POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

DevOps: development of a toolchain in the banking domain

Supervisor:

Prof. Maurizio Morisio

Candidate:

Matteo Fresco

Academic Year 2020-2021

Abstract

The final goal of this project is the implementation of a DevOps toolchain, through which it is possible to achieve Continuous Delivery, for a software company that develops an Android mobile app for one of the major Italian banking companies. Through the analysis of the state of the art of the DevOps market, in conjunction with the needs and resources of the software company, a toolchain based on tools such as GitHub, Slack, Gradle, SonarQube and most importantly Jenkins has been developed, resulting in the automation of the DevOps steps from the build of the application to its delivery and providing considerable long-term profits in terms of effort and time invested in such process. In order to achieve these results, features of Jenkins such as the Declarative Pipeline syntax and the Pipeline-as-code philosophy have been exploited, facilitating the definition of a Continuous Delivery pipeline by an untrained employee in short time. The successful outcome of the project resulted also in an increase of knowledge of the software company regarding the capabilities of DevOps, making it interested into extending, in the future, the automation mechanism also to other projects.

Contents

List of Figures

List of Tables

Chapter 1

What is DevOps?

1.1 History

In order to understand the true meaning behind the word "DevOps", we must first understand the natural evolution of the approaches that have been used in developing software.

The first realm, the **Waterfall approach**, was popularized in the 1970s: focused on breaking down project activities into a set of phases, sequential and linear, each depending on the completion of the previous one: it is forbidden to pass to the next step until this condition has been verified. The original model, provided by Winston W. Royce, spanned from the creation of a requirement document down to the maintenance of the completed system. Its adoption has been wide-scale thanks to it being easy to manage and clear in its documentation, however, as time went by, its flaws began to arise: not suited for large projects, it made moving back to make changes in the previous phases quite difficult and also, as testing was performed only when development was over, bugs were found only later in the sequential chain, thus making them expensive to fix.

The need for a new method that better suited this constantly evolving, increasingly more dynamic field culminated in the creation of a new philosophy: the **Agile approach**. Born from the frustration of long lead times and incapability of changing decisions made earlier in the project, its main principles were described in a Manifesto published in 2001: a stronger focus on individuals and interactions, deliverable software, customer collaboration and capability of projects being responsive to changes. Since then, many frameworks like Scrum and Kanban started to be defined, polished and adopted by industries leader of the sector, and are still being introduced nowadays by those who strive for change. Agile methodologies, although not exempt from limitations (such as being more difficult than Waterfall to implement and perform correctly), have the merit of having diminished the gap between Customer and Developer.

As new challenges emerged, many began to think if this approach could be further improved: which parts of the IT process needed to be tackled, in order to provide a faster and more robust software delivery model? The answer was in the making, in the form of a new philosophy whose name was coined by Patrick Debois: **DevOps**. Even if its etymology goes back to the *DevOpsDays* conference held in 2009, the term is the result of many different influences [1]:

- In 2007, Debois was working on a data center migration project for the Belgium government: his main duty was to handle relationships between the Application Development teams and the Operations teams (which handled server, database networks and so on). This experience developed in a sense of frustration regarding the lack of cohesion and communication between these two teams.

- In 2008, at the *Agile Conference* in Toronto, Debois took part into a meeting to discuss "Agile Infrastructure" led by Andrew Schafer: sharing their ideas and having common beliefs, together they gave birth to the *Agile Systems Administration Group* on Google.

- In 2009, at the *O'Reilly Velocity Conference*, John Allspaw and Paul Hammond gave their famous presentation "10+ Deploys per Day: Dev and Ops Cooperation at Flickr", aiming to show the conflicts and problems that usually emerge between Development and Operations teams. Debois was unable to attend in person, nonetheless he followed the event by video stream and, inspired by it, decided to form the DevOpsDays conference.

Going forward, this new ideology grew vastly in importance, starting a race to adoption performed by many software companies and also becoming a branch for prestigious minds to work on. In fact, many innovators took DevOps as a "stepping stone", the initial point from which new concepts could be added: embedding security (DevSecOps) or artificial intelligence (AIOps) in the process are only few examples. However, the creation of many buzzwords should not deviate from the original concept that is the soul of DevOps and what it aims to resolve.

1.2 Development vs. Operations

In developing software, companies tend to define teams, in charge of handling distinct jobs, composed of several experts specialized in heterogenous fields. Historically, a major distinction was made between the **Development** team, the "makers" of the product, and the **Operations** team, the "ones that deal with the creation after its birth". Which and how many experts are included in each team usually depends from company to company, but it is an undeniable fact that most companies put the main focus on the development side: the Agile philosophy, which is not yet adopted by many, still does not take into account the delivery and the maintenance part of the product lifecycle, a limitation that DevOps aims to overcome. Therefore, there have been cases where the Operations team was made of a small number of employees or even non-existent: its importance has been highlighted over the course of the years, as the need for a service to be reliable and always available increased.

It is important not to make oversimplifications: the Development teams does not only include the programmers, those who write the actual code, but all the employees who work on the creation of the product: Testers, quality assurance personnel and so on. The Operations team may also include database and system administrators, network technicians, security experts and all those employees who oversee shipping the code into production and also monitor and maintain the product. The shape of these teams depends on the internal strategy of the company: in some cases, the quality Assurance department may not be deemed as necessary, in another one the network may not be under the control of the company. While varying in forms, the main distinction remains.

Development and Operations have ultimately the same goal: the increase of productivity inside the company in order to improve profits. However, the sole existence of these teams leads to a conflict which can be expressed as follows: ***"Development strives for change, Operations fears it"*** (as seen in Figure 1). That is because, while the Development team aims to introduce new features (for example, a more user-friendly UI or a new Database architecture that improves efficiency), bug fixes and all sorts of changes, which at the end must be put into production, the Operations team wants to avoid the introduction of new code, which could undermine the stability, availability, or robustness of the product/service.

