fow10, HF10].

A DSL compared to a GPL best fits the requirements needed to solve the problem in question. DSLs can improve developer productivity, improve communication with domain experts, be eas-

ier to understand, less error-prone, and faster to write, and modify [Fow10]. However, despite having these benefits that point us to as an obvious solution to the problem we try to solve, there are problems that are associated with creating DSLs, and have already been felt by other companies as shown in chapter 3. Among these problems are the costs associated with the evolution and maintenance of DSL, the initial cost of building a DSL, and the fact that a DSL may not be flexible enough [Fow10].

With this definition of DSL, we conclude that the solution presented by this dissertation should be a DSL, since what we want to offer to developers is a new language that focuses on the manipulation of build pipelines. In addition, there are large companies, such as Pivotal and Amazon, who built DSLs to handle build pipelines that worked very well for them.

With this, there are several decisions that have to be taken in order to build a solution. One of them is the type of DSL to be developed, which there are two [Fow10]:

- Internal DSL;
- External DSL.

These next two subsections are largely based on Fowler's book about DSLs [Fow10].

4.3.2 Internal DSL

An internal DSL consists of a language that is represented within a GPL for a particular domain. This means that when we are developing a script of an internal DSL we are writing valid code of a GPL, but the features used are made in a particular style so that we can handle the domain in which we created the DSL .

In this type of DSL, the focus should not be limited expressivity since an internal DSL is a GPL, so the limited expressiveness here comes only from the way it is used.

Another point that defines an internal DSL is that the user to develop in this type should feel that the code they write is fluid and not just a set of independent commands.

4.3.3 External DSL

An external DSL allows to create a language that "is separate from the main programming languages" in which developers usually work. Here a new syntax, or a similar to what developers are costumed to work, can be used.

The threshold of building an external DSL is defined by losing focus on the domain for which we create the DSL and developing more than necessary, becoming a GPL.

With these two types of DSL available, the choice fell on an external DSL because of the flexibility to create a totally new syntax, in order to create something easily understood by the developers.

4.3.4 Configurations

The developed DSL has the designation of PACE (Pipeline As CodE), a word defined by "the speed at which one or something moves, or with which something happens or changes" [Cam]. The definition of this word refers to the overall objective of this dissertation in wanting to accelerate the development of teams, giving them autonomy to develop at their own pace. In addition to the name of the DSL, this word eventually became the extension of the files developed with the DSL, therefore ".pace".

Once we know the type of DSL to implement, it follows the part where we have to decide how the connection between PACE and GoCD will happen. At the outset we already know that the code to be generated by PACE must be JSON, however, it is necessary to decide if the concepts of the GoCD remain unchanged in PACE.

Regarding this decision, the main concepts that were decided to change were the concepts of Pipelines, Stages, and Tasks.

In GoCD the concept of Pipeline consists of one or more Stages, each Stage can have one or more Jobs, and each Job is made of one or more Tasks. Pipelines may or may not run in parallel depending on the workflow defined by the user. This workflow is defined because we can create complex build pipeline systems by interconnecting different Pipelines (downstream and upstream) with the ability to fan-out and fan-in. Stages within a Pipeline always run sequentially, in the order they are defined, and Jobs always run in parallel if there are agents available. Tasks within a Job run sequentially. In GoCD there is also the concept of environment, which is characterized as being a set of several Pipelines where the user can define environment variables, which can be used by all pipelines belonging to that environment. If the various Pipelines of the same environment are connected together, they are called pipeline systems. These concepts can be seen graphically in figure 4.1.

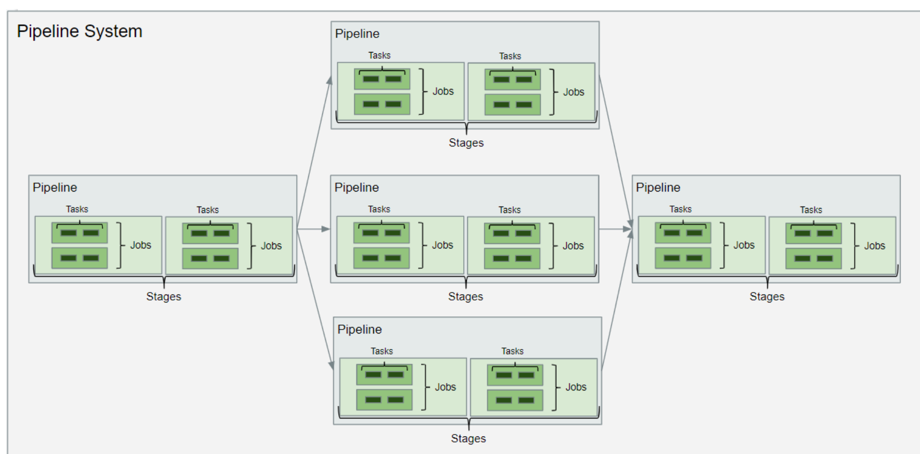


Figure 4.1: Pipeline system in GoCD.

PACE gives a different meaning to some of these concepts. The Pipeline concept consists of a set of steps that allows to execute our Task and is composed of one or more Stages. Normally a Stage offers a level of confidence about the product we are validating, and they may or may not run in parallel depending on the workflow defined by the user, and these are composed of

one or more Jobs. Jobs are defined as how we are going to achieve the goal, and always run in parallel if there are agents available. Each Job consists of only one Task, that will be called "run". This Task can have one or more actions, programmed with a well-known language by OutSystems developers, Python. These concepts can be seen graphically in figure 4.2.

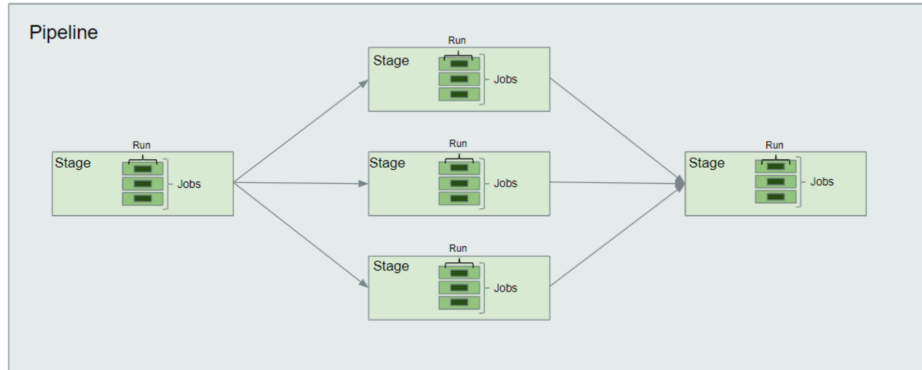


Figure 4.2: Pipeline in PACE.

In addition to these main concepts, a mapping of other concepts was also defined, which mostly served to reduce the complexity and facilitate the understanding by the developers. In the table 4.1, we can then see the concepts that are mapped from PACE to GoCD.

Table 4.1: PACE to GoCD Concepts.

PACE	GoCD
Pipeline	Environment with multiple pipelines
Stage	Pipeline with a single Stage
Job	Job
Run and Telemetry	Tasks
Variables	Parameters and Environment Variables
Discardable Agents Values and Agent Resources	Environment Variables
Extends, Override, Default Values, Functions, and Includes	(No direct mapping)

This difference between concepts means that at compile time, where code written with PACE syntax is transformed into JSON code, transformations have to be made so that PACE concepts are mapped to GoCD concepts.

The concepts Extends, Override, Default Values, Functions, and Includes are concepts where there is no mapping for GoCD concepts, because they are only used at compilation level. The reason for adding these concepts is to be possible to reuse code while implementing build pipelines in PACE.

Another decision that needed to be made was to know which configurations the developer would have the total freedom to change and those that would be hidden at compile time. This kind of decision had to be taken with some consideration. It was necessary to find a balance between the level of freedom that the developer has in choosing the configurations, and the complexity that these configurations bring to PACE.

The decision on the configurations fell on three points:

- Configurations that the user can define through PACE syntax, and they are mapped directly to the CI/CD tool;
- Configurations that PACE abstracts through an expression, which at compile time generates much of the code needed to be valid in the CI/CD tool;
- Configurations that are pre-defined that developers cannot change. These configurations belong to GoCD, and this does not prevent using the same syntax to generate code for other CI/CD tools since the configurations are assigned at the compilation level and not at the syntax level.

The configurations for each of these topics are now presented.

The configurations that the user can specify through PACE syntax, and which map directly to the CI/CD tool, are the configurations that are more likely to undergo changes when creating or modifying build pipelines, according to the studied in the industrial context of this dissertation. In addition to the code that the user needs to write to create a build pipeline, he can configure:

- **Timeout** - Maximum time a Job has without show any output.
- **Trigger type** - Can be automatic, manual or by a timer. This configuration allows the build pipeline to be triggered, depending on the trigger type chosen. The existing trigger types are: automatic, when a developer commits to a repository; manual, when the user needs to press a button for the build pipeline to start running; and a timer, that allows the user to schedule the triggering of the build pipeline through of a cron expression.
- **Location of the repositories checkout and the location of the artifacts** - PACE allows the user to define the folder to checkout the repositories, and also allows defining the location of where artifacts will be stored.
- **Working directory for the Job** - The code written in Python belonging to the Job can be executed in any directory, so the user has the freedom to choose where it will run.
- **Artifact type** - Artifacts generated by the Job can be assigned two types: test or build. The difference between these two types of artifacts is that if the artifact is of test type, the GoCD will try to interpret its content, in order to show test failures.

Due to the need to give more configuration options to the user without increasing the level of complexity, we chose to generate compile-time code in order to get configurations from what the user writes. These configurations include:

- **Telemetry** - This is a configuration existing in the context of this dissertation and was inserted by the Engineering Productivity Group, not being a configuration belonging to the CI/CD tool. We detected that several tasks were always present in most build pipelines, so we decided to abstract this into an expression, giving the option to developers to use or not the telemetry, which allows performance metrics to be saved. With the abstraction of this concept, it is possible to increase productivity without increasing the complexity of PACE. This configuration is an example of PACE being able to be extendable, showing that it is possible to transform a common task in different build pipelines in a task supported by the language.

- **Discardable agents** - Similar to the previous case, the concept of discardable agents was inserted by the Engineering Productivity Group, not being a configuration belonging to the CI/CD tool. This mechanism allows having a cost control mechanism on the machines responsible for running the Jobs of the build pipelines. Because this is a concept in which the developer should not worry, the PACE language abstracts this mechanism, generating in compile time what is necessary to have this mechanism implemented.

- **Repositories that do not trigger the build pipeline when a commit is done** - There is sometimes a need to use materials from which the user do not want to trigger the build pipeline when a commit is made. In this way, the user will have the possibility to specify these repositories in a simple way, and at compile time filters are added to the repositories configuration so that the build pipeline is not triggered.

- **Tasks** - Allows defining the code that will run in a Job. Here, the GoCD receives a set of tasks, which end up being commands that can be executed in the command line. In order to give the user consistency in the definition of the tasks, it was decided that in PACE the tasks become a single call to a Python file that contains all the code that the user defines. So the user defines the code in Python with PACE without worrying about any configurations that GoCD needs to run the tasks, and at compile time a Python file is generated which will be called by GoCD with the necessary configurations.

- **Variables** - In GoCD there are two types of variables: the environment variables; and the parameters. In order to reduce the complexity for the user, these two concepts were unified in the concept of variables present in PACE. Thus, the user does not have to worry about using environment variables or parameters, as this choice will be determined at compile time. Due to the fact that PACE is being developed in the industrial context of OutSystems, it is necessary that PACE be compatible with the existing scripts that validate the product. Therefore, almost all variables are transformed into environment variables in the GoCD so that the existing Python scripts can access the variables. The variables that are used in the artifacts are the only ones that are transformed into parameters.

- **Environment** - The environment where the pipeline we are programming will belong is another configuration that GoCD offers and that the user does not have to worry about. In PACE the build pipeline is seen by GoCD as a set of Pipelines, so it was decided that this set of Pipelines will be a GoCD environment with the same name as the Pipeline. In PACE, all variables defined at the Pipeline level are then defined in the GoCD as environment variables. With this, we are reducing the need for developers to know more about a concept without removing any expressiveness and flexibility. We are also standardizing how to use GoCD concepts making it easier to understand and read final GoCD settings, that is always required for example in case of troubleshooting problems and follow up pipelines execution.

- **Resources** - In GoCD and others CI/CD tools, we need to tag agents with a property that will be used to assign a Job to these agents. In PACE, these resources are inferred through a set of data explained in more detail in section 5.3, thus abstracting another concept of the developers.

Finally, there are the GoCD configurations that the user has no control and which are pre-defined in the compiler. Because the Engineering Productivity Group always uses the same configurations

for the build pipelines that currently exist, it was decided that these configurations could be abstracted in PACE, and be generated at compile time. These configurations are:

- **Pipeline locking** - The pipeline locking configuration serves to ensure that only a single instance of the pipeline runs at a time, that is, if a pipeline is locked then another instance will not be scheduled until the one that is running completes its tasks. This configuration can have three types of values:
  - Unlock when finished - Only one instance of the build pipeline runs each time, being unlocked when it is finished. PACE uses this configuration.
  - Lock on failure - Only one instance of the build pipeline runs each time, and remains locked if it fails.
  - None - Multiple build pipeline instances can run concurrently.
- **Clean working directory** - This configuration allows removing all files and directories from the agent's working directory. PACE sets this configuration to "true".
- **Never cleanup artifacts** - This configuration causes artifacts never to be purged if purging artifacts is configured at the server level. PACE sets this configuration to "false".

Due to the existing requirement that PACE needs to support code reuse, a need arose for a new decision to be made. Should we continue to reuse the code offered by GoCD through the templates, or should we arrange a new form of code reuse for PACE? Once the GoCD started to support pipeline as code through JSON and YAML, and since GoCD's current pipeline as code solution does not support the definition of templates, it was decided to support a new form of reuse for PACE, leaving aside the concept of templates offered by GoCD. This new way to do code reuse allowed to have more granularity in reuse in the structure of the build pipeline when compared to the templates, therefore, more flexibility is given to developers when using PACE, than when using GoCD JSON or even XML.

The decisions made about these configurations, and the concepts to have in PACE, had to be weighted in a way to balance the freedom given to the user and the complexity that this freedom introduced to PACE. Although we implemented this to compile to GoCD pipeline as code, all decisions were made pondering that any concepts would possibly have to be compiled to another language to be used by other CI/CD tools. For example, if there is a need to move to another CI/CD tool, we should not need to make major changes to PACE syntax, so that PACE code should not be useless. Another example is that if GoCD offers new features, it should be easy to change PACE compiler to take advantage of these new features.

If these configurations are implemented as described, we will be able to hide some configurations and concepts from the users, reducing the learning curve in pipeline manipulation. The ease in learning makes it possible to handle the build pipelines in a simple and quick way, solving the problem described in chapter 1.

4.4 Conclusion

The proposed solution and the requirements that we propose to tackle, in order to solve the problem related to the complexity associated with the manipulation of build pipelines is pre-

sented in this chapter. This problem can lead to a lack of autonomy by the teams in the manipulation of build pipelines, which may later lead to a decrease in the speed of delivery of value to the customer.

The proposed solution, when compared to the current way of manipulating build pipelines, and when compared to what was presented in chapter 3, reduces the amount of configurations and concepts related to build pipelines. With this, it is possible that PACE can reduce the learning curve so that the time and difficulty in learning the concepts are considerably lower than existing alternatives. This proposal combines the techniques of use of libraries and use of a DSL, studied in chapter 3, so that the objective is reached.

The next chapter will outline the tools used to implement PACE, the existing details, and decisions made during its implementation.

Chapter 5

Solution Implementation

This chapter discusses how the solution was implemented. In section 5.2 the tools chosen for implementation are discussed. In section 5.3, the implementation model is presented together with explanations of the evolution of PACE. In section 5.4 the final solution structure is presented, in section 5.5, the implementation details of the solution proposed in the previous chapter are presented, and in section 5.6 a brief comparison is made between how to create build pipeline with JSON and with PACE.

5.1 Introduction

The solution's implementation goal is to demonstrate that the designed solution can in practice facilitate the manipulation of build pipelines by developers that have little to no context of such operations.

The built prototype is intended to demonstrate that the use of pipeline as a code that allows to abstract details about the underlying technology and reuse of code is a valid option to follow if we are trying to make different autonomous teams. As this is a prototype, some parts of the proposed solution were not fully implemented, since they, predictably, would not impact the conclusions. This leaves some space for future improvements and they will be presented in chapter 7.

As already explained in chapter 4, the solution design was based on the context of OutSystems, so the implementation was thought along the same lines. However, the core components of the implementation can be easily adapted to a more generic industrial context.

5.2 Toolset

In the implementation of PACE, two tools emerged that could help on its implementation.

The first tool was ANTLR[1], that allows building compilers, translators, DSLs, among other things. Developers usually choose this tool because it can reduce time and effort in building and maintaining the language [ANT12].