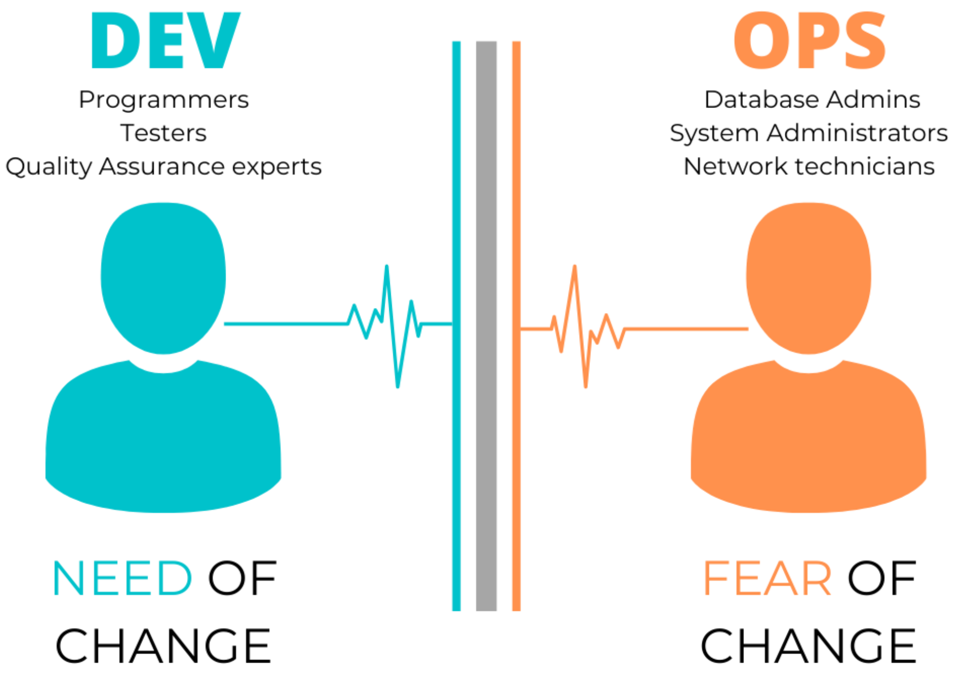


FIGURE 1 - DIFFERENCE BETWEEN DEV AND OPS

The main result of this division is that each team works by trying to improve its own profit, not being interested in communicating with the other one until a new problem emerges. This can be explained with the metaphor of **isolated silos** [2]: each team acts as if its own work was independent from the one of the other team, while in truth they are intertwined and tightly connected. Another concern lies in potential criticalities that may emerge due to this lack of collaboration: for example, it is not uncommon to see Developers claim that their code ran perfectly in a testing environment, while Operations lament the arise of bugs or other kind of problems about the behavior of the software in the production environment. Since both teams think they made no mistake, they may start pointing fingers at each other, a habit called "**Blame Game**", resulting in the creation of a tense and sub-optimal working environment and, most importantly, in a huge waste of time which decreases the company efficiency. If both teams had better communication from the beginning, they could have steadily found the root of the problem (like a minor difference between the testing and the production environment) or avoided the problem at all.

Sometimes, it may even happen that the **Operations team acts as a bottleneck** because a decision has been made of ignoring part of the changes made by Developers: they may indeed hold up on deploying everything new that has been produced if these modifications have been delivered to them near the scheduled time for deploy. The main reason behind this choice could be that the Operations team does not feel any kind of assurance that the newest version of the product provided to them is reliable until it performs checks and tests by itself. And this lack of trust, in most cases, accomplishes nothing but a loss in one of the most valuable software delivery metrics: **Cycle Time**, or "*the total time that elapses from the moment when the work is started on a task until its completion*" [3].

1.3 DevOps Practices

The presence of these inherent conflicts cannot be completely solved due to the nature of the two opposing teams. However, what can be done is minimize their impact: promoting collaboration, trust and the creation of a polite environment, but also maximizing the effectiveness of the methodologies used during all the software development cycle. All these elements lie the foundation of DevOps, which can ultimately be defined as an ideology based on **Culture**, **Processes** and **Tools** (the last of which will be discussed in the State of Art chapter).

1.3.1 Culture

In a company scenario, be it in the field of IT or not, both teamwork and cohesion between employees form the base for success. The human factor is sometimes underestimated, but even the most automation-centered industries have still human employees that work behind the scenes. And when talking about software development companies that work in teams, this aspect surely becomes more relevant. Therefore, Development and Operations must first and foremost align around common visions and goals, which translates into:

- Respect for the work and the opinion of each other
- Commitment to shared goals

- Adoption of shared values
- Shift of focus from roles to competencies
- Idea of collective ownership and responsibility

The last one is undoubtably interesting: the message behind it is an invitation to avoid thinking of the personal work performed on the product as an "individual" work. Each employee, be it part of the Dev or the Ops team, should feel responsible for the quality of the software: thinking "*I have not done it, so it is not my fault*" should be forbidden at all costs.

1.3.2 Processes

DevOps itself cannot be defined in a precise set of processes: its flexibility allows whoever wants to adopt it to choose which practices to implement and which to avoid. Nonetheless, there are still some key patterns that are usually involved and that help viewing DevOps as a Business Process [4]:

- **Systems Thinking**: the performance of the entire system matters the most, more than the individual performance of each department. Consequently, the work from Development to Operations should be delivered in small batches: fewer changes provided at higher frequency.
- **Shift-left Feedback Loop**: teams should focus on preventing problems, rather than detecting them, and developing tests earlier than usually performed [5]. This translates in adopting two DevOps practices, Continuous Testing and Continuous Deployment, in order to shorten test cycles and increase overall reliability of the entire streamline.
- **Systematic Experimentation**: as the modern field of software development revolves around innovation and agility, being dynamic is the key for achieving great advantage in the market. Therefore, this mentality could be adopted through practices such as allocating more time to improve daily work, or reward teams that take successful risks, or even inject faults inside the system on purpose to test resiliency.

Apart from these general practices, it is possible to define several common steps involved in software development and maintenance with a DevOps approach (see Figure 2). First and foremost, the overall process can be thought as a pipeline composed of such phases [6]:

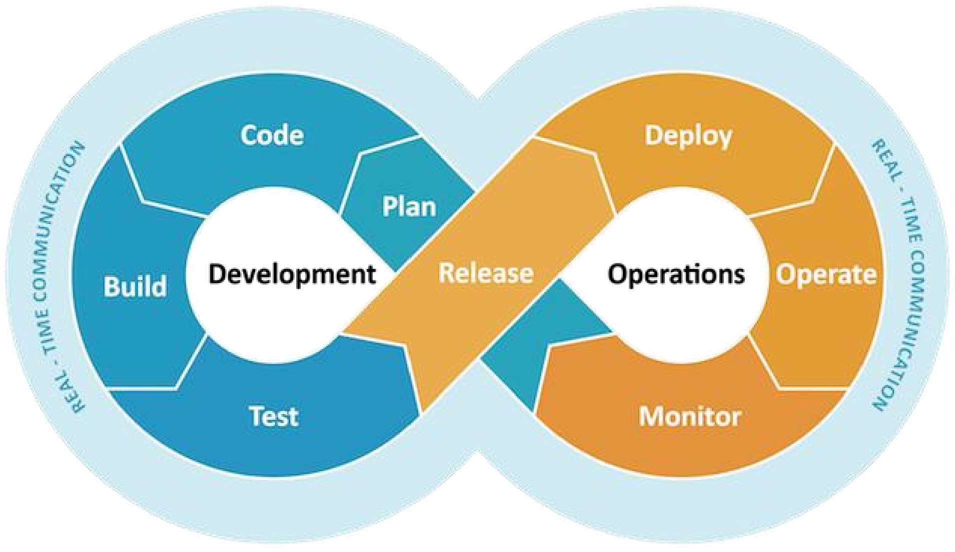


FIGURE 2 – DEVOPS COMMON SOFTWARE CREATION LIFECYCLE

Planning, where customers, stakeholders and product managers define requirements altogether; **Coding**, the phase in which the actual development of the software begins; **Building**, during which the benefits of DevOps emerge: it could be possible, for example, to stop a pull-request requested by a developer in an automatic way, without the need for a manual review done by another employee; **Testing**, performed only in safe environments and if the build succeeds: if automated, not only it would save time, but also guarantee robustness as the liability of human error is avoided. During this phase, it is also possible to set quality parameters that, if not met, may result in an automatic termination of the entire process; **Releasing**, the point in which the build is deemed as ready to be shipped to the production environment. As this step is crucial, a company may decide if it would be wise to automatically ship the build that has met all required standards from previous steps, or if other strategies that require manual intervention are preferred; **Deploying**, the step in which the software is moved from a controlled environment to others: different versions may exist, each subsequent one with more features than the predecessor, but only the older ones may be available to end users. This allows for higher control over business planning, a major factor; **Operating**, where employees such as network supervisors or database administrators check that the overall system works in the intended way; **Monitoring**, during which data and analytics on customer behavior, performance metrics and potential issues are gathered and sent directly to the Development team, to conclude this loop.