The second tool was Xtext[2], developed and designed to implement DSLs, also reducing the effort in the development, similar to the ANTLR. This is a tool maintained by the Eclipse Foundation[3],

[1] https://www.antlr.org/
[2] https://www.eclipse.org/Xtext/
[3] https://www.eclipse.org/

and any DSL programmed with this tool can be interpreted by an editor that supports Language Server Protocol (LPS) or through a web browser [Xtec].

Because of the problem presented in section 4.3 related to the impact of creating and manipulating a DSL, we chose to use the Xtext tool because it facilitates the manipulation of DSLs, and the OutSystem's internal knowledge on how how to use this tool, thus reducing the impact on creating and maintaining the DSL [Sé16]. Furthermore, because the DSL created can be integrated by multiple IDEs, it does not force the developers to work in a different editor than the one used on a day-to-day basis.

Xtext allows defining, among other things, the grammar, the code to be generated at compile time, and custom validation rules. The grammar is done as shown in figure 5.1 provided by Xtext tutorial [Xteb]. In this figure, a language is defined where its syntax is a set of 0 or more lines, which contains a greeting, such as "Hello reader!".

```
Model:
    greetings+=Greeting*
;

Greeting:
    'Hello' name=ID '!'
;
```

Figure 5.1: Grammar example.

The code to be generated is defined through a language called Xtend, which is part of the set of tools that Xtext provides. Xtend is described as being "a flexible and expressive Java dialect" that compiles to Java code [Xtea]. One of its features that facilitates the generation of code is template strings, as seen in figure 5.2.

```
def compile(Model p)'''
    People to greet:
    «FOR s:p.greetings»
        «s.name»
    «ENDFOR»
'''
```

Figure 5.2: Example of a function written in Xtend.

Figure 5.2 represents code written in Xtend, where the compile function will generate a single output of type String. The work of this function will be the generation of a string with the name of all the people that have been defined in the DSL with the grammar corresponding to figure 5.1.

As already mentioned in chapter 4, the code generated by PACE is valid JSON code for GoCD. GoCD is an open-source CI/CD tool that currently supports pipeline as code that can be defined in two languages, JSON[4] or YAML[5]. Pipelines are represented by files that are stored on a git repository that, once loaded, can be used by GoCD's runtime logic. Thus the code will be under version control giving rise to the created build pipeline.

[4]https://github.com/tomzo/gocd-json-config-plugin
[5]https://github.com/tomzo/gocd-yaml-config-plugin

5.3 Implementation Decisions

PACE was developed in an incremental model, constantly obtaining feedback from the target users - the OutSystems R&D developers. This allowed PACE to undergo major changes over time, which led to the final result presented in section 5.4. However, there were some insights from each iteration that influenced the style of the existing syntax. This section documents these findings.

Initially, we started with 3 DSL options for the developer to choose parts that they most liked, in order to improve the syntax to create the final DSL. In addition to the structural decisions presented in the course of this section, there was also an evolution in the keywords to be used, so developers could easily understand the concepts. This evolution of keywords will not be addressed here in detail since it did not have a great impact on the programming of the compiler.

An example of keyword evolution is the case of the artifacts keywords. Initially this concept was defined as “outputToServer” and “inputToServer”. However, the feedback obtained said that it was not clear what could or could not be saved, so it was decided to change to “save” and “load”. Again, these keywords were not well accepted for similar reasons, which led to a search for what was commonly used in the industry. From this research, it was concluded that “artifacts” and “fetchArtifacts” should be chosen so that in case of doubt about what this concept means, developers can easily find an explanation not only in DSL documentation but also online.

Although decisions have been made in several iterations, they will be presented on topics that represent the main purpose of validation.

Validation goal: Concepts Abstraction
We began by designing PACE with a very high level of abstraction. However, this also eliminated the possibility for developers to configure or customize certain parts of the build pipeline, leaving no flexibility when creating new build pipelines. With this, it was decided to implement the language from a lower level of abstraction, and rising it until the ideal balance between the flexibility of configurations and the abstraction level that reduced the language complexity was found.

When we started to design PACE through a low level of abstraction we received feedback that what was designed did not diverge from the GoCD concepts, being only a translation of what exists in the JSON pipeline as an easier-to-read syntax. At this point, there was practically no abstraction that would allow code to be reused efficiently since we continued to use the template definition present in the GoCD.

With this feedback, we began to abstract the language from the GoCD concepts so that PACE syntax could be adapted to other CI/CD tools. Due to the industrial context where this dissertation is inserted, it was decided that the prototype would only generate code for GoCD, however, it was defined as a requirement that PACE should have a syntax that would easily generate code for any other CI/CD tool. In this way, we can create a solution that allows us to differentiate from those that currently exist in the market.

A concept that was decided to abstract from PACE, thanks to the feedback received, was the

concept of environment variables and parameters that exist in GoCD. The developer found difficult to understand the difference between these two concepts, so it was decided to incorporate everything into a single concept of variables that in compile-time are transformed into environment variables or parameters. The variable concept is transformed into parameters when they are used in artifacts, and in environment variables in the remaining cases to be compatible with existing Python scripts that validate the OutSystems product. These Python scripts where, themselves, a way for the Engineering Productivity Group to abstract certain tasks. However, this abstraction only happens at the level of the task language and not at the level of build pipeline architecture. The abstraction of these two concepts that are closely connected to the GoCD tool to just one concept, allowed the DSL syntax to be easier to map to another CI/CD tool than GoCD. So, in addition to abstracting developer concepts, we are also making PACE easily mappable to other CI/CD tools.

Of the various concepts where we look for creating new abstractions, there were two that stood out: the agents responsible for running the tasks; and the build pipeline telemetry.

In GoCD and in other systems that allow the creation of build pipelines, there is the concept of agents, where each of these agents may have a set of labels or resources associated. The fact that agents have these associated resources makes it possible to specify that a Job will run on an agent that satisfies a set of resources. One feedback from the first design was that developers do not need to know and define these configurations. So, the concept of resources was removed entirely from PACE, and the necessary configurations are automatically generated at compile time, assigning resources to the corresponding Jobs, following an algorithm represented in figure 5.3.

```
if 1 job per stage:
        Type_of_System, Type_of_Machine + Name_of_Pipeline
else:
        Type_of_System, Type_of_Machine + Name_of_Pipeline, Name_of_Job
```

Figure 5.3: Resource assignment to Jobs.

What happens in this assignment is that at compile time if there is only one Job per Stage, the resources assigned to a Job will be the system type (SS or PS), and the name of the machine type concatenated with the name of the build pipeline. In case there is more than one Job per Stage we have to allocate one more resource in addition to those already explained. This additional resource will be the name of the Job so that they are not the same machines running in different Jobs, on the same Stage.

Telemetry was another concept that was decided to abstract in PACE. Telemetry is important because with it we can collect metrics related to the build pipeline, for example, execution times, stages status and others. Without this abstraction in PACE, the developers would have to add specific and overly complicated tasks before and after the tasks that would do the real Job work. What was decided to do was to add a flag that at compile time will generate the necessary code to have telemetry implemented in the build pipeline. As already explained in chapter 4, this is not a build pipeline architecture abstraction but an abstraction at the level of the automation job's logic, since this concept is not part of the underlying CI/CD tool. With

the abstraction of this concept, it is possible to reduce the complexity of PACE. In PACE the telemetry is active by default, so the user does not need to add anything. In the GoCD JSON, for the developer to have the telemetry it is necessary to add the code represented in the listening 5.1. If developers do not want telemetry in the build pipeline, in PACE they need to add the line of code represented in the figure 5.4, while in the GoCD JSON they do not need to add anything.

```
"tasks": [
 {
  "command": "cmd.exe",
  "arguments": [
   "/c",
   "if␣exist␣platform\\QA\\ContinuousIntegration␣rmdir␣platform\\QA\\ContinuousIntegration␣/s␣/q"
  ],
  "run_if": "passed",
  "type": "exec"
 },
 {
  "command": "aws",
  "arguments": [
   "s3",
   "cp",
   "s3://%GO_ARTIFACTS_S3_BUCKET%/CIScripts/ContinuousIntegration.7z",
   "platform/QA/ContinuousIntegration.7z",
   "--only-show-errors"
  ],
  "run_if": "passed",
  "type": "exec"
 },
 {
  "command": "7z",
  "arguments": [
   "x",
   "platform\\QA\\ContinuousIntegration.7z",
   "-oplatform\\QA\\ContinuousIntegration"
  ],
  "run_if": "passed",
  "type": "exec"
 },
 {
  "command": "cmd.exe",
  "arguments": [
   "/C",
   "del",
   "/Q",
   "platform\\QA\\ContinuousIntegration.7z"
  ],
  "run_if": "passed",
  "type": "exec"
 },
 {
  "command": "python",
  "working_directory": "platform/QA/ContinuousIntegration",
  "arguments": [
   "update_pip_requirements.py"
  ],
  "run_if": "passed",
  "type": "exec"
 },
 {
  "command": "python",
  "working_directory": "platform/QA/ContinuousIntegration",
  "arguments": [
   "print_ci_scripts_info.py"
  ],
  "run_if": "passed",
  "type": "exec"
 },

/*
In the space of this comment is where the tasks that the job will execute are specified.
*/

 ,{
  "command": "python",
  "working_directory":  "s3_download_keys",
  "arguments": [
  "task2_Green_PS_Trunk_PACE_Green.py"
  ],
  "run_if": "passed",
  "type": "exec"
 }
 ,{
  "command": "python",
  "working_directory": "platform/QA/ContinuousIntegration",
  "arguments": [
   "telemetry_job_end.py",
   "--pipeline_name",
   "%GO_PIPELINE_NAME%",
   "--pipeline_counter",
```

```
    "%GO_PIPELINE_COUNTER%",
    "--stage_name",
    "%GO_STAGE_NAME%",
    "--stage_counter",
    "%GO_STAGE_COUNTER%",
    "--job_name",
    "%GO_JOB_NAME%",
    "--branch",
    "%BRANCH_NAME%",
    "--system",
    "%CI_SYSTEM%",
    "--status",
    "Passed"
   ],
   "run_if": "passed",
   "type": "exec"
  }
  ,{
   "command": "python",
   "working_directory": "platform/QA/ContinuousIntegration",
   "arguments": [
    "telemetry_job_end.py",
    "--pipeline_name",
    "%GO_PIPELINE_NAME%",
    "--pipeline_counter",
    "%GO_PIPELINE_COUNTER%",
    "--stage_name",
    "%GO_STAGE_NAME%",
    "--stage_counter",
    "%GO_STAGE_COUNTER%",
    "--job_name",
    "%GO_JOB_NAME%",
    "--branch",
    "%BRANCH_NAME%",
    "--system",
    "%CI_SYSTEM%",
    "--status",
    "Failed"
   ],
   "run_if": "failed",
   "type": "exec"
         }
]
```

Listing 5.1: Tasks needed to implement telemetry in GoCD JSON.

```
telemetry="false"
```

Figure 5.4: PACE telemetry.

Validation goal: Syntax Structure

At the beginning, the tasks were shell commands to run sequentially, similar to what GoCD offers [GoCc]. In this way, there would still be a technological problem because developers would have to know how to handle shell, the DSL, and Python in order to manipulate build pipelines, and this does not reduce the learning curve in build pipeline manipulation. With this, it was decided that the language to define a Task would be Python, without the need to explicitly state so. As a result, everything written within the concept of “run” is considered Python code, as we can see in the figure 5.5.

```
run:
    '''
    f = open("cintia.txt", "r")
    f1 = f.readlines()
    for x in f1:
        print(x)
    '''
```

Figure 5.5: Example of run.

The opinions between choosing a DSL with the structure that allowed the creation of code blocks with brackets, or through the indentation were very divided. With the decision of the task language being Python, it followed the decision for a coherent structure definition through indentation.

PACE - Pipeline as CodE

In relation to the PACE structure, the feedback was still obtained regarding the difficulty in looking at the PACE file and make a mental model of the constructed build pipeline. This feedback was obtained because initially the dependencies between Stages were defined at the beginning of the file, and this meant that when there were complex build pipelines there was a huge list of dependencies that would be difficult to interpret with a quick look through the file. This led to the PACE structure being rethought once again.

Repetition of keywords on multiple lines, like “trigger”, was also pointed out by feedback, stating that it caused too much noise. To solve this, code blocks were added in places where this repetition happened, simplifying their usability and readability.

Validation goal: Code Reuse

GoCD and other CI/CD tools allow reusing code through the use of templates. However, when they where implemented in PACE, we realized that they did not have the granularity needed to empower PACE with the desired level of code reusability. GoCD templates are only applied to Pipelines, being impossible to reuse, for example, Jobs or Stages.

It was then decided to make code reuse through a function call. This solution was not very well accepted because it was necessary to pass a large set of arguments and this way it is difficult to understand the correct order of these arguments. It was also pointed out that variables could be reused as well. With the possibility of reusing variables, it is possible to have a file with a set of variables, which can be seen as a configuration file, which can be included in the file that we are writing. With this solution implemented, most of the arguments to be passed between functions were reduced, making it easier to reuse code and allowing to be able to do code reuse through a function call.

One of the feedbacks suggested having several levels of abstraction so that different people with different levels of expertise can manipulate build pipelines. As a result of this feedback, the concept of inheritance was implemented in PACE, a common concept in object-oriented programming. So a developer with more expertise can determine the level of abstraction that a developer with a lower level of expertise will have.

When all these evolution’s stabilized, it was necessary to create documentation for PACE, to make easier to show all the features that it has. This documentation later served as support for the experimental validation presented in chapter 6.

For a better understanding of the evolution that PACE has undergone from the first design to the one that was used to make the experimental evaluation, we can compare figure 5.6 that represents the first design of PACE, and the figures 5.7 and 5.8 the current prototype of PACE.

```
include telemetry

pipelineGroup = "group_pipeline_name"

def pipeline "name_of_pipeline":
    dependencies :
        svn "repo_link" to "checkout_location"
        git "repo_link" to "checkout_location"
        pipeline pipeline_name

    resources :
        resource_name
        resource_name

    stage stage_name = [job_name1, job_name2]


def job job_name1 runAllMachines :
    inputFromServer :
        artifact_location/artifact_name from pipeline_name stage stage_name job job_name

    - python create_test_run_v2.py --branch Trunk at "location_to_run_the_command"
    - python stage_run_v2.py
    - python end_test_run_v2.py --branch Trunk

    outputToServer :
        artifact_location/artifact_name
```

Figure 5.6: First PACE design.

```
import import_name

include "file_name.pace"

telemetry=false //True by default, generating telemetry for the pipeline.

var this_is_a_var="var value"

pipeline pipeline_name:
    repositories:
        trigger:
            svn "repo_link" to "checkout_location" login "username":"password"
        git "repo_link" to "checkout_location"

    stage stage_name:
        discardableAgents:
            quantity=7
            type="type_of_agent"

        job job_name:
            fetch:
                "artifact_location/artifact_name" from stage "stage_name" job "job_name"

            run at "location to run the python code":
                '''
                f = open("cintia.txt", "r")
                f1 = f.readlines()
                for x in f1:
                    print(x)
                '''

            artifacts:
                "artifact_location/artifact_name"
```

Figure 5.7: Current PACE design.

```
def stage function_of_type_stage:
    discardableAgents:
        quantity=7
        type="type_of_agent"

    job job_name:
        fetch:
            "artifact_location/artifact_name" from stage "stage_name" job "job_name"

        run:
            '''
            python_code
            '''

        artifacts:
            "artifact_location/artifact_name"

def job function_of_type_job(var arg1, var arg2):
    fetch:
        "artifact_location/artifact_name" from stage "stage_name" job "job_name"

    run:
        '''
        python_code
        '''

    artifacts:
        "artifact_location/artifact_name"

def function_of_type_python:
    '''
        python_code
    '''
```

Figure 5.8: Current PACE design.

5.4 Final DSL Structure

Now having an idea of how the final syntax structure is after all the iterations presented, we can look in more detail at each element of this structure.

Import

The concept of import was added to the language because of the task language be Python. In a file written with PACE the developer can have several runs where they can have imports in common. So that the user does not have to define these common imports in each Run, if the import is declared at the beginning of the script, it will be added to all Runs defined in our script. However, if "runs" need to have specific imports that are not shared, these imports should be made inside the Run along with the Python code. Therefore, all possible constructs for these imports are the same as those for Python, as we can see in figure 5.9.

```
import os
from os import path
from os import *
import os as a
```

Figure 5.9: PACE import.

Include

Include is a concept that distinguishes PACE from the pipeline as code offered by GoCD, and of the other DSLs and tools presented in chapter 3. Through include, developers can reuse code that is written in other files. In these files variables and functions can be defined, that can later be used in various build pipeline.

An example of how to do an include in PACE is presented in figure 5.10.

```
include "codeReuse.pace"
```

Figure 5.10: PACE include.

Telemetry
Telemetry is a configuration that allows developers to simply and seamlessly add metric performance to the build pipeline. By default the metrics are active since they are important for measuring the feedback loop. To create a pipeline without having metrics, the telemetry variable must be set to false, as shown in figure 5.11.

```
telemetry="false"
```

Figure 5.11: PACE telemetry.

Pipeline
In the concept of pipeline is where developer define the set of steps that they need for the product to be validated.

To define a pipeline developers open a block of code using the indentation and next to the keyword “pipeline” they write the identifier of the pipeline, as we can see in figure 5.12.

```
pipeline FirstPipeline:
```

Figure 5.12: PACE pipeline definition.

Within the pipeline code block, there are two more concepts, Repositories and Stage.

Repositories
Currently, PACE supports code from two repositories, SVN, and git. In this concept is where developers can define the repositories that contain code that they will need to use within the pipeline. This code can be divided into two types:

- Code that triggers the pipeline when a commit is done in the specified repository:
  - In this case developers open a block of code with the keyword “trigger” and inside they define the repositories specifying the type of repository, the Uniform Resource Locator (URL) of the repository, and the folder where they will want to checkout. If developers have to use an SVN repository, the login is done using the keyword “login” followed by the username and password separated by “:”;
  - Since this is the code block that will trigger the pipeline, developers can specify how they want this trigger to happen. The trigger can happen automatically, that is, whenever a commit is made in the repository, or it can also be manual, or by a timer. In case developers want a manual trigger, in front of the keyword “trigger”, they add the keyword “manual”, as we can see in the figure 5.13. If it is a triggered by a timer, in front of the keyword “trigger”, developers add the keyword “cron” followed by a cron expression, as the figure 5.14 represents. Finally, if developers want an automatic trigger, they do not need to add anything.

- Code that developer will need to use in the pipeline, but that they do not want to trigger the pipeline:
  - Here the developers just need to write within the block of code “repositories” the definition of repositories as explained previously. The definition of the repository where developer does not want to be triggered can be seen in figure 5.15.

```
pipeline FirstPipeline:
    repositories:
        trigger manual:
            git "https://github.com/nelsu/DSL_Scripts" to "code"
            svn "https://srvptsvn.***.***.com/svn/rd/platform" to "platform/HubServer" login "username":"password"
```

Figure 5.13: PACE repositories definition with manual trigger.

```
pipeline FirstPipeline:
    repositories:
        trigger cron "0 * * ? * *":
            git "https://github.com/nelsu/DSL_Scripts" to "code"
            svn "https://srvptsvn.***.***.com/svn/rd/platform" to "platform/HubServer" login "username":"password"
```

Figure 5.14: PACE repositories definition with schedule trigger.

```
pipeline FirstPipeline:
    repositories:
        trigger:
            git "https://github.com/nelsu/DSL_Scripts" to "code"
            svn "https://srvptsvn.***.***.com/svn/rd/platform" to "platform/HubServer" login "username:password"
        svn "https://srvptsvn.***.***.com/svn/rd/QA" to "platform/QA" login "username:password" /*This repository
                                                                                                 will never trigger
                                                                                                 the pipeline.*/
```

Figure 5.15: PACE repositories definition with automatic trigger, and with a repository not being triggered.

Stage

The Stage concept gives the user a level of confidence about the product that the developers are validating. This concept can be defined within the pipeline multiple times, and they may or may not run in parallel depending on the workflow that the user defines.

To define a Stage developers open a block of code using the indentation and next to the keyword “stage” they write the identifier, as we can see in figure 5.16.

```
pipeline FirstPipeline:
    repositories:
        trigger:
            git "https://github.com/nelsu/DSL_Scripts" to "code"
            svn "https://srvptsvn.***.***.com/svn/rd/platform" to "platform/HubServer" login "username":"password"
        svn "https://srvptsvn.***.***.com/svn/rd/QA" to "platform/QA" login "username":"password"

    stage FirstStage:
```

Figure 5.16: PACE Stage definition.

If a Stage needs to depend on another Stage, developers can add the keyword “dependsOn” next to the Stage identifier followed by the Stage identifier from which this new Stage will depend. Multiple dependencies is possible by adding “&” between the Stages’ identifiers. Here we can see an example where we can only have dependencies of stages that are defined, thus taking advantage of a language with syntax and semantics that allow having validations of this type. An example of a Stage that depends on three other Stages can be found in figure 5.17.

```
stage FifthStage dependsOn FirstStage & SecondStage & ThirdStage:
```

Figure 5.17: PACE Stage dependencies definition.

As stated previously, PACE supports inheritance of Stages and Jobs, meaning that developers can create generic pieces of code as functions of these types, being able to do extends or override. The explanation of how to do this reuse can be found at the end of this section in “code reuse”.

Within the Stage code block there are two more concepts, the concept of Discardable Agents and Job.

Discardable Agents

Discardable Agents were implemented in the GoCD by the Engineering Productivity Group due to three points:

- Long lived test environments need cleanup policies;
- Long lived test environment accumulate garbage that can adversely impact tests;
- New agent requirements need considerable effort to rollout new agents for an entire pipeline.

This implementation was done in GoCD using the specification of three environment variables:

- Agent Quantity - Number of agents to use;
- Agent Type - Type of agent. The value of this variable represents the type of agent that developer want to use, which must be present in the Cloud Formation template;
- Agent Resources - Resources to use when the agent auto registers in GoCD server.

With these three environment variables defined, a separate pipeline will run at the start of the day to create the agents needed for the pipelines to run. At the end of the day, another pipeline will run that will eliminate agents that have no pending work. For time reasons, the pipeline responsible for creating and destroying agents is not being automatically generated at compile time. However, in the future these pipelines should be generated as part of this concept, abstracting details related to build pipeline infrastructure management.

This concept was passed to PACE as a block of code which is optional, and when used needs to be defined two variables:

- Quantity - The value of this variable represents the number of agents that developer want to use;
- Type - The value of this variable represents the type of agent that developer want to use, which must be present in the Cloud Formation template. Currently this is the way to have discardable agents implemented by the Engineering Productivity Group, however as it is work in progress, no other way of abstraction of this concept was thought.

The third variable, agent resources, that the agent creation pipeline needs to have specified, does not need to be specified by the developers because a value is automatically generated based on the system type, machine type, Pipeline name, and Job name. Figure 5.18 shows the

discardable agents defined in PACE.

```
stage FirstStage:
    discardableAgents:
        quantity=1
        type="CIA_Platform"
```

Figure 5.18: PACE discardable agents definition.

Job

Still within the block of Stage code, the concept of Job is defined. The concept of a Job is one that is mandatory and it is where developers define how they are going to achieve the stage's goal. One or more Jobs can be defined, and they will always run in parallel if there are machines available.

To define a Job, inside a Stage the developers open a block of code, where in the header they write the keyword "job" followed by the Job identifier, as shown in figure 5.19.

```
stage FirstStage:
    discardableAgents:
        quantity=1
        type="CIA_Platform"
    job FirstJob:
```

Figure 5.19: PACE Job definition.

As previously mentioned, in similarity to the definition of Stages, here developers can also inherit code being able to do extends or override. The explanation of how to do this reuse can be found at the end of this section in "code reuse".

At the head of the Job definition, developers can also optionally define the maximum time that the Job will be active before being canceled. This definition can be specified ahead of the Job identifier through the keyword "timeout" followed by the maximum period of time (in minutes) that is expected to generate an output, as represented in figure 5.20.

```
job FirstJob timeout 40:
```

Figure 5.20: PACE Job timeout definition.

Within the Job code block, can be declared the last three concepts present in PACE, the concept of Fetch, Run, and Artifacts.

Fetch

Fetch is an optional concept and is used when developers want to define a set of artifacts to be fetched and used from an ancestor Job. This block of code is opened using the keyword "fetch", and within this block of code, developers can define the artifacts by defining their location, followed by the Stage and Job source location. As an option, developers can define the location where they want to store the artifact, as shown in figure 5.21.

```
fetch:
    "my-artifact.html" from stage FirstStage job FirstJob to "fetched"
```

Figure 5.21: PACE fetch artifacts definition.

Run

The block of code Run is a mandatory block and is where developers define the code that will run the tasks/validations that they want. This block of code is opened by using the keyword "run", and inside this block of code is where developers write the Python code that they want to execute. Ahead of the keyword "run", developers can optionally specify the directory where they want the code to run. In figure 5.22, we can see an example of a run.

```
run:
    '''
    print "Hello Guys!!"
    ##showMetrics(#artifact)
    '''
```

Figure 5.22: PACE Run definition.

In figure 5.22, we can notice that there is a line with the content "##showMetrics(#artifacts)". In addition to Python code within the code block run, we can reuse Python-type functions that can be defined in the PACE script. Further explanation on how to do this reuse can be found at the end of this section in "code reuse". The use of two "#" means that developers are calling a function of Python type, already mentioned previously, that can be declared in the same or another script. Using a single "#" means that developers are accessing a value of a variable that is declared in the same, or another file. Variables are declared using the keyword "var" followed by the identifier of the variable. The value assigned to variables doesn't have a type so developers always pass the values as Strings. In figure 5.23 we can see the definition of a variable.

```
var destination="out.py"
```

Figure 5.23: PACE variable definition.

Artifacts

The last concept within the Job code block is the artifacts, and this is another optional block of code. Here is where developers can define the artifacts they will want to save and then later be fetched. This block of code is opened by using the keyword "artifacts", and within this block they choose the artifacts to be saved. The artifacts can have two types:

- Build - Developers just have to define the location of the artifact, and optionally they can define the location where they want to save it.

- Test - An artifact of this type means that GoCD will attempt to interpret the artifacts generated [GoCa]. When developers want an artifact of type test, the only difference in relation to the artifact of type build is that before the definition of the location of the artifact they add the keyword "test".

In figure 5.24 we can see an example of a block of code of type artifacts.

```
artifacts:
    test "code/test.html" to "TestResults"
    "code/my-artifact.html" to "Results"
```

Figure 5.24: PACE save artifacts definition.

Code Reuse

As already mentioned throughout this section and throughout the document, it is possible to have code reuse. Here is now explained how this reuse of code can be done in PACE.

As stated previously, PACE supports inheritance of Stages and Jobs, meaning that developers can create generic pieces of code as functions of corresponding types, as shown in figures 5.25 and 5.26, that they can then reuse on any pipeline when defining Stages or Jobs.

```
//Definition of a Stage function type
def stage FirstStageFunction(var arg1, var arg2):
    job ...
        .
        .
        .

//Using a Stage Function type
pipeline FirstPipeline:
    .
    .
    .
    stage FirstStage: FirstStageFunction("1st","2nd")
```

Figure 5.25: PACE function of Stage type definition.

```
//Definition of a Job function type
def job FirstJobFunction(var arg1, var arg2):
    .
    .
    .

//Using a Job Function type
pipeline FirstPipeline:
    .
    .
    .
    stage FirstStage:
        job FirstJob: FirstJobFunction("1st","2nd")
```

Figure 5.26: PACE function of Job type definition.

Additionally, developers can create Python functions, as shown in figure 5.27, that can also be reused anywhere within a Run definition.

```
//Definition of a Python function type
def FirstPythonFunction(var arg1, var arg2):
    .
    .
    .

//Using a Python Function type
pipeline FirstPipeline:
    .
    .
    .
    stage FirstStage:
        job FirstJob:
            run:
                '''
                ##FirstPythonFunction("1st","2nd")
                '''
```

Figure 5.27: PACE function of Python type definition.

The reuse of Stage or Job functions is done by placing after of the definition of a Stage or Job (depending on the type of function being inherited) the name of the function followed by the arguments to be passed, as shown in the figures 5.25 and 5.26.

To reuse Python code, inside the “run” code block we use two “#” to call a Python function that can be declared in the same or another script, as shown in the figure 5.27.

By doing this, the new Stage will behave as defined in the Stage type, with the desired parameters. However, developers can override or extend these behaviours by adding the corresponding keywords after the function call, as shown in figures 5.28 and 5.29.

```
stage FirstStage: FirstStageFunction("1st","2nd") override:
    job FirstJob:
        run:
            '''
            print "Hello Guys!!"
            '''
```

Figure 5.28: PACE Stage using override feature.

```
stage FirstStage: FirstStageFunction("1st","2nd") extends:
    job JobExtended:
        run:
            '''
            print "Hello Guys!!"
            '''
```

Figure 5.29: PACE Stage using extends feature.

By using the extends developers can add new Jobs, new fetches, new artifacts and/or new variables. If developers are overriding what the Stage inherited, they need to give the same name as to what they need to override.

As mentioned, the concepts of override and extend can also be applied to the job, as shown in the figures 5.30 and 5.31.

```
job FirstJob: FirstJobFunction("1st","2nd") override:
    run:
        '''
        print "Hello Guys!!"""
        '''
```

Figure 5.30: PACE Job using override feature.

```
job FirstJob: FirstJobFunction("1st","2nd") extends:
    artifacts:
        "output/Hello.txt"
```

Figure 5.31: PACE Job using extends feature.

5.5 Implementation Details

With the set of tools to be used during the implementation presented, the details of the proposed DSL implementation in chapter 4 on the industrial context of OutSystems are now described in this section.

The development of PACE after its design be completed was divided into three main points:

- Definition of grammar rules;
- Code Generator Programming;
- Creating Custom Validation Rules.

5.5.1 Definition of grammar rules

A correct definition of grammar rules is what allows PACE to be accepted by the compiler.

A small part from the grammar defined for PACE can be found in figure 5.32. In this figure, we can see that the first grammar rule defines that a PACE script is composed of 1 or more (+) ProgramElements, defining that this script does not compile if the file is empty. The ProgramElements rule delegates the Import, or (|) Functions, or (|) Include rule, and so on. Finally, the rule Telemetry defines that begins with the keyword "telemetry =", followed by an identifier that will save values of type boolean.

```
grammar org.xtext.dsl.Pace with org.eclipse.xtext.common.Terminals

generate pace "http://www.xtext.org/dsl/Pace"


Program:
    (elements+=ProgramElements)+
    ;

ProgramElements:
    Import |
    Functions |
    Include |
    Pipeline   |
    Telemetry |
    Variable
;

Telemetry:
    'telemetry' '=' value?=BOOLEAN
;
```

Figure 5.32: Example of PACE grammar.

Following this, the remaining rules were defined. These rules, when carefully defined, allowed

for compile-time validations to be made automatically. In addition, it enabled IntelliSense without any effort on the part of the implementation, which was a surprise because it turned out to be a valuable help and accelerator for developers to know what they could write without having to consult the documentation about the PACE structure.

The definition of grammar is therefore not only useful for the parser, but also to be able to use various features, "such as handling of cross-references, code completion, navigation, syntax coloring, and validation" [Xtec].

The grammar implemented in the context of this prototype is represented in appendix F. This grammar formally represents PACE composition that was designed over several iterations.

5.5.2 Code Generator Programming

The programming of the code generator is what will allow the code that is written in PACE to be compiled into JSON. This is where the transformations of the PACE concepts to the concepts of the GoCD presented in section 4.3.4 are made. The code is done using the Xtend language, as presented in section 5.2.

In figure 5.33 we can see a function developed for PACE where we can see in blue with a gray background the part of the code that belongs to a template, meaning that it will always be the same for all the build pipelines created using PACE. At the end of the figure, we see that part of the code to be generated depends on what the user defines in the PACE script. All of the PACE code generation for JSON is done in a way that is similar, where there is a template to be manipulated to generate the code defined in the PACE script.

```
public def static telemetry_end(String state)'''
    ,{
        "command": "python",
        "working_directory": "platform/QA/ContinuousIntegration",
        "arguments": [
            "telemetry_job_end.py",
            "--pipeline_name",
            "%GO_PIPELINE_NAME%",
            "--pipeline_counter",
            "%GO_PIPELINE_COUNTER%",
            "--stage_name",
            "%GO_STAGE_NAME%",
            "--stage_counter",
            "%GO_STAGE_COUNTER%",
            "--job_name",
            "%GO_JOB_NAME%",
            "--branch",
            "%BRANCH_NAME%",
            "--system",
             "%CI_SYSTEM%",
            "--status",
            «IF state.equals("passed")»"Passed"«ELSE»"Failed"«ENDIF»

        ],
        «IF state.equals("passed")»"run_if": "passed"«ELSE»"run_if": "failed"«ENDIF»,
         "type": "exec"
    }
    '''
```

Figure 5.33: Example of PACE code generator.

Among the various code transitions from PACE to JSON, a few are highlighted:

- Resources
    - Detection of resources required to run a Job on a particular machine is done at compile time. The explanation of how this detection is made is seen in section 5.3.