The first four steps fall under the Development part of the project, while the last four on the Operations one. Traditionally, all these phases would be conducted in an independent way by each team, without giving to the other team elements and tools to assure stability and reliability. As previously said, this is especially true in the Development team approach: the end-result of its effort, which is the build, may not be guaranteed to meet common requirements. To solve this problem, the idea of **automation** comes into play: by using it, software errors are reduced, many factors such as scalability and productivity are increased, time is saved but, most importantly, reliability is practically guaranteed.

1.4 Role of Automation: CI/CD

There are many theories behind which degree of automation should be introduced, but they mostly fall in two categories: as many supports the "Automate Everything" motto, others believe that it should touch only few key elements of the entire process, therefore the doubt remains on which of the two currents of thought is the ideal one. Undoubtably, striving for the most amount of automation possible should be the main goal. However, according to the second philosophy, removing completely the human factor would highlight the flaws of this process. First of all, a full automated system is not advantageous from a business perspective in a software development/maintenance scenario, due to the fact that agility and dynamicity are major aspects: the cost of changing parts of an automated process is quite relevant, so being forced to modify it would result in an economical loss, rather than growth. Another important factor lies in the *Paradox of Automation*: the more the human component is removed from a process, the more it becomes crucial. This can be easily demonstrated by thinking that, if the automated system manifests an error inside it, the system itself would not be able to identity it. Only an act of human intervention could solve this but, being fewer the parts in which an employee can intervene, they become of utmost importance. Considering these motivations, a trade-off would be the optimal solution.

As the need for automation increased, many innovators started to inject automation on subsequent parts of the DevOps pipeline, starting from the initial stages. However, over the course of time this method has been extended, reaching even the latest phases, the ones handled by the Operations team. And so, the idea of having a "Continuous" flow was born: in the world of DevOps buzzwords, the most common to hear is the abbreviation **CI/CD**, which means **Continuous Integration/Continuous Delivery**. These two practices are key pillars of the DevOps pipeline, allowing to make full use of the automation strategy. It is possible to analyze in depth each term :

- **Continuous Integration**: it is a software development practice involving automatic building and testing performed after developers merge their code to a shared central repository (usually a Version Control System such as GitHub or Bitbucket) at a frequent rate. This practice focuses mainly on the Build phase and it is made of two components: one regarding automation (typically in the form of a CI tool) and a cultural one (the commitment to integrate the code to a VCS more frequently) [7].

- **Continuous Delivery**: it is an extension of the previous practice, which brings the improvement that code changes are prepared to be delivered to a testing or production environment in an automatic way. This enhancement consists in creating a delivery-ready build artifact which has passed a standardized process of testing [8].

Continuous Integration lies down the basis: the cultural aspect behind it tries to solve common deficits in the software development process such as, for example, the merging process performed only when the entire work is completed, a process that becomes difficult to handle and more prone to bug accumulation. From the cultural point of view, Continuous Integration means also adopting a set of sub-practices: developers should work in their private repositories, in order to keep possible failures/errors/bugs as isolated as possible; developers should rebase and check-in frequently the code to keep up with the latest changes approved in the baseline; developers should adopt instant notifications, and so on. On a practical aspect, it can be implemented as follows: once a developer has finished working on a series of small changes, he can push the resulting code to a VCS. Now, thanks to a specific trigger, a CI server (a set of which will be seen in detail in the next chapter) will obtain the latest version of the code from a branch and start to automatically perform the Build and Test phases. If both phases end on a successful note, a build artifact will be generated and possibly the whole team will be informed.

Advantages of this practice translate into improvement of team productivity, higher capability of finding and tackling bugs quickly and having the newest version of the code available to all developers at a higher rate. Another advantage lies in the flexibility of this practice: being mostly a philosophy, in practice it can always be enhanced by adopting tools that could, for example, help achieving higher automation (while remaining careful not to exceed) or better communication.

Continuous Delivery aims to go a step further: by extending automation to the Release step, in the presence of testing/staging environments set up before the real production environment, this practice allows for builds to be delivered to them in shorter time, considering also that this process is always tedious and time-consuming. As many customers would not like to have different versions of the product deployed frequently and directly to the end-user, Continuous Delivery perfectly fits in such scenario, exactly because newest version of the build can be automatically available for the Operations team to perform extra testing and analysis.

Apart from these two core DevOps aspects, which are commonly accepted, people started to define other Continuous patterns, trying to expand the automation process to all remaining DevOps phases. Closing the loop is not an easy task, as the work performed by the Operations team always involves a human factor: this is shown easily by looking at user feedback, a rich component which is not meaningful to automate. Nonetheless, the direction of the market is moving towards that direction, with the creation of new patterns (seen in Figure 3) such as:

- **Continuous Deployment**: in recent times, this practice is trying to take the place of Continuous Delivery in the CI/CD acronym. Fundamentally, it aims to cover also the Deploy step, which means to make the product available to the users after all previous phases have been successful. While typically this step requires manual intervention for the reasons explained above, this practice could still be implemented if the process has been mastered and there are no business reasons to hold it otherwise.

- **Continuous Feedback**: the ideal application of automation in the DevOps process, where data and analytics gathered on the last two steps in the DevOps pipeline are elaborated automatically in a meaningful way for the Development team and sent to it. It is the most ambitious one yet, as gaining feedback cannot be completely put under control due to the unpredictable nature of the users.

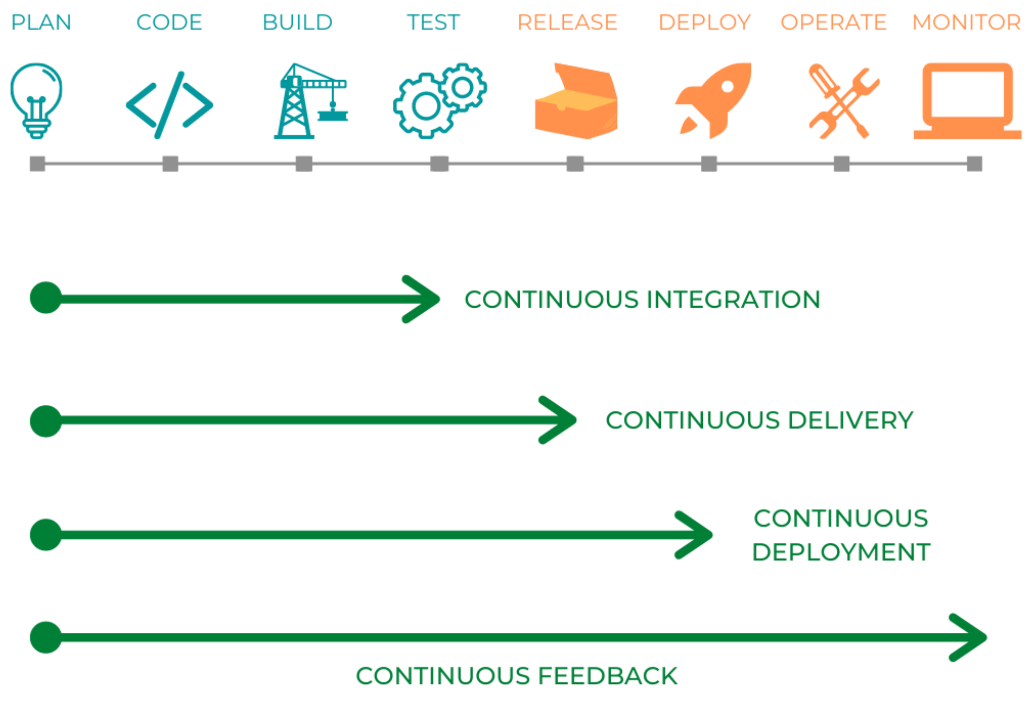


FIGURE 3 - DEVOPS CONTINUOUS PATTERNS

In conclusion, the more a company tries to extend automation, the harder it becomes to handle, as many different layers should be controlled at the same time: tracking development milestones, checking results of code building and testing, spotting code vulnerabilities, managing the health of the infrastructure, checking application logs and user activities. Software companies should not be too eager to automate the loop, but should instead focus on tools, introducing them one by one, learn and fully integrate them in the process.