- Dependencies
    - Dependencies in GoCD are seen as materials, and we can specify both repositories and pipelines as materials;
    - In PACE, the definition of dependencies between Stages is not done with the specification of the repositories, but when the Stage is defined. So at compile time, the repositories and dependencies will be joined to generate the correct JSON code for the GoCD.
- Variables
    - In PACE it was decided to have only one concept of variables, unlike GoCD where there were environment variables and parameters;
    - At compile time, the variables involved with the artifact concept will be transformed into GoCD parameters and the rest into environment variables. This decision was made so that the existing Python scripts in OutSystems can access the variables since they are made to access environment variables and not to variables passed as arguments.
- Default Values
    - PACE allows some values that are not set by the user to have a default value. Among these values are: the timeout; the trigger (by default is automatic); and when developers want to save an artifact, if we do not specify if the type is build or test, by default the type is build. Another default value is telemetry that is set to true.
- Templates and Environment
    - Due to the support of code reuse implemented in PACE it is no longer necessary to have the concept of templates present in GoCD;
    - The choice of environment to which the build pipeline will belong is made at compile time using a name convention based on the name assigned to the pipeline.
- Concept of Tasks, Jobs, Stages and Pipelines
    - The concept of Tasks, Jobs, Stages, and Pipelines in the GoCD is different from the concepts presented in PACE, as already presented in subsection 4.3.4. At compile time is when the mapping of PACE to GoCD concepts are done;
    - Regarding the concept of tasks that is present in the GoCD and other CI/CD systems, in PACE this concept disappears, and the tasks that a Job will run must be programmed in Python. This way, at compile time a Python file is generated which will then be called in the GoCD as being a task to make a call to a Python file from the command line.
- Telemetry
    - Telemetry is a set of commands defined as a template in the code generator. When this feature is true, the code belonging to the telemetry is added to the generated JSON files.

5.5.3 Creating Custom Validation Rules

Since the validation rules generated automatically by the grammar may not be enough, it is necessary to create the rules that we want.

These validation rules allow for increased developer productivity and achievable due to the fact that PACE works at a greater abstraction level. This distinguishes PACE from standard pipeline as code solutions that treat code as simple tool configurations with little or no semantics behind the structure.

Figure 5.34 is an example of a validation created in the context of this DSL that verifies if there are variables with the same name within the definition of a certain Stage. If so, at compile time a message is displayed specifying the error, and where does the error manifest itself.

```
@Check
def checkDuplicatedVariablesStage(Stage s) {
    val names = newHashSet
    var i=0
    for(i=0;i<s.variable.length;i++)
        if(!names.add(s.variable.get(i).name))
            error('You can not have variables with the same name.', PacePackage.Literals.STAGE__VARIABLE,i)
}
```

Figure 5.34: Example of PACE validation rule.

5.6 Comparison of build pipelines made with DSL and JSON

In order to be able to observe the difference in the implementation of a build pipeline in JSON and PACE, this section will show the implementation of the build pipeline represented in figure 5.35.

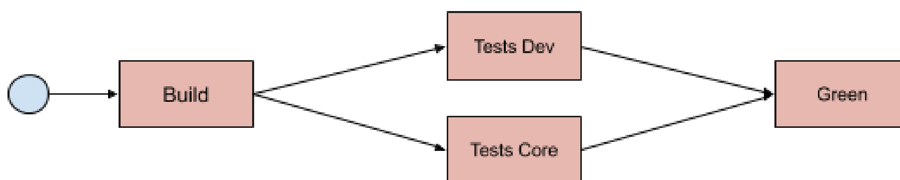


Figure 5.35: Build pipeline implemented.

Figure 5.35 represents a build pipeline that is manually activated and is responsible for building software and for doing Dev and Core tests. When the tests pass, Green is responsible for printing the revision number.

For this example let's assume that build and test code can be reused.

The build pipeline when being programmed in PACE would need to program the code represented in figure 5.36, where the reader can see a total of 23 Lines of Code (LOC) programmed in a single file.
The pipeline when being programmed in JSON would be necessary to program the represented in figures 5.2, 5.3, and 5.4, where in total 573 LOC divided by 6 files are programmed. Due to

```
include "library.pace"

pipeline dissertation_example:

    repositories:
        trigger manual:
            git "https://github.com/OutSystems/GoCD-Configurations" to "platform/Scripts"

    stage Build: build_ps ("trunk","true")

    stage Tests_Dev dependsOn Build : test_PS("dev", "Build")

    stage Tests_Core dependsOn Build: test_PS("CorePlatform", "Build")

    stage Green dependsOn Tests_Dev & Tests_Core:
        job Green:
            fetch:
                "s3_download_keys/cintia_ps_dependency.txt" from stage Build job build_ps.Build to "s3_download_keys"
            run at "s3_download_keys":
                '''
                f = open("cintia.txt", "r")
                print(f.read())
                '''
```

Figure 5.36: Build pipeline, represented in figure 5.35 implemented in PACE.

the number of LOC, only the build, environment and Pyhton files will be represented, since the rest are similar to the build file.

```
f = open("cintia.txt", "r")
print(f.read())
```

Listing 5.2: Python file to create the build pipeline represented in figure 5.35.

```
{
 "format_version" : 3,
 "name": "dissertation_example",
 "environment_variables": [],
 "pipelines": [
  "Build",
  "Tests_Dev",
  "Tests_Core",
  "Green"
 ]
}
```

Listing 5.3: Environment file to create the build pipeline represented in figure 5.35.

```
{
 "format_version": 3,
 "group": "dissertation_example",
 "name": "Build",
 "label_template": "${COUNT}",
 "parameters": [],
 "lock_behavior": "unlockWhenFinished",
 "environment_variables": [],
 "materials": [
  {
   "url": "https://github.com/***/GoCD-Configurations",
   "destination": "platform/Scripts",
   "name": "1",
   "auto_update": true,
   "type": "git"
  }
 ],
 "stages": [
  {
   "name": "Build",
   "fetch_materials": "true",
   "never_cleanup_artifacts": false,
   "clean_working_directory": true,
   "approval": {
    "type": "success",
    "users": [],
    "roles": []
   },
   "environment_variables": [
    {
     "name": "CI_SYSTEM",
     "value": "PlatformServer"
    },
    {
     "name": "MSBUILDDISABLENODEREUSE",
```

```
      "value": "1"
    },
    {
      "name": "FEATURE_FLAG_REDUCE_BUILD_SIZE",
      "value": "True"
    },
    {
      "name": "branch_root_directory",
      "value": "https://****.outsystems.com/svn/platform"
    },
    {
      "name": "MSBUILD",
      "value": "C:\\Program Files (x86)\\BuildTools\\MSBuild.exe"
    }
   ],
   "jobs": [
    {
      "name": "Build",
      "environment_variables": [],
      "tabs": [],
      "resources": ["Build"],
      "artifacts": [
       {
        "source": "platform/QA/build_reports/*.txt",
        "destination": "build_reports",
        "type": "build"
       },
       {
        "source": "platform/QA/output_html.html",
        "type": "build"
       },
       {
        "source": "platform/s3_download_keys/*.txt",
        "destination": "s3_download_keys",
        "type": "build"
       }
      ],
      "properties": [],
      "run_instance_count": null,
      "tasks": [
       {
        "command": "python",
        "working_directory": "platform/Scripts/PACE/PythonFiles",
        "arguments": ["dissertation_example_Build_Build.py"],
        "run_if": "passed",
        "type": "exec"
        }
      ]
    }
   ]
  }
 ]
}
```

Listing 5.4: Build file to create the build pipeline represented in figure 5.35.

As we can saw in these two examples presented, in PACE we just use one file to program the entire build pipeline, while in JSON we need to create 6 files. One represents the environment in which the build pipeline is inserted, another represents the Python file needed to program the code for the green run, and the rest represent the definition of build, test, and green.

In addition, we note that there are much more configurations that need to be specified in JSON than in PACE since in PACE some of the settings are generated either automatically, or by default, or through what the user specifies.

In relation to telemetry, we can observe that in development in JSON we need to add several LOC, passing several arguments to the Python file that is being called. In PACE to have telemetry does not have to add anything, since it is a configuration that is set a true by default. Furthermore, contrary to JSON, the telemetry code is not hardcoded in every pipeline stage's code, instead is injected during compilation time in the final code depending on the telemetry flag. It is treated by PACE as a cross-cutting aspect that is automatically included in the final code.

In JSON we can also see that we need to specify the resources for a given Job, whereas in PACE they are generated automatically.

Looking into variables, it can be observed that in JSON there are two concepts, environment variables, and parameters. Regarding environment variables, they can still be specified in several places, which can lead to some confusion on the part of the developer. In PACE only exists one type of variable that can be set anywhere in the file.

In general, we can realize that programming directly in JSON requires the developer to produce much more LOC, and several files that connect to each other through names which can be error-prone. In addition, it also requires the developers to know much more concepts and details to be able to program a build pipeline, which can lead to a higher learning curve in JSON than in PACE. We can also see that the only type of reuse in JSON is that we can call the same Python file several times, while in PACE we can reuse it at various levels and we can also apply the concept of inheritance. Finally, in the case of the stage Green presented in the example, when developing in JSON it was necessary to create a separated Python file and then make the call to that file in JSON, whereas in PACE it was all done in the same script.

Having into account these two different approaches to implement build pipelines, we can state that creating build pipelines with PACE is faster and more productive. In order to provide evidence to support this statement, in the next chapter experimental validations are made within the industrial context of OutSystems.

5.7 Conclusion

This chapter presented the whole process that allowed the solution proposed in chapter 4 to be implemented. Here we can see a description of the tools used during the implementation, the iterative decisions about the design to be implemented, the final design, the most important implementation details for understanding how PACE syntax translation is done for JSON, and final comparison in the development of build pipelines using PACE or JSON.

During the implementation of this prototype, certain decisions were made due to the fact that what was being implemented was only a prototype for proof of a concept, not a final version of PACE. Among these decisions was the usage of non-typed variables and not supporting recursively reuse of code, since the implementation added complexity and did not offer much value to demonstrate that DSL solves the problem presented in chapter 1.

The implementation of this prototype was done in the industrial context of OutSystems but can be easily integrated into a different industrial context, since the solution designed does not have a strong connection with the context of OutSystems.

In the next chapter, the experimental validation of this prototype is done in the real context of OutSystems, so that evidence be collected on whether the described problem is solved or not.

Chapter 6

Experimental Validation

The previous chapter presented how the DSL PACE prototype was implemented. In order to provide evidence that the prototype achieves the initially proposed objectives, a set of experimental validations have been designed and conducted. These experiments are described in detail in this chapter. In section 6.2 we can find the plan created, in section 6.3 the results, and in section 6.4 a critical analysis of the obtained results.

6.1 Introduction

Normally, build pipelines that validate software are complex, and the current pipeline as code implementations offered by the CI/CD tools analysed in chapter 3 do not reduce that complexity.

To create build pipelines there is a set of configurations and concepts that the developers need to know, and there is no way to free a developer to know these configurations. Currently, in industry, it is common to have a team (or teams) enabling the development teams to manage and create their build pipelines in a self-service manner. OutSystems presents a complex product, so is sometimes hard to get a team to become a service provider instead of enablers. This is considered an anti-pattern that can hurt the developer teams flow, and that is the exact situation that OutSystems is trying to avoid.

In this thesis, we propose a DSL that helps the different R&D teams create build pipelines with a tool provided by Engineering Productivity Group, thus moving towards a self-service philosophy regarding build pipelines.

The DSL allows developers to create build pipelines logic, program tasks in Python language that is well known by the developers, create chunks of code that can be reused elsewhere, while having a simple and understandable syntax that abstracts concepts used by the CI/CD tool, and maintaining all expressiveness in creating build pipelines.

The purpose of this chapter is to present a set of validations, where a set of metrics were collected to verify that PACE prototype can reduce the learning curve, and improve the developer's productivity in the creation of build pipelines.

A validation strategy has been defined, which will be explained in more detail in section 6.2. This strategy was based on the vast OutSystems internal experience in performing usability testing with OutSystems product and on the study of information related to experimental validations.

With this knowledge, a strategy was defined where it was necessary to use a control group, which performed the tasks in JSON, and an experimental group, which performed the tasks with

PACE prototype, in order to:

- Check if PACE increases productivity;
- Check if PACE reduces the possibility of errors being made;
- Identify usability issues in PACE;
- Evaluate the cost of learning PACE.

With these four evaluated topics during the execution of two tasks explained in section 6.2, we were able to evaluate the effectiveness, efficiency, and usability of the constructed prototype. These characteristics are evaluated through an analysis of the execution time of each task, the behavior of the person performing the test, the completion of a form that allows obtaining the SUS and NPS, and the analysis of the correction of the proposed tasks. The SUS is a set of 10 questions that allow measuring the usability of our system [Jef11], whereas the NPS is a question that allows determining the user experience that the subject had during the experimental validation [NIC].

6.2 Plan

The objective of this experimental validation is to be able to find evidence that the requirements defined in chapter 1 are being fulfilled in the designed and implemented prototype. In addition, the performed validations also aim to identify usability issues for improvements to be implemented in the prototype.

In order to evaluate the effectiveness and usability of PACE, a validation plan was created. The plan was designed to obtain metrics to evaluate the effectiveness and usability of the PACE with only one experimental validation. To determine the number of people to interview for the analysis of the efficacy and usability of PACE, the recommendations of Jakob Nielsen were followed [Jak00]. Although these recommendations are for usability testing, as already mentioned, due to the timeframe they were also used to analyze the effectiveness of PACE.

According to Jakob Nielsen, with 3 users we can detect most of the usability problems, and the best results come from the first 5 people if they belong to the same user group [Jak00]. As we increase the number of users to be tested the gain ends up decreasing more and more [Jak00]. Jakob Nielsen and Tom Landauer have shown that the number of usability problems encountered in tests with n users is represented in the equation (6.1) [Jak00].

$$P = N(1 - (1 - L)^n) \quad (6.1)$$

where:

- P = Number of usability problems found;
- N = Total number of usability problems in the design;

- L = The proportion of usability problems discovered while testing a single user. The value of L used which represents the curve of the figure 6.1 is 31% due to the number of projects studied by Jakob Nielsen and Tom Landauer [Jak00];
- n = Number of test users.

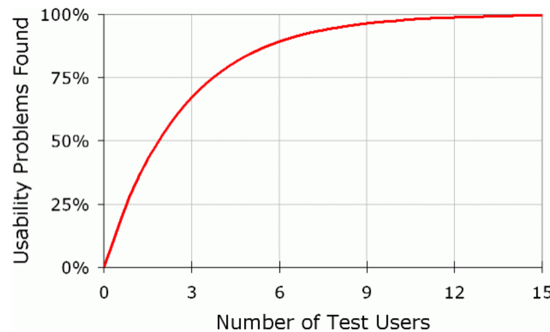


Figure 6.1: Curve for L =31%.

In order to control the gains between the use of PACE and the use of JSON we used two distinct groups, each of which have 5 OutSystems R&D developers responsible for product development, with practically no context on the concepts of build pipelines, and on the creation and maintenance of build pipelines. In this way, we were following the recommendations of Jakob Nielsen.

In the remaining of this chapter, we will refer to these two groups by experimental group and control group:

- Control Group - Group composed of 5 people, who do not belong to the experimental group, responsible for performing the proposed tasks using JSON pipeline as code that the GoCD has. In this way, we can represent the impact we have when manipulating and creating build pipelines in the way that the Engineering Productivity Group works.
- Experimental Group - Group composed of 5 people, not belonging to the control group, responsible for performing the proposed tasks using the developed prototype. In this way, we can evaluate the development impact between PACE and the GoCD pipeline as code.

The tasks were done individually, to understand all the difficulties that each person felt during the development. The “thinking aloud” technique was used, which allows us to realize the difficulties that the test subjects encounter along the way, since in the end people tend to be nice and not give a true opinion about the difficulties experienced. In addition, the execution time of each task was also measured. At the end of the experimental validation, the errors made by each person and their answers to the questionnaire were evaluated.

A short presentation (15 minutes) about concepts related to build pipelines, and the explanation of how to program a build pipeline using an example, was done to all participants at the beginning of the experiment. While performing tasks the test users could access the material previously presented, but their questions during this time were not answered. In the end, the test users’ questions were answered and a questionnaire was answered composed of a SUS, NPS, and improvements that would make to the pipeline as code that they were using.

Tasks being made individually introduced the risks of group contamination if there was a dialogue between the different users about the test being made. To eliminate this risk, each person was asked not to tell anyone about what happened during their test.

During the experiment, we measured the time each user took to complete each task. After a result analysis was done with all the measured times. To do this, the test user was asked to give an alert when he thought the task was complete, to stop the timer and provide him the material for the next task. In this way, it was also possible at the end of each experiment to make an analysis of the correction of the tasks that each test user performed, to collect metrics related to effectiveness.

In order for everyone in both groups to be in the same test environment, a timeframe has been defined. Each experimental validation lasted for one hour, where within this time, the time was divided as follows:

- 5 minutes - Present the purpose of the test, and how it will run;
- 15 minutes - Presentation of the concepts of Pipeline/Stage/Job and explanation of how to program a simple pipeline (in JSON, or PACE, depending on the group);
- 35 minutes - Execution of the tasks;
- 5 minutes - Questions and filling out a questionnaire.

The timeframe was strictly followed, and all material presented before, after and during the execution of the tasks was the same so as not to influence results.

The tasks to be performed during the test were based in real build pipeline scenarios currently being used by OutSystems to validate its product. This decision was made to make the tests scenarios closer to reality.

In both tasks, the users could access the pipeline as code documentation they were using, the presentation, the example shown previously to the task, and the Python documentation. While we allowed users to access the Python documentation, since the purpose of the test was not to test the development capabilities in Python, the proposed tasks were made to use Python code that was in the example initially shown.

In task 1 the goal was to manipulate a build pipeline, and in task 2 the goal was to create a build pipeline. This build pipeline was similar for both tasks and is represented by figure 6.2.