Chapter 2

State of the Art

2.1 Tools and Market State

It is now mandatory to pay attention towards the third main aspect of the DevOps philosophy: **Tools**, the instruments that help translate the ideology into a concrete mechanism. While it is undeniable that the role of Culture and Processes should not be underestimated, this last component is the one that mostly captures the interest of software companies all around the world. Over the course of the years, starting from the inception of the Continuous Integration and Continuous Delivery market, many competitors started not only to adopt third-party tools, but also to push forward their capabilities of creating such instruments. Amazon, through its IT branch *Amazon Web Services*, has been notorious for its active role in the development and improvement of the DevOps state of art by not only adopting the model, but by creating new tools from scratches. This move, performed by such a big player in the sector, showcases that the production of a single proprietary tool can indeed put a company in an advantageous spot, which translates into a sentence not only valid for this specific case, but for basically all economical fields. Therefore, a race towards the development of the latest, most efficient DevOps technologies has begun, shaking the market and injecting even more dynamicity into an already compelling sector.

Starting from the root mechanism of DevOps, the CI/CD pipeline, innovators wanted to define instruments that could help achieve higher efficiency in all the stages seen in the previous chapter, therefore striving for higher degrees of automation, but also tackling other more human-like aspects such as the need for better project organization, faster means of communication, and so on. The result has been (and even currently is) that while some areas have finally started to settle down, new ones are still being introduced into the mix.

So, trying to understand the current State of the Art of DevOps, it is possible to look at some estimates performed in 2017 regarding the growth of the DevOps market by *Markets and Markets* [9]: the overall size of said market was expected to go from $2.90 billion in 2017 to $10.31 billion in 2023, with a CAGR (Compound Annual Growth Rate) of 24.7% over the course of these 5 forecasted years, with United States being the major player.

Report Metrics	**Details**
Market size available for years	2016-2023
Base year considered	2017
Forecast period	2018-2023
Forecast units	Values (USD)
Segments covered	Solutions, Services, Deployment Model, Organization Size, Vertical & Region.
Geographies covered	North America, APAC, Europe, MEA, and Latin America
Companies covered	CA Technologies (US), IBM (US) , Atlassian (Australia), Micro Focus (UK), Puppet (US), Red Hat (US), AWS (US), Microsoft (US), Google (US), Oracle (US), Cigniti (India), GitLab (US), RapidValue (US), Chef Software (US), TO THE NEW (India), XebiaLabs (US), CFEngine (US), Docker (US), CollabNet (US), Electric Cloud (US), HashiCorp (US), Rackspace (US), Perforce (US), Clarive (Spain), OpenMake Software (US)

TABLE 1 - SCOPE OF THE *DEVOPS MARKET* REPORT

Up until this point, such prediction has been accurate enough: in 2019, *Global Market Insights* has published its own coverage report, involving similar segments, geographies and companies, concluding that the current value of the DevOps market lies around $4.0 billion. This report also provides another prediction, forecasting the period between 2020 and 2026, with the projection of a growth of the market that will reach $17.0 billion in the last year.

With these evidences in hand and taking into account what has been said so far, it should now be possible to focus on which elements of the DevOps pipeline have become the central subjects of the market and also what the market itself has to offer.

Before that, let us first illustrate one of the core keywords of this field, that is the **DevOps toolchain**: as the term "toolchain" itself suggests, it is a "*combination of the most effective tools for developing, delivering and maintaining software according to agile principles*". This definition goes in parallel with the idea of the DevOps pipeline: for each stage, there are one or more tools that can help achieve DevOps principles and that can be chained in order to create a flow. It must be noted, however, that a single tool may also influence more than one stage: it all depends on the impact of the corresponding category that the tool belongs to. Even though this definition may seem quite elementary, setting up a proper toolchain is no easy task. Due to the huge amount of tools belonging to different fields, one of the main problems lies in **compatibility**, as not all tools can be chained together: tool developers with such value in mind make their product suitable for the market, but that is not always the case. Another important aspect is that some products may be available under payment, or present different low-budget versions with limited features while keeping the main ones at high price, so the **cost** to adopt such tools and to setup the toolchain may in some cases be quite relevant. These factors shape two main strategies that a software company may adopt to setup a toolchain [10]:

- **In-the-Box**: an in-the-box toolchain is a solution created by a third-party company, providing a predefined set of standards and tools. This strategy allows to ensure compatibility of tools inside the chain and to save time and effort, but coming at the expense of costs and flexibility.

- C**ustom**: by choosing this strategy, the company selects autonomously its own set of tools, or even creates them, allowing for maximum flexibility and freedom. On the other hand, this method may demand for massive effort and time investment (especially for small companies), or even become budget prohibitive if the project becomes too ambitious.

There is no way to tell a priori which strategy is the most efficient: each company should decide based on their needs and the available budget. In the case of a Custom DevOps toolchain, one last aspect to consider is indeed the amount of tools to which choose from: that is why it is fundamental to have a clear image of the tools panorama and the categories they fall in.

2.2 Source Code Management

As in software development tracking code changes has become hugely relevant, **SCM systems** (also called **Version Control Software**, or **VCS**) have taken the spotlight since the birth of Waterfall processes. The main advantages offered to DevOps teams include empowering team collaboration, management of multiple versions of the same project, assets control and many others: the key aspect is to streamline the software development collaboration process, a huge factor for DevOps.

2.2.1 GitHub

Owned by Microsoft, it is a cloud-based Git server founded in 2008 under the name of *"Logical Awesome"* and it is currently one of the most adopted by companies, with more than 100 million total repositories created and 40 million registered users. Being so popular, it offers the great advantage of having plenty of documentation and wikis, but it also provides many collaboration features aimed to facilitate large open-source projects (Node.js, React Native, Visual Studio Code to name a few projects that use it). Other useful elements are the support of Markdown for issue-tracking, user comments, wikis and so on, and the implementation of a simple user-friendly interface that gives an easier access to the underlying Git features. However, there are some limitations: first of all, GitHub supports only projects that use Git VCS (so, for example, no direct Subversion support: a bridge is needed to communicate `svn` commands to GitHub): another concern may lie in the fact that some GitHub features are locked behind paid versions, but this is balanced by the fact that, since January 2019, it offers free unlimited repositories.

2.2.2 Bitbucket

Property of Atlassian and launched in 2008, it is used by more than one million teams and 10 million registered users, being slightly less popular than the market rival. With due respect to GitHub, Bitbucket does not only support Git (although only since October 2011), but also has historically supported Mercurial (which has been deprecated starting from June 2020). Having started with a complex to navigate user interface, which lead to a decrease in usage by the software community, it has been improved over the course of time, providing now measurements to track efficiency and speed of software development projects. In regards to tool integration, being part of the Atlassian suite of developer tools it offers natural compatibility with instruments such as Jira and Trello, but also third-party tools like Slack, JFrog or even AWS DevOps solutions and Microsoft Azure. About pricing, Bitbucket charges per user, instead of per repository, providing however unlimited public repositories [11].

2.3 Collaboration

It may seem a simple concept, but implementing inside a DevOps toolchain instruments that facilitate fast, clear and customizable communication is a must. Having, for example, the capability of alerting when a Continuous Integration/Delivery process is starting and of notifying what is the result to Dev and Ops teams may end up saving much time and avoid potential misunderstandings and blame games.

2.3.1 Slack

Developed by Slack Technologies and acquired by Salesforce from December 2020, Slack is a messaging platform whose workspaces are divided into channels. Main features of such tool are the presence of a searchable history to track back relevant messages, files or even people, but also the ability to perform voice or video one-to-one calls and, most importantly for DevOps, a high amount of third-party applications that can be integrated with the Slack workspace (in particular Continuous Integration servers which will be seen later).

2.3.2 Others

Apart from the aforementioned instrument, one of the biggest rivals in the category is **Microsoft Teams**: offering conference calls up to 250 people (while Slack provides it under payment) and also a similar concept of Slack channels, that is the chat room feature. However, the strength of this instrument lies in its creator: being owned by Microsoft, it offers great integration with other Microsoft tools like Office 365, providing the ability of combining all tools in one service. Another competitor is **Mattermost**, which aims to impose itself in the market by offering a superset of Slack integrations, especially when talking about webhooks. The choice between these tools, however, may be dictated also by employees preferences regarding User Experience (UX), an important factor not to be underestimated.

2.4 Build Automation

This category may be considered one of the most unrelated, but choosing the right tool to build a project ties strictly with one of DevOps aims, that is time saving. Once a development team has pushed to code to a Source Control Management system, the first step of Continuous Integration lies in compiling and building the code.

2.4.1 Apache Maven

Born in the *Jakarta Alexandria* project in 2001, this Java-written tool has been widely adopted ever since and has a robust workflow defined around it. Maven uses an XML file called *pom.xml* to describe several aspects of the project: build order, plugins needed and even dependencies to third-party modules that help achieving the complete build. Being in the market for so long, the documentation is vast and the support is efficient. It is considered the successor of **Apache Ant**, but it differs from it by being declarative (due to the use of the XML file) and not procedural and by having conventions to place source or compiled code, avoiding the need to provide project information inside the *pom.xml* file.

2.4.2 Gradle

Being the result of many contributors' work, with the first version born in 2007, Gradle is a fully open-source build automation system that expands ideas from Ant and Maven. First of all, its main feature is that it uses a DSL (domain-specific language) based on the programming language **Groovy**, instead of an XML file like Apache did, providing a flexible, simple and short way to create and maintain the build script [12]. The architecture of Gradle can also be defined as based on easy-to-write plugins, but being a recent tool the amount of them is not comparable to the ones Maven offers. The documentation is already defined and explicit and its adoption has begun to be performed by more and more companies. This is especially true in the case of Google, which has made Gradle **the standard build tool for Android projects**. The reasons of that choice can be identified in the same ones that are making Gradle rise over Maven [13]: higher flexibility, better UX and also an improvement of build time. It is evident that this last factor has great impact on the DevOps philosophy, and it is possible to achieve due to three key elements that Gradle introduces: Incrementality (Gradle's ability to track inputs and outputs of tasks and running only the necessary ones), Enhanced Build Cache (build outputs can be reused between builds that have the same inputs) and the presence of a Gradle Daemon (a long-lived process that maintains important build information inside the memory). An immediate comparison can be seen in the graphic below.

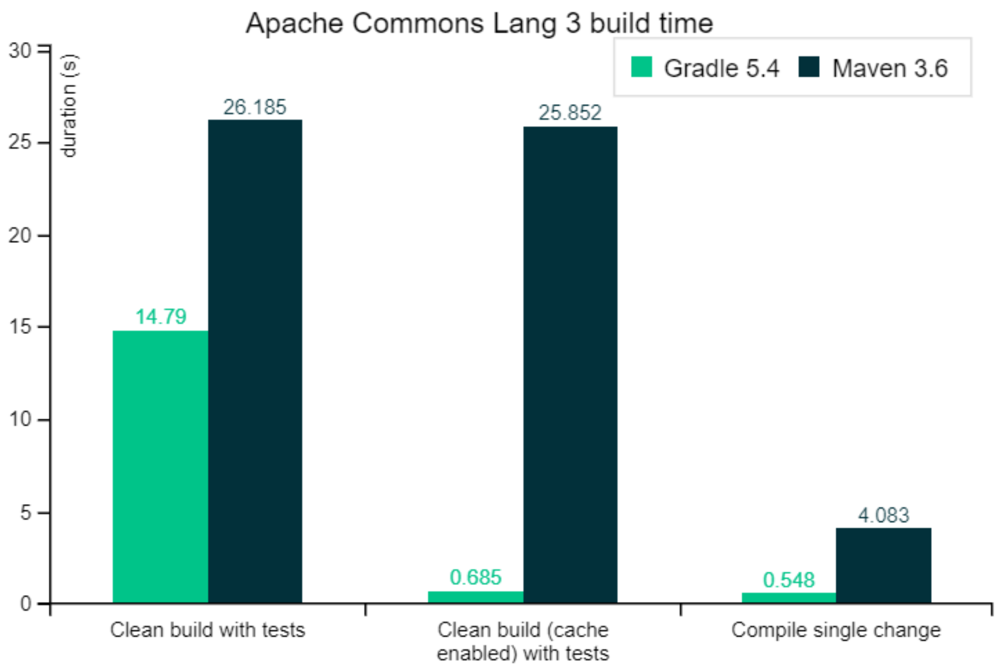


FIGURE 4 - PERFORMANCE COMPARISON BETWEEN GRADLE AND MAVEN

2.5 Continuous Integration

Definitely the most crucial category, as CI tools unite the entire DevOps toolchain by offering methods to orchestrate several stages into a unified instance. These instruments have undoubtably the greatest impact on the market, as the competition is harsh and also involves many big companies like AWS and Microsoft. Another important factor is how these instruments can push forward the Continuous flow: many may offer support to achieve only Continuous Integration, while others may extend their capabilities to support stages like Deployment or Monitoring. That is why, when analyzing this category, it is mandatory to take into even greater consideration previously mentioned factors such as costs, maintainability and compatibility with not only historical tool categories, but also new and emerging ones.

2.5.1 Jenkins

Jenkins is a CI tool considered by many as the leading instrument in the market, due to not only being entirely free (all features are available) and open-source, but also to its popularity and the large community behind it, which helped enhancing the capabilities of such product. This automation server is written in Java and so are its plugins: hundreds of them, developed not only by independent users, but also by companies which officially offer integration with their proprietary tools. A concern that may rise regarding this instrument may be its age: its first version goes back to 2011, when it was released as an alternative variant to **Hudson**, another build server. However, support of new technologies (like Containers) has been achieved indeed thanks to the active work of the community. Other main features are: easy installation and upgrade on various Operating Systems (Windows, macOs, Linux and other Unix-like OSs), support of distributed builds with a Master-Slave architecture, support of shell and Windows command execution, capability to run inside containers or as a standalone service (using Java Runtime Environment), built-in email communication system, presence of environmental variables and a user interface simple and customizable. A deeper analysis of this tool will be performed in the TrustBank Mobile case study, which will be seen in the next chapter.

2.5.2 GitLab

Released in the same year as Jenkins, GitLab strives to provide not only a build server, but a suite to manage the entire software development cycle: it offers its own SCM service, as well as features to implement a CI/CD pipeline, monitoring, security and even elements for early stages such as management and planning. It was built through programming languages Ruby and Go and launched under an MIT license. Being a single source of tools, the main advantage is the embedded compatibility it presents. However, it also provides several features such as integrated issue tracking, easy management of git repositories with access controls and of merge requests, capability to perform code reviews, a better container integration than Jenkins and so on. Nonetheless, GitLab does not provide all of its features available for free and, even though the amount of third-party tools that can be integrated is remarkable, it still does not catch up with the overall compatibility with external instruments that Jenkins offers, or even the flexibility given by its plugins. [14].