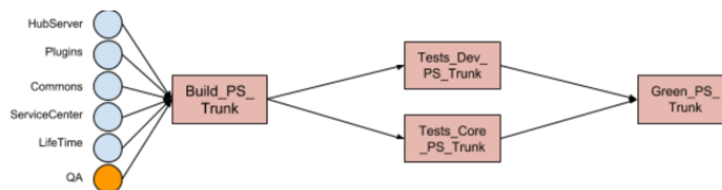


Figure 6.2: Pipeline to be created in both tasks.

In figure 6.2, the blue balls represent the SVN repository folders that will trigger the pipeline whenever commits are made to files within those folders. The orange ball represents the use

of a folder that is present in the SVN repository, but for which we do not want to trigger the pipeline whenever a commit is made in that folder.

In appendix E the script for the control group can be found and in appendix D the script for the experimental group.

6.2.1 Task 1

For this task, the user only had access to the materials previously specified, and the purpose of this task was the addition of new artifacts produced by one of the Stages to be stored. This task was actually inspired by a former real case that occurred in the past and had to be solved by the Engineering Productivity Group.

The test user was asked to make a change to an existing build pipeline that was defined within a file. This change was saving two new artifacts to be produced by the Tests_Core_PS_Trunk Stage. These artifacts are:

- Artifact 1
  - Source: platform/QA/Results/**/*.obtained.png
  - Destination: Obtained
- Artifact 2
  - Source: platform/QA/Results/**/*.obtained.log
  - Destination: Obtained

6.2.2 Task 2

For the second task, the user was given new material that he could use during the task. Here the user could already access a file that contained a set of variables that could be useful, and another file that contained code responsible for building and testing the PS. During this task, the user was also told that he could not access the previous task scripts but could continue to see the example.

In this task, it was requested that through the figure 6.2 a build pipeline was created that was manually triggered, were each Stage represented the following:

- Build_PS_Pace
  - This Stage is responsible for building the PS and will generate a set of artifacts that will then be used by the remaining Stages.
- Tests_Dev_Pace & Tests_Core_Pace
  - These two Stages are responsible for running the dev and core tests against the PS that was built.
- Green_PS_Pace

- This Stage, in this example, will just fetch an artifact produced by the build that contains the build version and prints the green version on the screen.

The goal with this task was for users to create a build pipeline reusing the code for build and tests, and to program everything needed for the Stage Green_PS_Pace. Remembering that the Python code required for this last Stage was already programmed in the example initially presented.

6.3 Results

Here we present the results obtained from the experimental validation plan previously presented in section 6.2.

The global results represent the results obtained from the questionnaire done at the end of the tasks, which is composed of the SUS and NPS questionnaire, and the question of two or three things that could be improved in the DSL PACE, or in the GoCD JSON pipeline as code. In the table 6.1 we can see the results of the control group, and in the table 6.2 the results obtained for the experimental group.

Table 6.1: JSON questionnaire results.

	SUS	NPS
Test 1	35 Awful	4 Detractor
Test 2	40 Awful	2 Detractor
Test 3	37.5 Awful	6 Detractor
Test 4	50 Awful	6 Detractor
Test 5	80 Good	7 Neutral
	SUS = 48.5 Awful	NPS = -40 Needs Improvement

Table 6.2: PACE questionnaire results.

	SUS	NPS
Test 1	80 Good	8 Neutral
Test 2	77.5 Good	10 Promoter
Test 3	77.5 Good	8 Neutral
Test 4	85 Excellent	8 Neutral
Test 5	82.5 Excellent	9 Promoter
	SUS = 80.5 Excellent	NPS = 40 Great

The SUS score test is done through 10 questions that can get a result from 1 to 5, 1 being "strongly disagree" and 5 "strongly agree". With these questions, we can measure the usability of our system [Jef11]. The questionnaire can be found in the Appendix A. To calculate the result, for the odd items we must subtract 1 from the answer, for even-numbered we must subtract 5 from the answer, and finally add all these converted results and multiply the total by 2.5 [Jef11]. This result is now in percentile rank, which allows us to compare our score with others in the database, and if our result is above 68 it means that we are above the average [Jef18]. These percentiles can be transformed into adjectives for a better understanding of the results. The mapping of percentiles to adjectives is represented in the table 6.3. The SUS result of all validations is done by averaging the individual results obtained.

Table 6.3: Table adapted from the mapping of the SUS result in percentiles to adjectives [Had18].

SUS Score	Adjective Rating
>80.3	Excellent
68 - 80.3	Good
68	Okay
51 - 68	Poor
<51	Awful

In relation to NPS, it is measured with the question "How likely is it that you would recommend this pipeline as code to a friend or colleague?" which can be answered with a value from 0 to 10 where 0 corresponds to very unlikely and 10 very likely. With this question, we can see the user experience that had during the experimental validation [NIC]. The interpretation of the result obtained individually from each user can be found in table 6.4. The NPS result of all validations is calculated using the equation represented in (6.2). The interpretation of the final result can be done according to the table 6.5 [Sar19].

Table 6.4: Response grouping table.

NPS Answers	Group
9 - 10	Promoters
7 - 8	Neutrals
0 - 6	Detractors

$$\text{NPS} = \%p - \%d \quad (6.2)$$

where:

- NPS = Net Promoter Score;
- p = Promoters;
- d = Detractors.

Table 6.5: Table adapted from [Sar19].

NPS Score	Adjective Rating
70 -100	Excellent
30 -70	Great
0 - 30	Good
-100 - 0	Needs Improvement

The proposed improvements and criticisms made by the test users belonging to the experimental group (group using PACE language) were as follows:

- The management of artifacts should allow to change the names of the artifacts, or match the location to a variable so that if it is needed to use the variable in the “run”, there was no need to specify the full path for that artifact each time that is used;
- There should be a type system;
- Using indentation to create code blocks may not be obvious.

In relation to the proposed improvements and criticisms made by the test users belonging to the control group (group using GoCD JSON) were the following:

- Learning curve is high;
- There should be a validation of the intermediate steps;
- Documentation should be together with examples;
- Easy to make mistakes due to being all connected by names.

In the table 6.6 we can see the development time of task 1 in the experimental and control group, whether the task is finished or not, and the average time in the development in both groups.

Table 6.6: Results of task 1.

	PACE	Completed?	JSON	Completed
Test 1	00:02:14	Yes	00:02:34	Yes
Test 2	00:02:45	Yes	00:03:41	Yes
Test 3	00:02:45	Yes	00:05:07	Yes
Test 4	00:03:11	Yes	00:06:06	Yes
Test 5	00:10:13	Yes	00:07:05	Yes
	Average = 00:04:14		Average = 00:04:55	

In the table 6.7 we can see the development time of task 2, whether the task was completed or not, the time remaining in the development, and the average development time, and remaining time of both groups.

Table 6.7: Results of task 2.

	PACE	Completed?	Time Left	JSON	Completed	Time Left
Test 1	00:17:37	Yes	00:15:09	00:28:54	No	00:00:00
Test 2	00:16:28	Yes	00:15:47	00:29:53	No	00:00:00
Test 3	00:21:43	Yes	00:10:06	00:31:19	No	00:00:00
Test 4	00:20:55	Yes	00:03:52	00:27:55	No	00:00:00
Test 5	00:17:44	Yes	00:14:31	00:32:26	No	00:00:00
	Average = 00:18:53		Average = 00:11:53	Average = 00:04:55		Average = 00:00:00

6.4 Discussion of the Evaluated Results

In this section we will discuss the results obtained and presented in section 6.3.

The objectives with these tests were:

- Verify that PACE increases productivity;
- Verify that PACE reduces the possibility of errors being done;
- Identify usability issues in PACE;
- Assess the cost of learning.

In order to achieve these objectives, the analysis carried out can be divided into:

- Formal Analysis - In this type of analysis, we are able to compare the resolution times of each task, whether or not the test users are able to complete the task, the result of the SUS and NPS questionnaire. Furthermore, using statistical tests it is possible to obtain evidence that a difference exists, or not, between the two groups. These analyzes mean that the conclusion we get, any other person can get.
- Informal Analysis - Conclusions about the behavior and difficulties experienced by the test user during the test, and the analysis made of each one of the answers of the SUS questionnaire are drawn. Contrary to formal analysis, here the conclusion we get may not be the same as someone else's.

6.4.1 Formal Analysis

The purpose of this analysis is to be able to answer the following questions:

- RQ1: Does the use of PACE improve a programmer's efficiency when creating/maintaining a build pipeline?
- RQ2: Does the use of PACE improve a programmer's effectiveness when creating/maintaining a build pipeline?
- RQ3: Is PACE usability better than JSON usability?
- RQ4: Is there a better experience in creating/manipulating build pipelines with PACE than JSON?

For this, was made an analysis of the time spent by each of the test users in each of the tasks, and an analysis to the result of a questionnaire composed by a SUS, NPS, and improvements that they would make to the pipeline as code. In each of the tasks, we will see the result of the development time of the tasks, their average, and whether or not the tasks are finished. Mann-Whitney U statistic test was calculated for each of the tasks with the metrics of the time spent by each of the test users.

The Mann-Whitney U statistical test is a non-parametric test that allows comparing two samples from the same population to see if there is evidence that they are equal or not [Sta]. For this statistical test to be done we have to assume that [Sta, Soc]:

- The sample drawn from the population is random;
- Independence within the samples and mutual independence is assumed. That means that an observation is in one group or the other (it cannot be in both);
- The data is continuous;
- Ordinal measurement scale is assumed.

The equation of this test is represented in the equation (6.3). This equation is applied to two samples, and the smallest result is our Mann-Whitney U test result. This result must be equal to or less than the critical value in order to have statistical evidence. The critical value differs from the number of samples used, and we can find it in the table represented in the appendix B. The significance level used for this experimental test was 0.05, and the assumptions to be approved or rejected are described below:

- H_0 Null hypothesis - There is no difference in productivity between the use of PACE and the use of JSON;
- H_1 Alternative Hypothesis - There is an increase in productivity with the use of PACE instead of using JSON.

$$U = n_1 n_2 + \frac{n_2(n_2+1)}{2} - \sum_{i=n_1+1}^{n_2} R_i \quad (6.3)$$

where:

- U = Mann-Whitney U test;
- n_1 = Sample size one;
- n_2 = Sample size two;
- R_i = Rank of the sample size.

6.4.1.1 Task 1

RQ1: Does the use of PACE improve a programmer's efficiency when creating/maintaining a build pipeline?
Starting with the development time corresponding to task 1, we can observe that the average development time is very similar with only 41 seconds of difference, as we can observe in table

6.6. What causes this time difference to be greatly reduced is the time that the test user 4 led solving task 1. This test user used a different approach to the problem of the remaining ones. He started by first looking at the documentation, for the example shown, and the scripts for task 1 in order to know what he was dealing with. It was only after this analysis that this user test passed to the resolution of the task, and this caused the time of task 1 to be approximately 10 minutes, thus increasing the average time of resolution of the first task. With the exception of this case, if we look at the graph 6.3 we can observe that the resolution time is lower for those who solve this task in PACE than in JSON.

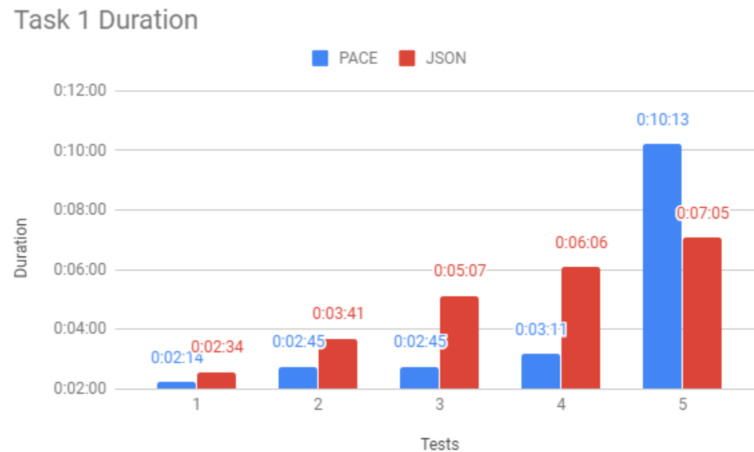


Figure 6.3: Task 1 duration times.

The lowest result obtained in the Mann-Whitney U test was 8. Since 8 is greater than 2, and 2 is the critical value of Mann-Whitney U for a significance level of 0.05, the test had no statistical evidence.

Although the graph 6.3 shows that the resolution time is generally lower for those who develop in PACE than in JSON, as we have seen, the result of the Mann-Whitney U test does not present enough statistical evidence leading to accept the null hypothesis, evidencing that the productivity was equal between the use of PACE and the use of JSON. Although this is the result obtained, the fact that there is no statistical evidence does not mean that productivity does not increase with the use of PACE, since by the analysis of the graph we can see that there is an increase in productivity for those who used the PACE prototype.

With this analysis, we have evidence that PACE may increase the efficiency of maintaining build pipelines.

RQ2: Does the use of PACE improve a programmer's effectiveness when creating/maintaining a build pipeline?

Still in relation to task 1, it was possible to observe that all the test users were able to finish the task, without making mistakes. In this way, we cannot show evidence that the use of PACE or JSON changes the effectiveness in relation to the maintenance of build pipelines.

6.4.1.2 Task 2

RQ1: Does the use of PACE improve a programmer's efficiency when creating/maintaining a build pipeline?
In relation to task 2, by the analysis of table 6.7, we can see that the average between the two different groups were substantially different, showing clear differences in productivity related to the creation of a build pipeline using PACE and JSON. The difference in the average development time was approximately 11 minutes, whereas the control group, despite having spent more time in the development, no test user was able to complete the task, unlike the experimental group, which had an average remaining time of approximately 12 minutes, and where all users successfully completed the task.

The lowest result obtained in the Mann-Whitney U test was 0. Since 0 is lower than 2, and 2 is the critical value of Mann-Whitney U for a significance level of 0.05, the test had statistical evidence.

The result observed by the measurement of the times was supported by the statistical test result, where sufficient statistical evidence was presented that led to the rejection of the null hypothesis and to accept the alternative hypothesis that there is a difference in productivity between the use of PACE and JSON. This statistical result showed evidence that using PACE may increase the productivity of developers when they create build pipelines, compared to the current build pipeline development in OutSystems R&D.

As mentioned, no person in the control group was able to complete task 2 within the proposed 35 minutes for both tasks. However, there was a person (test user 3) who was very close to the end since he was reviewing the whole exercise to confirm that it was correct when the time finished. The remaining test users were almost in the middle of the exercise.

With this analysis, we have been able to show evidence that using PACE increases the efficiency in creating build pipelines when compared to JSON.

RQ2: Does the use of PACE improve a programmer's effectiveness when creating/maintaining a build pipeline?
Although this task has been marked as completed by the experimental group, some errors were made. However, the errors were less than the errors made by the control group. Due to the development time left in the experimental group, they were asked to review the exercise and during this review, some of the errors were found and corrected. For the errors not found, the test users were guided towards them and some test users were able to correct them.

The errors made by both groups were:

- Not specifying the trigger of the build pipeline as being manual, a requirement that was requested;
- The checkout location of the different repositories was always the same, which would be an error in GoCD.

The errors made by the experimental group were mostly related to the fetch location of an artifact that was being generated by the build.

The errors made by the control group were:

- Not passing the arguments necessary for the python files run;
- There was no connection from the repositories to the build pipelines;
- Confusion with the fetch concept, having questions as to how and where it could make fetch to a material;
- All repositories triggered the build pipeline when a commit was made, and it was requested that one of the repositories should not trigger the build pipeline.

Most of the errors were due to:

- Copying existing code without changing it afterwards;
- Not knowing the syntax.

However, despite these errors, the experimental group was able to identify and correct most of them.

From this analysis it is possible to find evidence that when creating a build pipeline with PACE developers are more effective than with JSON, being fewer errors made.

6.4.1.3 General

RQ3: Is PACE usability better than JSON usability?
Analyzing the tables 6.1 and 6.2 we can see the SUS result obtained by the experimental group was 80.5, which means that in terms of usability PACE presents the minimal of excellence, while in the control group the score was 48.5, which means that in terms of usability is horrible. Therefore, in terms of usability, PACE language presents better results than JSON.

RQ4: Is there a better experience in creating/manipulating build pipelines with PACE than JSON?
Regarding the NPS result, the experimental group had a result of 40, which means that the experiment was great, whereas the control group had a score of -80, which means that improvements are necessary.

These results, therefore, demonstrate that users prefer to use the PACE prototype instead of using the pipeline as code provided by GoCD.

6.4.2 Informal Analysis

Regarding the conclusions that we can get through what was observed during the experimental validations, we identified several challenges felt by the test users that lead us to reach several conclusions. Therefore, the purpose of this analysis is to be able to answer the following questions:

- RQ1: Is PACE easier to learn than JSON?

- RQ2: How can the usability of PACE be improved?
- RQ3: Do developers feel more confident in using PACE instead of JSON?

RQ1: Is PACE easier to learn than JSON?
One of the topics to be evaluated is related to the learning curve present in the control group and in the experimental group. Although the same help and the same material were given to both groups, in the end, the control group showed that there was a difficulty in understanding all the settings presented initially. With this, and with one of the questions asked in the SUS questionnaire, where answers are found in the appendix C, we can conclude that the learning curve to develop with GoCD's JSON pipeline as code is higher than with PACE. One factor that makes this learning curve higher in JSON is that the documentation was not well structured since the explanation of the settings was separate from the examples.

The lower learning curve, along with the fact there are fewer concepts to learn and less code to write, allow us to justify that there is an increase in the development speed when using PACE when compared with using JSON.

RQ2: How can the usability of PACE be improved?
Through the analysis of the challenges experienced during development, we have been able to detect a number of usability problems related to PACE:

- Confusion with the concepts of artifacts - Here the test user has experienced difficulties in using the concept of artifacts, having to consult the documentation in order to understand the semantics of this concept.
- PACE only consider one native datatype, the string type - All values directly managed by PACE are of type string, and this makes the test user confused.
- Not being able to test as it develops - Although the test users can tell if what he was programming was syntactically correct, they lacked being able to code and test the build pipeline gradually in order to validate if they were programming what has been requested.
- Not be possible to know which arguments pass to the functions without having to see the header of the function - The test user showed interest in having help to know what arguments should be passed when a function is called without having to consult the function header.

RQ3: Do developers feel more confident in using PACE instead of JSON?
An interesting fact emerged from the analysis of the SUS questionaire listed in Appendix C. The SUS questionnaire has a question related with confidence when using the system. By analyzing the answers we can see that the experimental group felt more confidence in the manipulation of build pipelines than the control group, which is a good indicator of a potential good adoption of PACE by the developers.

6.5 Conclusion

The purpose of this chapter is to provide evidence that with the solution proposed in chapter 4 we can provide developers with a DSL where they can manipulate complex build pipelines in a

simple way. In order to assess the relevance of our proposal, we have undertaken an experimental validation that gave a collection of different metrics that allowed to reach some conclusions.

In the two tasks performed, we were able to show that PACE increases the productivity of the developers since the tasks developed were faster for development in PACE than in JSON, as we can see in the tables 6.6 and 6.7, and by the figure 6.3. Although we have seen that in both tasks the development time was lower with PACE, there is stronger statistical evidence of increased productivity with the use of PACE. Although the Mann-Whitney U statistical test did not present statistical evidence in task 1, it does not mean that PACE does not improve productivity for simple tasks, since looking into figure 6.3 we can notice that with the exception of one case there was an increase in productivity.

In addition to this conclusion, we also have evidence that PACE has a lower learning curve than JSON, and that the use of PACE shows evidence that errors are less likely to be done since during this experimental validation this was verified.

From this experimental validation, some future improvements to PACE were also identified.

In this experimental validation were also detected some usability problems that must be considered in future versions of this prototype, so that the adoption of PACE by the developers might be done in a smoother way.

With the evidence shown in this chapter, the validation results show that PACE prototype successfully solves the problem presented in chapter 1, where the complexity of the OutSystems product lead to the creation of build pipelines that over time have become very complex, and this become difficulty for different teams to follow and understand all the terms, concepts, and techniques used to create a build pipeline that validates the OutSystems product.

Summarizing, with the collected evidence we can answer the questions initially made as:

- Formal
  - RQ1: Does the use of PACE improve a programmer's efficiency when creating/maintaining a build pipeline?
    - Yes.
  - RQ2: Does the use of PACE improve a programmer's effectiveness when creating/maintaining a build pipeline?
    - Yes.
  - RQ3: Is PACE usability better than JSON usability?
    - Yes.
  - RQ4: Is there a better experience in creating/manipulating build pipelines with PACE than JSON?
    - Yes.
- Informal
  - RQ1: Is PACE easier to learn than JSON?

    * Yes.

  - RQ2: How can the usability of PACE be improved?

    * Yes.

  - RQ3: Do developers feel more confident in using PACE instead of JSON?

    * Yes.

The problem initially defined as: The great complexity in the industries products, together with the lack of knowledge about pipelines in different development teams, makes them unable to change pipelines and build their own validation process, delaying the development of the product. It can be said as solved since with the experimental validation, we have been able to show that different elements with no context on the concepts of build pipelines have become autonomous and develop build pipelines in a productive way, which does not happen with GoCD JSON.

Chapter 7

Conclusion

This chapter presents the conclusions of the work that was done in the context of this dissertation. An overview is also made of the initially defined objectives and whether they are achieved with the implemented solution. Finally, this chapter also presents some directions on the future work that might be carried out on top of the implemented prototype.

7.1 Conclusion and Final Results

The main motivation of this dissertation was to propose a solution that allows different teams to be autonomous in handling build pipelines while hiding certain specifics that are being used in the construction of build pipelines. This reduces the level of complexity associated with build pipelines and allows a developer to quickly and simply have enough context to handle build pipelines, hence increase his productivity.

The dissertation was conducted within the industrial context of OutSystems. This context allowed to apply the problem in a real context, where there is a complex product to be validated by complex build pipelines that are maintained only by one team. This problem, also existing in other companies, means that the various teams within the company do not have the autonomy to develop their end-to-end component. All this context meant that the work developed was applied to a real case, allowing it to be evaluated in a real scenario.

In the state of the art, we can see that this is a problem present in the industry and with several solutions already implemented. These different solutions are divided into three main topics:

- The use of existing CI/CD tools;
- Creation of build pipeline libraries;
- Creation or use a DSL.

Of these three solutions it was concluded that, when applied individually, we would not be able to reach the objectives initially defined. In the case of the first topic, we would not be able to get an abstraction of the concepts related to build pipelines. In the second topic, we would not be able to eliminate the dependencies of the several teams to the team responsible for creating libraries, and its scope is limited to scenarios where there is a great degree of build pipeline standardization among different teams, that allows that those libraries can be reused by all teams. In the case of the last topic, the difficulties are related to implementation that can lead to the evolution and maintenance of DSL to become a costly process, and related to the difficulty to find the correct level of abstraction and the right balance with the DSL complexity.

The proposed solution was to combine the last two topics in order to have a set of build pipelines libraries that can be easily created by the developers through a DSL that frees them from the

complexity related to the build pipelines. Here there was a need to be careful how the DSL would be implemented in a way that would prevent the evolution and maintenance process from becoming a costly process. This proposed solution can reduce the number of configurations and concepts that the developer needs to know before starting to manipulate build pipelines. This increases the productivity of the developer since there are fewer configurations to learn, and less code to write when compared to the current build pipeline development.

The prototype, when implemented in the context of OutSystems, allowed to perform experimental validations in order to get metrics that show that the objectives were achieved. These experimental validations allowed to make an analysis of the execution time of each task, the behavior of the person being tested, and was also asked the users to fill out a form that allows obtaining the SUS and NPS. With these metrics, the Mann-Whitney U statistical test was also performed.

With the metrics, the behavior observed, and the results obtained in the experimental validation we can see that PACE shows evidence that:

- Productivity in the development of build pipelines increases;
- The possibility of mistakes being made is lower with PACE;
- It is less costly to learn how to use PACE than the current form of development in OutSystems.

It is also important to point out that all this work was able to implement the set of requirements defined initially:

- The ability to describe different build pipelines used by each R&D team, using a programming language that allows them to program their build pipelines as they do for the product they develop;
- The ability to support “code reuse” so that developers can reuse each others pipeline constructs;
- Should be able to support complex build pipeline definitions by composing modules defined in multiple source code files, according to each team’s needs;
- Should have built-in support for the most common pipelines jobs and constructs;
- Should have native support for task parallelization;
- Should be very simple to use. Developers should be able to use it with the minimum knowledge of build pipeline concepts and without the need to know anything about the underlying CD system being used or details about the jobs implementation;
- Should be able to abstract concepts related to build pipelines;
- Should be able to have language built-ins to deal with the managing (creating and destroying) the necessary agents needed to execute the jobs;
- Should be extensible;
- Be able to create build pipelines for different CI/CD tools;

- Should allow versioning of build pipeline changes using source control version systems.

Regarding the topic, “Being able to create build pipelines for different CI/CD tools”, it can be said as done because DSL PACE can easily be programmed to generate code for other CI/CD tools. Currently, this solution is only compatible with GoCD since it is the tool used within the OutSystems business context.

In short, the developed prototype was successfully validated, thus showing that the requirements and objectives initially proposed were successfully achieved. In spite of this, this is just a first prototype that was intended to prove the concept that pipeline as code, which allows configurations to be hidden and allows reuse of code, is a valid path to follow if we are trying to make different teams autonomous. Being this the first prototype, there is still a lot of space for improvements, and the next section presents some of the improvements that should be added to the prototype.

7.2 Future Work

The prototype developed has several areas that can be improved, identified through the feedback collected during the validations described previously. The most evident where:

- A better abstraction of the concept of artifacts;
- Implement a type system for variables;
- Deploy automatically, enabling the user to test as they develop;
- Show what arguments a function is waiting to receive.

To make PACE easier to adopt by developers, we present here some ideas for improvements.

One of the existing ideas should be made at the compiler level, and the objective is to make the code that is currently generated for the GoCD be able to be generated for Jenkins or other CI/CD tools without major changes needing to be made. The fact that through a DSL we can compile for various CI/CD tools is a huge gain, which is not currently being explored by the industry.

Other ideas to consider as future work are to add more validations that can be made in the IDE and compile time so that the user make more validations and have faster feedback without leaving the IDE. A possible validation to be implemented at this level is to be able to design a visual representation of the build pipeline being defined to give the developer a better idea if the build pipeline architecture goes according to the idealized one.

Currently, the login and password to access the SVN repositories need to be specified in the code, so another suggestion to be considered as future work is to find a solution that allows this type of secrets to be stored outside the code.

Because the DSL prototype was developed in the context of OutSystems, and because OutSystems is a visual language, it is proposed that experimental validation be made as to whether it is more productive or not to develop in code than to do development visually.

In order to gather even more evidence that this solution meets the objectives, it is also proposed that experimental validations be made with the manipulation of build pipelines visually with the GoCD, and with the other tools analyzed in section 3.3.1. In this way, we can see if this prototype, when compared to the various tools of CI/CD, brings improvement in the productivity of the developers. This validation was not possible during this dissertation due to the existing timeframe and time constraints.

Bibliography

[Aar17] Aaron Bjork. What is agile? [online]. 2017. Available from: https://docs.microsoft.com/pt-pt/azure/devops/learn/agile/what-is-agile [cited 20 December 2018]. 13

[ANT12] ANTLR 3. Five minute introduction to antlr 3 [online]. 2012. Available from: https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687102/Five+minute+introduction+to+ANTLR+3 [cited 1 June 2019]. 45

[Bad17] Badri Janakiraman and David Rice. What does pipelines as code really mean? [online]. 2017. Available from: https://www.gocd.org/2017/05/02/what-does-pipelines-as-code-really-mean/ [cited 17 Junho 2019]. 4

[Bri16] Britney Pay. Knowledge silos: How they're impacting your organization [online]. 2016. Available from: https://www.efilecabinet.com/knowledge-silos-how-theyre-impacting-your-organization/ [cited 8 June 2019]. 5

[Cam] Cambridge Dictionary. pace [online]. Available from: https://dictionary.cambridge.org/dictionary/english/pace [cited 1 June 2019]. 38

[Chr19] Christian Melendez. What is cicd? what's important and how to get it right [online]. 2019. Available from: https://stackify.com/what-is-cicd-whats-important-and-how-to-get-it-right/ [cited 8 June 2019]. 1

[Cor] Damien Coraboeuf. private communication, From Dec 2018 to Jan 2019. 21, 28, 30, 31

[Cor16] Damien Coraboeuf. Getting out of the job jungle with jenkins. All Day DevOps, 2016. Available from: https://www.youtube.com/watch?v=VojJ35T3-jQ&feature=youtu.be&t=7231. 4, 28, 31

[CS16] M. Callanan and A. Spillane. Devops: Making it easy to do the right thing. *IEEE Software*, 33(3):53–59, May 2016. 21

[Dav18] Davef. Continuous integration and feature branching [online]. 2018. Available from: http://www.davefarley.net/?p=247 [cited 12 October 2018]. 13

[DM18] Denise Yu and Mark Hender. What if you treat your ci pipeline as a product? PIPELINE conference, 2018. Available from: https://learn.pipelineconf.info/2018/03/21/denise-yu-and-mark-hender-what-if-you-treat-your-ci-pipeline-as-a-product-pipeline-conference-2018/. 29, 31, 35

[DMDG07] P.M. Duvall, S. Matyas, P. Duvall, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. A Martin Fowler signature book. Addison-Wesley, 2007. Available from: https://books.google.pt/books?id=MA8QmAEACAAJ. 14, 15, 16, 22

[DPvH18] T. F. Düllmann, C. Paule, and A. v. Hoorn. Exploiting devops practices for dependable and secure continuous delivery pipelines. In *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 27–30, May 2018. 16

[Eri18] Erik Dietrich. Designing autodesk's future today:ci/cd for the whole product [online]. 2018. Available from: https://www.alldaydevops.com/blog/designing-autodesks-future-todayci/cd-for-the-whole-product [cited 20 December 2018]. 14

[Exn15] Ken Exner. Transforming software development. AWS Summit, 2015. Available from: https://www.youtube.com/watch?v=YCrhemssYuI. 30, 31, 35

[Fer18] Mario Fernandez. private communication, From Nov to Dec 2018. 21, 23, 29, 31

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. 36, 37

[GFFLB13] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. Development and deployment at facebook. *Internet Computing, IEEE*, 17:8–17, 07 2013. 13

[GoCa] GoCD. Concept 6: Artifact [online]. Available from: https://www.gocd.org/getting-started/part-2/#concept_artifact [cited 20 May 2019]. 58

[GoCb] GoCD. Gocd user documentation [online]. Available from: https://docs.gocd.org/current/ [cited 1 June 2019]. 7

[GoCc] GoCD. Task [online]. Available from: https://docs.gocd.org/current/introduction/concepts_in_go.html#task [cited 1 June 2019]. 50

[Had18] Hadi Alathas. How to measure product usability with the system usability scale (sus) score [online]. 2018. Available from: https://uxplanet.org/how-to-measure-product-usability-with-the-system-usability-scale-sus-score-69f3875b858f [cited 22 May 2019]. xvii, 75

[Hel19] Helen Beal, Chris Swan, Manuel Pais, Daniel Bryant, Steffen Opel. Devops and cloud infoq trends report - february 2019 [online]. 2019. Available from: https://www.infoq.com/articles/devops-cloud-trends-2019/ [cited 14 June 2019]. 4

[Hen14] Henrik Kniberg. Spotify engineering culture (part 1) [online]. 2014. Available from: https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/ [cited 21 December 2018]. 21, 31

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. 14, 16, 22, 36

[Jak00] Jakob Nielsen. Why you only need to test with 5 users [online]. 2000. Available from: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ [cited 22 May 2019]. 70, 71

[Jef11] Jeff Sauro. Measuring usability with the system usability scale (sus) [online]. 2011. Available from: https://measuringu.com/sus/ [cited 22 May 2019]. 70, 75

[Jef18] Jeff Sauro. 5 ways to interpret a sus score [online]. 2018. Available from: https://measuringu.com/interpret-sus-score/ [cited 22 May 2019]. 75

[Jen17a] Jenkins. Pipeline [online]. 2017. Available from: https://jenkins.io/doc/book/pipeline/ [cited 22 August 2018]. 1

[Jen17b] Jenkins. Pipeline [online]. 2017. Available from: https://jenkins.io/doc/book/pipeline/syntax/#scripted-pipeline [cited 13 June 2019]. 25

[Jen17c] Jenkins. Pipeline [online]. 2017. Available from: https://jenkins.io/doc/book/pipeline/syntax/#compare [cited 13 June 2019]. 25

[Jen18] Jenkins. Pipeline as code [online]. 2018. Available from: https://jenkins.io/doc/book/pipeline-as-code/ [cited 17 June 2019]. 1, 4

[JKSS17] F. Jacob, I. Karunanithi, P. Salian, and R. Sambhu. Bbc: A dsl for designing cloud-based heterogeneous bigdata pipelines. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1642–1645, Dec 2017. 29, 31, 35

[Kam18] Kamil Mówiński. Escape from merge hell: Why i prefer trunk-based development over feature branching and gitflow [online]. 2018. Available from: https://stxnext.com/blog/2018/02/28/escape-merge-hell-why-i-prefer-trunk-based-development-over-feature-branching-and-gitflow/ [cited 12 October 2018]. 13

[Kry18] Krystian Brazulewicz. Atlassian update – 31 july 2018 [online]. 2018. Available from: https://jira.atlassian.com/browse/BAM-14844?focusedCommentId=1848813&page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel#comment-1848813 [cited 28 December 2018]. 27

[KS08] P. King and C. Smith. Technical lessons learned turning the agile dials to eleven! In *Agile 2008 Conference*, pages 233–238, Aug 2008. 14