2.5.3 CircleCI

With the homonymous company founded in 2011, CircleCI is a cloud-based CI tool, a high-performant solution for automating the Build-Test-Delivery process that is making its rise in the recent times. Its main features are the capability of providing a quick setup, high amount of customization and, most importantly, the fact that it does not require a dedicated server to operate. In fact, by analyzing the differences between it and Jenkins, it can be noted that while Jenkins requires a dedicated server and manual installation of third-party tools and plugins, CircleCI is a cloud-based platform where any updated code is automatically executed in a new container, providing therefore better scalability and less maintenance. In CircleCI developers define all tasks in a single YAML file named *circle.yaml*, therefore only a small amount of information must be encrypted and stored [15]. Overall, this instrument poses itself as an important contender for this category, having been adopted by such a large amount of software companies that each month more than 12 million builds are run. It does not come, however, without limitations: paid versions with extra features are present and the amount of third-party tools to integrate with is still not comparable to Jenkins.

2.5.4 Others

Apart from the aforementioned solutions, there are also a number of relevant alternatives available in the market. Atlassian's proprietary solution, **Bamboo**, allows for parallel automated testing and high visibility into code changes and Jira software issues. It gains the same advantage that Bitbucket provides (although being in an entirely different category), that is built-in compatibility with other Atlassian tools. Another instrument that should be mentioned is **TeamCity**, developed by JetBrains, which provides interesting features like generation of reports involving real-time build progress, the presence of smart templates for reusing build configuration settings and also integrations with VCS and with several IDEs like Visual Studio [16]. Focusing instead on a big player in the IT industry such as AWS, it is undeniable that **AWS CodePipeline** is considered one of the most comprehensive CI solutions, although it is more of a suite of services rather than a single tool. Or even the solution provided by Microsoft, **Azure DevOps**, which is a conjunction of tools, too: both these proprietary suites extend themselves to Continuous Delivery rather than Integration and are the most reliable ones, with the disadvantage of coming with relevant costs.

2.6 Application Delivery

After having setup a CI pipeline, the next natural step would be to add the Delivery phase of the process into the automation flow. Even though some instruments defined in the previous category also allows for Continuous Delivery/Deployment options, it is possible to adopt tools used only to ship the build result of an external Continuous Integration process, exploiting only their distribution feature.

2.6.1 App Center Distribute

Being only a feature of the complete tool that is **Visual Studio App Center**, it allows for developers to release builds to end-user devices. An interesting aspect of App Center Distribute is that it is focused on supporting the Mobile Application field: the task of releasing applications written in programming languages such as Java, Kotlin, React Native or Xamarin and for operative systems like Android and iOS is backed up and well documented. This instrument can be integrated in the DevOps toolchain by receiving an application binary package from an external source, which will be distributed to one of the **Distribution Groups** that can be created and customized: options include adding or removing testers and making the group public, private or shared. However, App Center Distribute does not focus solely on beta testers, but also shows itself as an alternative way to deploy through public application stores like Google Play and Apple App Store [17].

2.6.2 JFrog Distribution

Developed by the IT company *JFrog* (founded in 2008), this instrument is, similarly to App Center Distribute, just a tool of a wider suite that is the **JFrog Platform**. In details, it is a centralized platform that allows the customer to provision software release distribution by allowing configuration of release contents, permission levels and target destinations. Allowing for these and other features, the main benefits of using this tool include the presence of a structured platform to distribute release binaries to multiple cloud or on-premise nodes (although within the same organization), secure deliveries by signing the release bundle, auditing and traceability of all releases. This instrument is available as SaaS (Software-as-a-Service) or in a self-hosted fashion.

Chapter 3

Case Study: TrustBank Mobile

3.1 Introduction

The case of study taken under observation will be the application of the DevOps methodology to a real corporate context. The software company in question, for privacy reason, will be called **SoftGenius**: branch of a major company group, it is specialized in software counseling for companies belonging to the finance and banking fields. The main idea that laid the foundation of this project was to implement the mechanism of Continuous Delivery to their main working app, which for privacy reasons will be called **TrustBank Mobile**. Let us first define the characteristics of such application:

- Being developed by the company for the client TrustBank, a major Italian banking company, the application is no mere toy. In fact, the enormous amount of users means that TrustBank Mobile has to be near bugless and always capable of satisfying thousands of requests in parallel. It can be defined as an **enterprise** application, that is a software solution too large and too complex for individual user or small company use.

- The application is aimed for the mobile devices market, in particular for Android operating systems: at first it was built in Java, but later on the Kotlin language was adopted in order to implement the latest features and to make advantage of the newest support provided by Google.

- TrustBank Mobile is built with a different architecture in mind, that is an architecture based on **microservices**. Rather than the typical monolithic application, built as a single unit and consisting usually of three key elements (a database, a client-side user interface and a server-side backbone), the focus is entirely on the business as each microservice embeds a core business capability.

Therefore, after having defined an overall picture of the application, a question may arise: why is this company interested in introducing a DevOps solution? Or rather, which benefits would the company obtain through the adoption of DevOps? We have seen them in theory, but can these benefits be translated into practice?

It must be first noted that the company is not just approaching this new philosophy: a cultural formation has already been in progress and a deep analysis of processes and tools was also being performed before. Even so, the company wished to push forward its knowledge by investing into this project, the creation of a Continuous Delivery toolchain. Also, an interesting aspect is that the company does not have ownership of the Operations department, which does instead belong to the client company TrustBank: this is due to the business nature of the collaboration between the two companies, as SoftGenius develops the code and advises the client, but ultimately it will be the client himself that will decide when to deploy the app according to its business planification. So, one of the main reasons for implementing a Continuous Delivery toolchain would be to provide builds, that meet quality standards, in an automated fashion to TrustBank in their safe environments (be it testing or not), gaining the clear advantage of reducing the gap between the two companies to which the Development and Operations departments belong to. This factor comes even more crucial in practice due to the context, as there is the need for collaboration not between two teams from the same company, but between teams belonging to different companies. In the end, the execution of this project will be performed to see if the chosen set of tools will prove useful to the needs of the company and if it will be possible to be improved and extended.

Thus, the company strives to enhance their adoption of DevOps starting from the creation of a toolchain. This strategy will also provide improvements to the cultural and processual aspects of DevOps: in fact, this way of thinking is adopted even by big companies such as Amazon, which is explained in the AWS Public Sector Summit (Washington D.C., 2018) [18]. It is based on the idea that a company, be it culturally ready like in this case to apply DevOps practices or not, may *"use Tools to push Culture"*. The introduction of tools for build automation will translate into a heckle for change, driving developers to take more steps towards this new philosophy, to make it their own.

Other advantages are obviously the ones obtained through sheer automation: rapid delivery of builds, security, reliability by removing the possibility of human error that may happen during checks and controls, but also time-saving which will be analyzed in a dedicated section.

3.2 Building the toolchain

With the goal in mind, it is now time to take a step towards the creation of the DevOps toolchain. The first choice is between the two main types of toolchains, Custom or In-the-box, with the decision ultimately falling on the first one. The main reasons are the following: the Custom strategy allows for a slower, more gradual introduction of tools inside the chain. Another point is the high flexibility that this approach allows: not only the toolchain can tackle exactly the needs of the company, but it may also be further improved over the course of time. Last point is the cost: as the company intends to perform an analysis of tools, it would prefer to avoid investing money into a solution that may not fit its requirements or working methods.