[Lau09] Laurel Wainwright. Critical values of the mann-whitney u [online]. 2009. Available from: http://ocw.umb.edu/psychology/psych-270/other-materials/RelativeResourceManager.pdf [cited 31 May 2019]. xiv, 97

[LMP+15] M. Leppänen, S. Mäkinen, M. Pagels, V. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, Mar 2015. 21

[LSS18] K. K. Luhana, C. Schindler, and W. Slany. Streamlining mobile app deployment with jenkins and fastlane in the case of catrobat's pocket code. In *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, pages 1–6, May 2018. 21

[Mar06] Martin Fowler. Continuous integration [online]. 2006. Available from: https://www.martinfowler.com/articles/continuousIntegration.html [cited 20 December 2018]. 15

[Mar18] Mario Fernandez. Modernizing your build pipelines [online]. 2018. Available from: https://www.thoughtworks.com/insights/blog/modernizing-your-build-pipelines [cited 28 December 2018]. 23, 29, 31

[Mat] Matthew Skelton, Manuel Pais. Devops team topologies [online]. Available from: https://web.devopstopologies.com/ [cited 14 June 2019]. 4

[Mug17] Ken Mugrage. It's not continuous delivery if you can't deploy right now. GOTO 2017, 2017. Available from: https://www.youtube.com/watch?v=po712VIZZ7M. 13, 16

[Nee18] Steve Neely. private communication, December 2018. 23

[NIC] NICE Satmetrix. What is net promoter? [online]. Available from: https://www.netpromoter.com/know/ [cited 22 May 2019]. 70, 75

[NS13] S. Neely and S. Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 Agile Conference*, pages 121–128, Aug 2013. 13, 22

[Out16] OutSystems. Outsystems recognized as a leader among low-code development platforms [online]. 2016. Available from: https://www.outsystems.com/news/forrester-low-code-platforms/ [cited 05 May 2019]. 3, 11

[Out18a] OutSystems. Outsystems platform server [online]. 2018. Available from: https://www.outsystems.com/learn/lesson/1338/outsystems-platform-server/ [cited 20 December 2018]. 12

[Out18b] OutSystems. Platform component and tools [online]. 2018. Available from: https://www.outsystems.com/learn/lesson/1309/platform-component-and-tools/ [cited 20 December 2018]. xiii, 12, 13

[Out19] OutSystems. Outsystems awarded 2019 software innovator of the year by dutch it channel [online]. 2019. Available from: https://www.outsystems.com/news/dutch-it-channel-software-innovator-award/ [cited 08 February 2019]. 3, 11

[Sam17] Sam Guckenheimer. What is continuous delivery? [online]. 2017. Available from: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-delivery [cited 23 August 2018]. 16

[Sar19] Sara Staffaroni. What is a good nps score? [online]. 2019. Available from: https://www.getfeedback.com/blog/what-is-a-good-nps-score/ [cited 22 May 2019]. xvii, 75, 76

[SDG+16] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30, May 2016. 1

[Soc] Social Science Statistics. Mann-whitney u test calculator [online]. Available from: https://www.socscistatistics.com/tests/mannwhitney/ [cited 22 May 2019]. 78

[Son15] M. Soni. End to end automation on cloud with build pipeline: The case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 85–89, Nov 2015. 16, 21, 22

[Sta] Statistics Solutions. Mann-whitney u test [online]. Available from: https://www.statisticssolutions.com/mann-whitney-u-test/ [cited 22 May 2019]. 78

[Ste16] Maxfield Stewart. Thinking inside the container- a continuous delivery story. dockercon, 2016. Available from: https://www.youtube.com/watch?v=YViFZBoKqjg. 30, 31, 35

[Swa18] George Swan. Designing autodesk's future today ci/cd for the whole product. All Day DevOps, 2018. Available from: https://youtu.be/UdaeTNtlqm0?t=180. 21

[Sé16] Sérgio Silva. Domain specific language prototyping and design made easy [online]. 2016. Available from: https://medium.com/outsystems-engineering/domain-specific-language-prototyping-and-design-made-easy-9cec69d2d0fc [cited 1 June 2019]. 46

[Tay18] Conor Taylor. Continuous delivery at spotify: Scaling through automation and autonomy. Code-Conf, 2018. Available from: https://www.youtube.com/watch?v=VJIb3qyWoeA. 21, 23, 31

[TTV+17] R. Tim, S. Tanachutiwat, M. Vukadinovic, H. Schlebusch, and H. Lichter. Continuous integration processes for modern client-side web applications. In *2017 International Electrical Engineering Congress (iEECON)*, pages 1–4, March 2017. 14

[Tun18a] Nick Tune. private communication, December 2018. 24, 31

[Tun18b] Nick Tune. Aligning teams and software for continuous delivery. Øredev, 2018. Available from: https://vimeo.com/302680583. 14, 24, 31

[Vis15] Vishal Naik. Enabling trunk based development with deployment pipelines [online]. 2015. Available from: https://www.thoughtworks.com/insights/blog/enabling-trunk-based-development-deployment-pipelines [cited 20 December 2018]. 13

[Xtea] Xtend. Java with spice! [online]. Available from: https://www.eclipse.org/xtend/ [cited 1 June 2019]. 46

[Xteb] Xtext. 15 minutes tutorial [online]. Available from: https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html [cited 1 June 2019]. 46

[Xtec] Xtext. Features [online]. Available from: https://www.eclipse.org/Xtext/ [cited 1 June 2019]. 46, 62

[Yu18] Denise Yu. private communication, November 2018. 29, 31

Appendix A

Questionnaire to calculate SUS

Having in mind the test we just had, please answer the following questions. Test:____________________________
Don't be afraid to hurt our feelings, we are trying to improve and need your help doing it :) !

	Don't Agree 1	**2**	**3**	**4**	**Agree 5**
I think that I would like to use this system frequently.					
I found the system unnecessarily complex. (The system should be simpler)					
I thought the system was easy to use.					
I think that I would need the support of a technical person to be able to use this system.					
I found the various functions in this system were well integrated.					
I thought there was too much inconsistency in this system.					
I would imagine that most people would learn to use this system very quickly.					
I found the system very cumbersome to use. (cumbersome – slow or complicated and therefore inefficient)					
I felt very confident using the system.					
I needed to learn a lot of things before I could get going with this system.					

Figure A.1: Questionnaire to calculate SUS.

Appendix B

Critical Values of the Mann-Whitney U

Critical Values of the Mann-Whitney U
(Two-Tailed Testing)

n_2	α	n_1																	
		3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	.05	--	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
	.01	--	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2	3	3
4	.05	--	0	1	2	3	4	4	5	6	7	8	9	10	11	11	12	13	14
	.01	--	--	0	0	0	1	1	2	2	3	3	4	5	5	6	6	7	8
5	.05	0	1	2	3	5	6	7	8	9	11	12	13	14	15	17	18	19	20
	.01	--	--	0	1	1	2	3	4	5	6	7	7	8	9	10	11	12	13
6	.05	1	2	3	5	6	8	10	11	13	14	16	17	19	21	22	24	25	27
	.01	--	0	1	2	3	4	5	6	7	9	10	11	12	13	15	16	17	18
7	.05	1	3	5	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
	.01	--	0	1	3	4	6	7	9	10	12	13	15	16	18	19	21	22	24
8	.05	2	4	6	8	10	13	15	17	19	22	24	26	29	31	34	36	38	41
	.01	--	1	2	4	6	7	9	11	13	15	17	18	20	22	24	26	28	30
9	.05	2	4	7	10	12	15	17	20	23	26	28	31	34	37	39	42	45	48
	.01	0	1	3	5	7	9	11	13	16	18	20	22	24	27	29	31	33	36
10	.05	3	5	8	11	14	17	20	23	26	29	33	36	39	42	45	48	52	55
	.01	0	2	4	6	9	11	13	16	18	21	24	26	29	31	34	37	39	42
11	.05	3	6	9	13	16	19	23	26	30	33	37	40	44	47	51	55	58	62
	.01	0	2	5	7	10	13	16	18	21	24	27	30	33	36	39	42	45	48
12	.05	4	7	11	14	18	22	26	29	33	37	41	45	49	53	57	61	65	69
	.01	1	3	6	9	12	15	18	21	24	27	31	34	37	41	44	47	51	54
13	.05	4	8	12	16	20	24	28	33	37	41	45	50	54	59	63	67	72	76
	.01	1	3	7	10	13	17	20	24	27	31	34	38	42	45	49	53	56	60
14	.05	5	9	13	17	22	26	31	36	40	45	50	55	59	64	67	74	78	83
	.01	1	4	7	11	15	18	22	26	30	34	38	42	46	50	54	58	63	67
15	.05	5	10	14	19	24	29	34	39	44	49	54	59	64	70	75	80	85	90
	.01	2	5	8	12	16	20	24	29	33	37	42	46	51	55	60	64	69	73
16	.05	6	11	15	21	26	31	37	42	47	53	59	64	70	75	81	86	92	98
	.01	2	5	9	13	18	22	27	31	36	41	45	50	55	60	65	70	74	79
17	.05	6	11	17	22	28	34	39	45	51	57	63	67	75	81	87	93	99	105
	.01	2	6	10	15	19	24	29	34	39	44	49	54	60	65	70	75	81	86
18	.05	7	12	18	24	30	36	42	48	55	61	67	74	80	86	93	99	106	112
	.01	2	6	11	16	21	26	31	37	42	47	53	58	64	70	75	81	87	92
19	.05	7	13	19	25	32	38	45	52	58	65	72	78	85	92	99	106	113	119
	.01	3	7	12	17	22	28	33	39	45	51	56	63	69	74	81	87	93	99
20	.05	8	14	20	27	34	41	48	55	62	69	76	83	90	98	105	112	119	127
	.01	3	8	13	18	24	30	36	42	48	54	60	67	73	79	86	92	99	105

Figure B.1: Critical Values of the Mann-Whitney U (Two-Tailed Testing) [Lau09].

Appendix C

Questionnaire Results

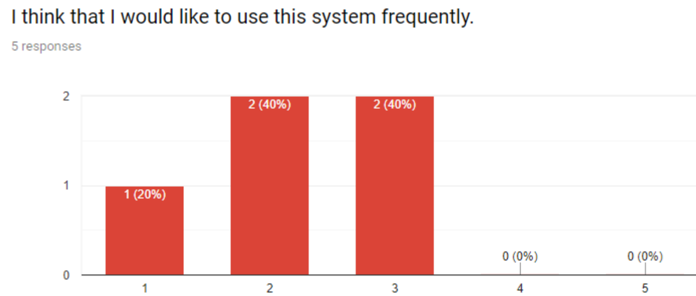


Figure C.1: Responses from control group to the affirmation “I think that I would like to use this system frequently.”.

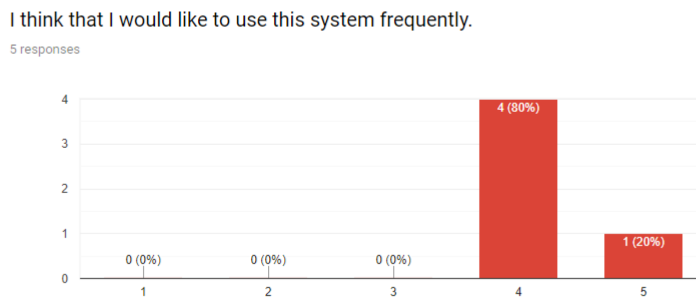


Figure C.2: Responses from experimental group to the affirmation “I think that I would like to use this system frequently.”.

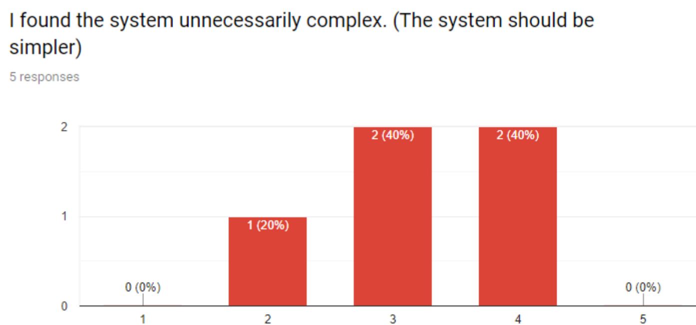


Figure C.3: Responses from control group to the affirmation “I found the system unnecessarily complex. (The system should be simpler)”.

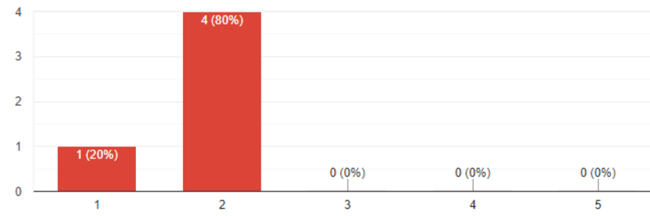


Figure C.4: Responses from experimental group to the affirmation "I found the system unnecessarily complex. (The system should be simpler)".

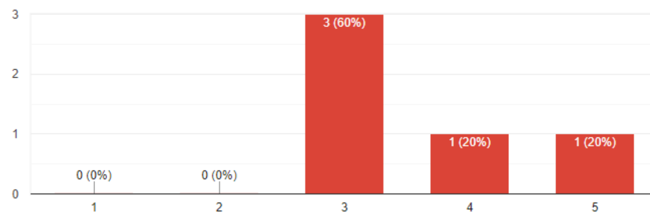


Figure C.5: Responses from control group to the affirmation "I thought the system was easy to use.".

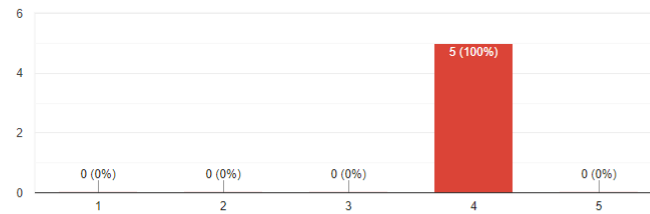


Figure C.6: Responses from experimental group to the affirmation "I thought the system was easy to use.".

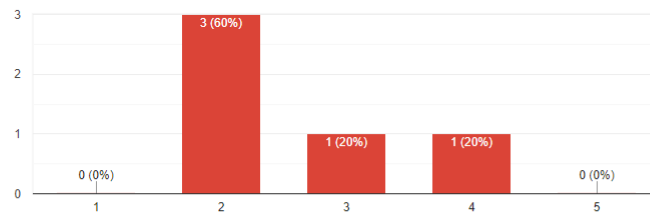


Figure C.7: Responses from control group to the affirmation "I think that I would need the support of a technical person to be able to use this system.".

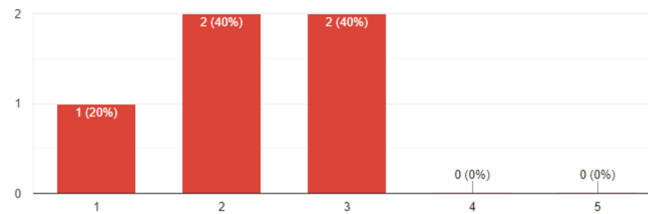


Figure C.8: Responses from experimental group to the affirmation “I think that I would need the support of a technical person to be able to use this system.”.

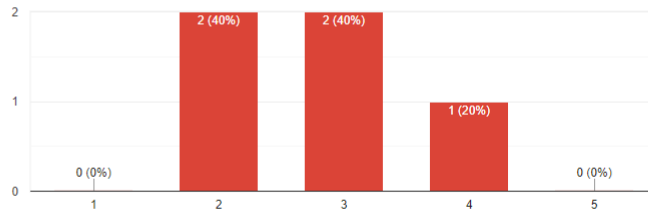


Figure C.9: Responses from control group to the affirmation “I found the various functions in this system were well integrated.”.

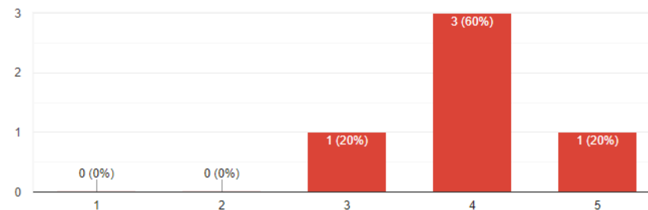


Figure C.10: Responses from experimental group to the affirmation “I found the various functions in this system were well integrated.”.

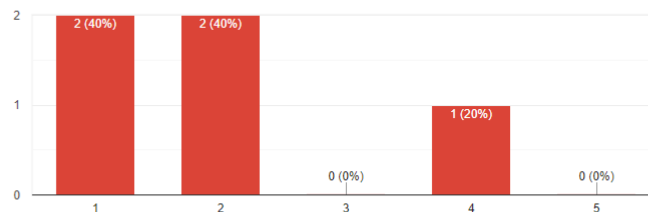


Figure C.11: Responses from control group to the affirmation “I thought there was too much inconsistency in this system.”.

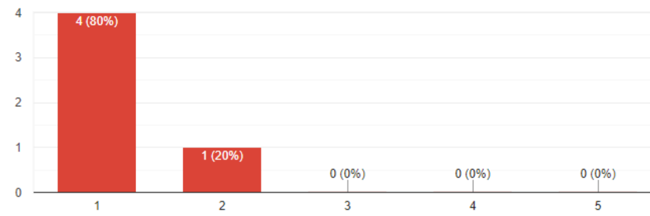


Figure C.12: Responses from experimental group to the affirmation "I thought there was too much inconsistency in this system.".

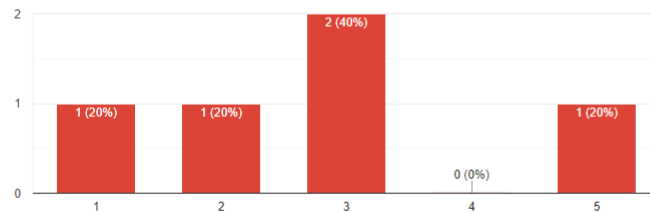


Figure C.13: Responses from control group to the affirmation "I would imagine that most people would learn to use this system very quickly.".

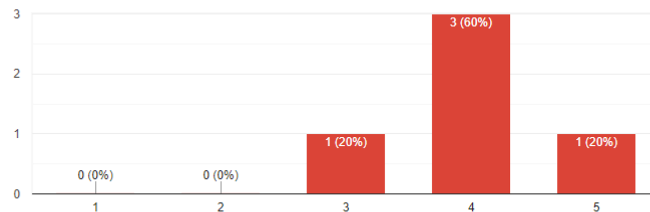


Figure C.14: Responses from experimental group to the affirmation "I would imagine that most people would learn to use this system very quickly.".

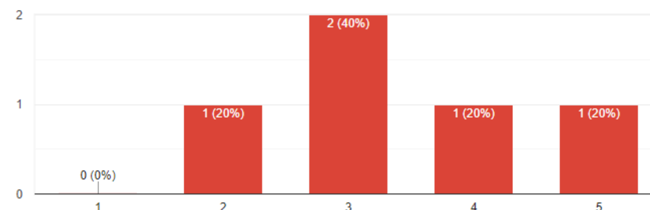


Figure C.15: Responses from control group to the affirmation "I found the system very cumbersome to use. (cumbersome - slow or complicated and therefore inefficient)".

I found the system very cumbersome to use. (cumbersome - slow or complicated and therefore inefficient)

5 responses

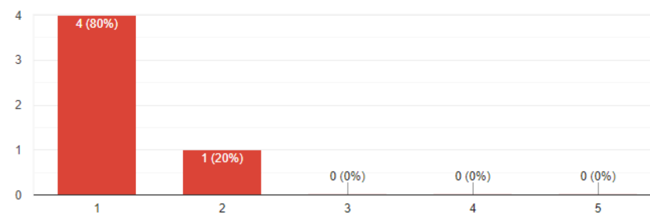


Figure C.16: Responses from experimental group to the affirmation “I found the system very cumbersome to use. (cumbersome - slow or complicated and therefore inefficient)”.

I felt very confident using the system.

5 responses

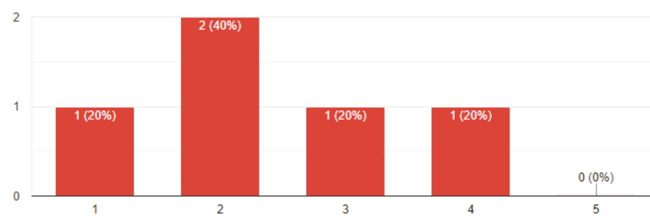


Figure C.17: Responses from control group to the affirmation “I felt very confident using the system.”.

I felt very confident using the system.

5 responses

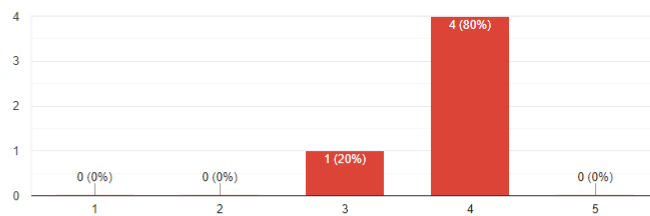


Figure C.18: Responses from experimental group to the affirmation “I felt very confident using the system.”.

I needed to learn a lot of things before I could get going with this system.

5 responses

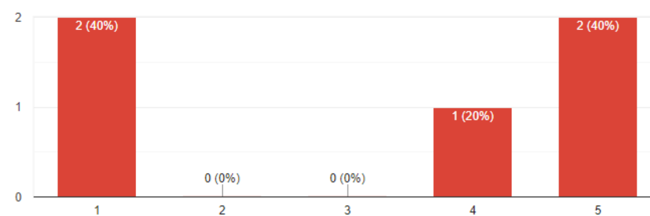


Figure C.19: Responses from control group to the affirmation “I needed to learn a lot of things before I could get going with this system.”.

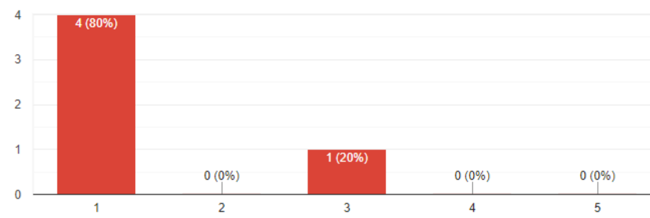


Figure C.20: Responses from experimental group to the affirmation “I needed to learn a lot of things before I could get going with this system.”.

Appendix D

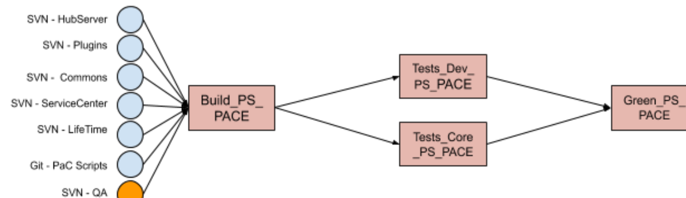
PACE Script

Remember that you can use:

- DSL Documentation
- A library with Build and Tests code. The file is named “library_test_code.pace”.
- A configuration file with several predefined variables, including those that give access to svn
- Online documentation for python
- Ctrl + space for some tips on what you can write next
- Eclipse to develop the necessary code
- The presentation made
- Our great friend google

You will have to perform two tasks during this test, and I need you to tell what you are thinking as you are doing the tasks.

The first task is to make a change to an existing pipeline. Inside the task1 folder you can find the file that defines the pipeline that is represented in the following scheme:

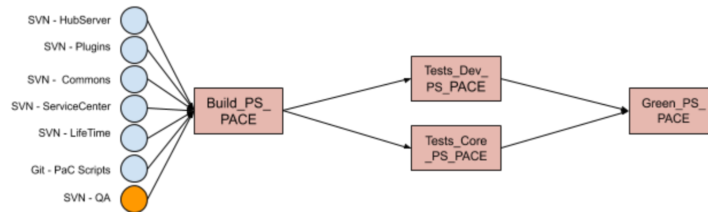


In this image, the blue circles represent the folders that will trigger the pipeline whenever commits are made to files within those folders. The orange circle represents the use of a folder that is present in the repository, but we do not want to trigger the pipeline every time a commit is done.

What you need to do is save a set of artifacts produced by Tests_Core_PS_Trunk_PACE, which are currently not being saved.
The artifacts I want to save now:

- Artifact 1
  - Source: platform/QA/Results/**/*.obtained.png
  - Destination: Obtained
- Artifact 2
  - Source: platform/QA/Results/**/*.obtained.log
  - Destination: Obtained

The second task consists in creating a pipeline similar to the previous.



Remember that you have a library that has Build and Test code for the PS system that can be reused, as well as a file containing several predefined variables. At this time **you can not access the files used in task1,** but you can see the example shown in the presentation.

The goal is to create a pipeline with your name (ex: nelson_fonseca) that is manually triggered, and what each stage presented in the image will do, is described below:

- Build_PS_Pace
    - This stage is responsible for building the platform server, which generates a set of artifacts that will later be used by the remaining stages
- Tests_Dev_PS_Pace & Tests_Core_PS_Pace
    - These two stages are responsible for running the dev and core tests against the platform server that we built

- Green_PS_Pace
    - This stage, in this example, will just fetch an artifact produced by the build that contains the build version and prints the green version on the screen. The name of this artifact is “cintia_ps_dependency.txt”

Appendix E

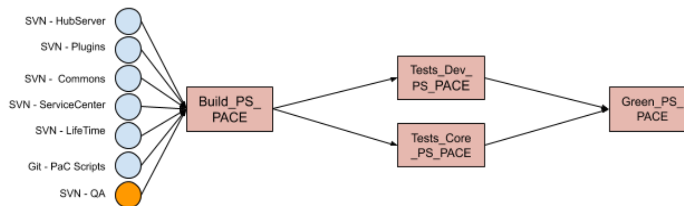
JSON Script

Remember that you can use:

- GoCD and GoCD pipeline as code documentation
- Python scripts responsible for doing tests and build
- A configuration file with several predefined variables, including those that give access to svn
- Online documentation for python
- VSCode to develop the necessary code
- The presentation made
- Our great friend google

You will have to perform two tasks during this test, and I need you to tell what you are thinking as you are doing the tasks.

The first task is to make a change to an existing pipeline. Inside the task1 folder you can find the files that define the pipeline group/system that is represented in the following scheme:

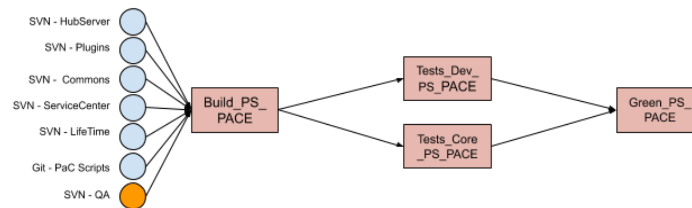


In this image, the blue circles represent the folders that will trigger the pipeline whenever commits are made to files within those folders. The orange circle represents the use of a folder that is present in the repository, but we do not want to trigger the pipeline every time a commit is done.

What you need to do is save a set of artifacts produced by Tests_Core_PS_PACE, which are currently not being saved.
The artifacts I want to save now :

- Artifact 1
  - Source: platform/QA/Results/**/*.obtained.png
  - Destination: Obtained
- Artifact 2
  - Source: platform/QA/Results/**/*.obtained.log
  - Destination: Obtained

The second task consists in creating a pipeline group/system similar to the previous.



Remember that you have a folder with python files responsible for making Build and Tests for the PS system that can be reused, as well as a file containing several predefined variables. At this time **you can not access the files used in task1,** but you can see the example shown in the presentation.

The goal is to create a pipeline group/system with your name (ex: nelson_fonseca) that is manually triggered, and what each pipeline presented in the image will do, is described below:

- Build_PS_Pace
  - This pipeline is responsible for building the platform server, which generates a set of artifacts that will later be used by the remaining stages
- Tests_Dev_PS_Pace & Tests_Core_PS_Pace
  - These two pipelines are responsible for running the dev and core tests against the platform server that we built
- Green_PS_Pace
  - This pipeline, in this example, will just fetch an artifact produced by the build that contains the build version and prints the green version on the screen. The name of this artifact is “cintia_ps_dependency.txt”.

Appendix F

PACE Grammar

Pace.xtext

```
grammar org.xtext.dsl.Pace with org.eclipse.xtext.common.Terminals

generate pace "http://www.xtext.org/dsl/Pace"

Program:
    (elements+=ProgramElements)+
    ;

ProgramElements:
    Import |
    Functions |
    Include |
    Pipeline   |
    Telemetry |
    Variable
;

Telemetry:
    'telemetry' '=' value?=BOOLEAN
;

Functions:
    DecStage |
    DecJob |
    DecFun
;

Variable:
    'var' name=ID ('='  value=STRING)?
;

VariableValue:
    str=STRING |
    var=VariableReference
;

VariableValueStage:
    str=STRING |
    var=VariableReference|
    id=[Stage|QualifiedName]
;

VariableValueJob:
    str=STRING |
    var=VariableReference|
    id=[Job|QualifiedName]
;

VariableValueINT:
    int=INT |
    var=VariableReference
;

VariableReference:
   '#'ref=[Variable];

QualifiedName:
```

Page 1

Pace.xtext

```
    ID ('.' ID)*;

Import:
    FromImport |
    NormImport
;

FromImport:
    'from' name=QualifiedNameWithWildcard 'import' functionName=QualifiedNameWithWildcard ('as'
shortName=ID)?
;

NormImport:
    'import' name=QualifiedNameWithWildcard ('as' shortName=QualifiedNameWithWildcard)?
;

QualifiedNameWithWildcard:
    QualifiedName '.*'?;

Repository:
    'repositories' ':'
    BEGIN
        RepositoryDeclaration
    END
;

RepositoryDeclaration:
    (variable+=Variable | repoComp=Trigger | repoDef+=RepoDef )*
;

RepoDef:
    (repo=REPO) url=VariableValue ('to' locationToSave=VariableValue)? ('login'
username=VariableValue':'password=VariableValue)?
;

Trigger:
    'trigger' (manual=MANUAL |'cron' cronExpression=VariableValue)? ':'
    BEGIN
        def=TriggerDeclaration
    END
;

TriggerDeclaration:
    (variable+=Variable | repoDef+=RepoDef)*
;

Pipeline:
    'pipeline' name=ID ':'
    (BEGIN
        (stage+=Stage | variable+=Variable |repo+=Repository )*
    END)?
;

Stage:
    'stage' name=ID('dependsOn' dependencies+=Dependencies ('&' dependencies+=Dependencies)*)?
':' ( superType=[DecStage|QualifiedName] (  '(' parm+=Parametrs (',' parm+=Parametrs)*')')?
(reuse=REUSE ':')?)?
```

Page 2

Pace.xtext

```
    (BEGIN
        (discardableAgent+=DiscardableAgent | jobs+=Job | variable+=Variable |repo+=Repository |
res+=Resources)*
    END)?
;

Resources:
    'resources' '=' name=STRING
;

DiscardableAgent:
    'discardableAgents' ':'
    BEGIN
        discardableAgentDeclaration=DiscardableAgentsDeclaration
    END
;

DiscardableAgentsDeclaration:
    ('quantity' '=' quantity=VariableValueINT) &
    ('type' '=' type=VariableValue)
;

Job:
    'job' name=ID ('timeout' timeout=VariableValueINT)? ':' ( superType=[DecJob|QualifiedName]
('(' parm+=Parametrs (',' parm+=Parametrs)*')')?  (reuse=REUSE ':')?)?
    (BEGIN
        (fetch+=Fetch |
        run+=Run |
        artifacts+=Artifacts |
        variable+=Variable)*
    END)?
;

Artifacts:
    {Artifacts} 'artifacts' ':'
    BEGIN
        (build+=Build | test+=Test | variable+=Variable)*
    END
;

Build:
    ('build')? artifactLocation=VariableValue ('to' saveLocation=VariableValue)?
;

Test:
    'test' artifactLocation=VariableValue ('to' saveLocation=VariableValue)?
;

Run:
    'run' ('at' location=VariableValue)? ':'
    BEGIN
        pythonCode=PYTHONCODE
    END
;

Fetch:
    {Fetch} 'fetch' ':'
```

Page 3

Pace.xtext

```
    BEGIN
        (fetchDec+=FetchDec | variable+=Variable)*
    END
;

FetchDec:
    artifactLocation=VariableValue 'from' 'stage' stage=VariableValueStage 'job'
job=VariableValueJob ('to' saveLocation=VariableValue)?
;

Dependencies:
    superType=[Stage|QualifiedName]
;

Parametrs:
    value=VariableValue
;

DecStage returns Stage:
    {DecStage}'def' 'stage' name=ID  ( '(' args+=Variable (',' args+=Variable)* ')' )? ':' (
superType=[DecStage|QualifiedName] ('(' parm+=Parametrs (',' parm+=Parametrs)*')')?  (reuse=REUSE
':')?)?
    (BEGIN
        (discardableAgent+=DiscardableAgent | jobs+=Job | variable+=Variable | repo+=Repository
| res+=Resources)*
    END)?
;

DecJob returns Job:
    {DecJob}'def' 'job' name=ID ( '(' args+=Variable (',' args+=Variable)* ')' )? ('timeout'
timeout=VariableValueINT)? ':' ( superType=[DecJob|QualifiedName] ('(' parm+=Parametrs (','
parm+=Parametrs)*')')?  (reuse=REUSE ':')?)?
    (BEGIN
        (fetch+=Fetch |
        run+=Run |
        artifacts+=Artifacts |
        variable+=Variable)*
    END)?
;

DecFun:
    'def' name=QualifiedName ( '(' args+=Variable (',' args+=Variable)* ')' )? ':'
    BEGIN
        pythonCode=PYTHONCODE
    END
;

Include:
    'include' importURI=STRING
;

terminal REUSE: 'extends' | 'override';
terminal REPO: 'git' | 'svn' ;
terminal MANUAL: 'manual';
terminal BOOLEAN: 'true'|'false';
terminal BEGIN: 'synthetic:BEGIN';
terminal END: 'synthetic:END';
```

Page 4

Pace.xtext

```
terminal PYTHONCODE:
    '\'\'\'' -> '\'\'\'';
```