The process of setting up a Custom toolchain, however, requires particular attention to the selection of the tools, as compatibility may be indeed a liability. After a long and detailed analysis, the following tools were ultimately chosen:

- **GitHub**: the first instrument must obviously be a Version Control System. Each developer, after working locally on the app, pushes its changes to the remote repository, so the Continuous Integration server will be able to act on the newest version of the code. GitHub was chosen due to its popularity, as many employees were already familiar with this tool.

- **Gradle**: it would be a waste not to use the advantages Gradle brings to the table, especially since it has become the de-facto standard for creating builds of Android mobile applications. In fact, it is the instrument that was chosen by the company for building the application even before having this project in mind: implementing it inside the toolchain is a natural step. Also, through Gradle it is possible to exploit some plugins that allow for the following tools to be directly adopted in the app architecture.

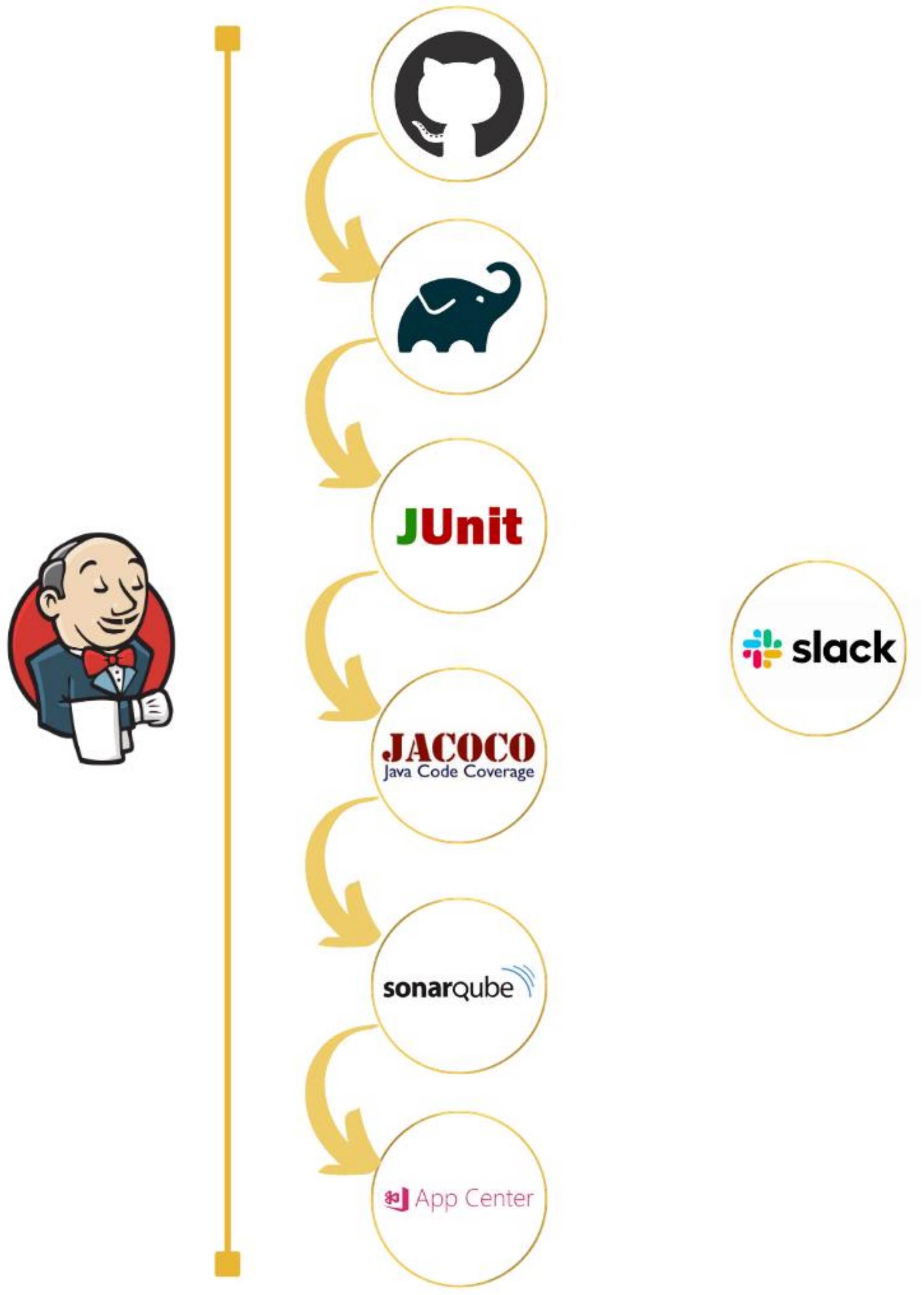


FIGURE 5 - DEVOPS TOOLCHAIN FOR TRUSTBANK MOBILE

- **JUnit**: it is a unit testing framework that works for both Java and Kotlin programming languages. Thanks to it, developers can write and run repeatable tests, which will be automatically performed in our case of study with the definition of a custom Gradle task.

- **JaCoCo**: created by the *EclEmma* team, it is more of a library for Java rather than a DevOps tool itself. However, it comes crucial as it allows to create reports about the *Code Coverage*, a family of measures used to define which is the percentage of code that has been exercised through tests by a test suite.

- S**onarQube**: an open-source platform, already well-established in the field of software development. A software product must not only be tested, but also its code must be analyzed to check for the presence of bugs, code smells, vulnerabilities and other flaws. All these factors define what is indeed the quality of a software: some metrics can be obtained through the inspection of the aforementioned elements, but also the ones regarding code coverage are of immense value. SonarQube allows for continuous inspection of code quality by performing static analysis of the code to detect flaws in it and the result of such action will also define if the code has to be improved furthermore of it meets quality standards defined by the client, being therefore ready to be delivered.

- **App Center**: the final step of the Continuous Delivery pipeline, thanks to its Distribute feature the possibility of storing the result of the build and making it available to a selected group of experts will be achieved.

- **Slack**: a useful addition to the toolchain, its integration results in the automatic creation of messages regarding the status of the pipeline in action. Since communication must be present in many phases, the usage of Slack can be linked and exploited several times inside the flow.

Putting these instruments together tackles all steps going from Coding (as Planning is not considered in this specific case regarding automation) to Delivery. Using them one by one would result in a standard software development process therefore, to automate their usage and to integrate them as a single unit, a Continuous Integration tool acting as server is needed: between all of the previously mentioned ones, the choice ultimately fell to **Jenkins**. Even though it is mentioned as last, it is actually the first instrument that was selected: being all its features free and having the highest amount of third-party tools that can integrate with, it was the choice that better suited the needs of the project. All the aforementioned tools can, in fact, be integrated with Jenkins and can be easily replaced if other tools will be deemed as more fitting.

Now that the tools have been chosen, it is time to setup the environment that will enable for the launch of these automated builds. It must be noted that the project will be performed using a MacBook Pro with the following features:

- Processor: 2.2 GHz Intel Core i7 quad-core
- Storage: 16 GB, 1600 MHz DDR3
- OS: macOS Catalina (version 10.15.7)

3.2.1 Install Jenkins

The first step is to setup a Jenkins instance inside the aforementioned computer. Since a macOS operating system is being used, the official Jenkins documentation provides two ways to perform this action: the first is through a native installer, which however is deprecated; the second (and only) solution is through the usage of a package manager tool called **Homebrew**. Therefore, the first action will be to install Homebrew by running the bash command:

```
/bin/bash-c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

It will now be possible to access the `brew` command from terminal. A prerequisite for Jenkins installation is to have Java installed inside the machine, with the choice falling to the Java Development Kit (or JDK) 11.0.9. It is still possible to use Java 8 or other Java 11 runtime environments, but other versions must be avoided as they are not supported by Jenkins.

There are two main categories of Jenkins versions that can be installed: the **Weekly Release Line** version, that rapidly delivers bug fixes and new features on a weekly cadence, and the **Long-Term Support** version (or LTS for short) which is targeted for more conservative users that prefer less changes and only important bug fixes to be received. Taking into consideration the fact that the instrument is just being adopted, the LTS version will be selected. To install the desired version, the following command must be run in the terminal:

```
brew install Jenkins-lts@MY_VERSION
```

replacing the `MY_VERSION` part with the desired version (in our case 2.249.2).

Now it is possible to start the Jenkins server and setting it to run automatically when the operating system is rebooted with the following command:

```
brew services start jenkins-lts
```

There are some preliminary settings that can be accessed and modified, for example networking parameters such as the IP address that the Jenkins instance is bound to (`--httpListenAddress`, with the default value set to 0.0.0.0) or the port to on which, using the standard HTTP protocol, the Jenkins listener is run to (`--httpPort`, with the default value set to 8080). These initial parameters can be found in the `/usr/local/opt/jenkins-lts/homebrew.mxcl.jenkins-lts.plist` file.

With the default configuration, the server instance will run at port 8080 of localhost. In order for Jenkins to be securely set up by administrators, the developers introduced a password written inside the log or to a specific file located to `/Users/administrator/.jenkins /secrets/initialAdminPassword`. When accessing localhost:8080 for the first time, this password will be asked to unlock Jenkins. After that, the rest of the configuration will be easy and immediate and culminate with the creation of the first user.

Now, let us define some key Jenkins features and how to access them:

- **Manage Plugins**: as it was discussed before, plugins are the primary means to improve Jenkins capabilities. Notably, some plugins may even be dependent on the presence of other plugins, allowing also for automatic installation of them. To access them, from the Jenkins homepage, on the left sidebar, it is possible to click on the "Manage Jenkins" tab. Then, under the section "System Configuration", to select "Manage Plugins" (or, to avoid it, it can be directly found at localhost:8080/pluginManager). Under the "Available" section, it is possible to search for whichever plugin and install it, the remaining sections are self-explanatory.

- **Global Tool Configurations**: some key tools may be configured in order to be used by whichever project may need them. Examples include executables like Ant, Maven, but also Gradle, Git and JDK. From "Manage Jenkins", the next step is to select "Global Tool Configuration" (or directly at localhost:8080/configureTools).

- **Manage Credentials**: as numerous third-party sites and applications can interact with Jenkins, system administrators of these services may configure credentials inside their applications for dedicated use by Jenkins. The Jenkins user will then add or configure these credentials inside the IDE, so they can be used by projects which embed said interactions. To maximize security, credentials are stored in an encrypted form through the usage of AES (in CBC mode with PKCS#5 padding and random Initialization Vectors) and are handled in Jenkins projects via their credential IDs. From “Manage Jenkins”, the user can select “Manage Credentials” (or go directly to localhost:8080/credentials).

- **Configure System**: a miscellaneous configuration page, it contains a variety of sections. From “Manage Jenkins”, the user can then select “Global Tool Configuration” (or go directly to localhost:8080/configure).

It is important to note also that Jenkins can manage distributed builds due to its **Master-Slave architecture**. However, in this project the usage of Slaves is not considered, as the computer defined in the previous section will act as single Master Node and will execute build jobs directly.

3.2.2 Integrate Jenkins with GitHub

To integrate the first piece of our toolchain with Jenkins, it is necessary to install the **Git client plugin**, which provides Git Application Programming Interfaces (APIs) to other Jenkins plugins. After having installed it, it would be wise to check if the corresponding global tool has been properly set inside Jenkins: on the Global Configuration Tool page, under the “Git” section, it is possible to add a new instance of Git installations. Obviously, a clear prerequisite is to have the *command line git* installed on the Master: if so, then it will be enabled by default and an instance with “Name” set to “Default” and “Path to Git executable” set to “git” will be created. If this is not the case, a custom instance can be manually created by giving a new name and by linking the path to the .exe file.

Subsequently, other plugins are needed: one under the name of **Git plugin** (not to be confused with the Git client one) can be installed. This is the actual plugin that will allow Jenkins to perform fundamental git operations such as poll, fetch, checkout, branch, list, merge, tag and push repositories. The final one is the **GitHub plugin**, the one that will allow for the actual integration with the GitHub platform by creating hyperlinks between Jenkins projects and GitHub itself.

From the Jenkins point of view, the last remaining step is to create an instance of credentials, in two different ways. One would be to store the user's GitHub Account credentials by specifying "Username with password" in the "Kind" field. However, it would be preferrable to instead make use of the personal API Token feature that GitHub provides, therefore creating the credential as "Secret Text". Another key feature would be the ability to trigger a build by groking HTTP POSTs from post-receive hooks, which however will be seen later.

3.2.3 Integrate Jenkins with Slack

This time it is necessary to look first at the Slack point of view: this instrument has its own set of applications that can be integrated with it and that can be found in the **Slack App Directory** section. Therefore, the first step of the integration consists in adding inside Slack the **Jenkins CI** app.

For the purpose of this project, a custom Slack workspace has to be created to mimic the behavior of the private one: it can be found at the **trustbankdevops.slack.com** URL. From this, it is possible to go to the Slack App Directory section (which can be found, in this case, at trustbankdevops.slack.com/apps). In the search bar, the next step is to type "Jenkins CI" to find the corresponding app and select it. Then, in the app personal page, selecting the "Add to Slack" option will open a new page. Inside the "Post to Channel" tab, simply choosing the channel where Jenkins notifications will be posted and clicking on "Add Jenkins CI integration" will make this operation successful. Apart from customizable options like the name or the icon that will represent the Jenkins message, a token will be generated.

Now, going back to the Jenkins IDE, the **Slack Notification plugin** must be installed. Afterwards, by going to the Configure System page, we must scroll down to find the "Slack" tab: inside it, the "Workspace" property was set to "trustbankdevops", the "Default channel / member id" property to the newly created channel "#trustbankbuilds" and a new instance of credentials defined, of type "Secret Text" and with the token as value.

This will allow for the two tools to be synchronized in a way that will be exploited later.

3.2.4 Integrate Jenkins with App Center

This is a scenario in which the integration capability has been introduced lately: in fact, the **App Center plugin** has been released in May 2019 and it is still currently in Alpha. For this specific reason, there are not so many features available, even though the ones currently present will be perfect for the ultimate purpose of uploading the build result.

From the Jenkins point of view, apart from the installation of the plugin, there are no settings to be manually inserted or modified. Instead, from the App Center point of view, the action that must be performed is the generation of an App API token, whose usage will be shown later.

3.2.5 Install SonarQube and Integrate with Jenkins

Static analysis of the code is an important step in the DevOps pipeline, therefore the presence of a running server is deemed as mandatory, which will act as a central server capable of holding the results of the analysis. As similarly performed for the Jenkins case, an instance will be installed inside the local machine, with two possible means: through Homebrew or manually, with the ultimate choice falling this time on the second option.

After downloading the selected version (8.5.1) from the official site, it is then possible to extract the SonarQube folder, rename it and move it to the local "Applications" folder for an easier use. It is now possible to make the server running with the following command from terminal:

```
/Applications/SonarQube/bin/macosx-universal-64/sonar.sh console
```

By default the server will be located at port 9000 of localhost, with a default user already created and *admin/admin* as username/password. Even in this case, a token is needed to invoke SonarQube services: it can be defined from the User > My Account > Security page, by entering a name for the token and clicking the "Generate" button.

Now it is possible to perform the integration from the Jenkins point of view: first of all the **SonarQube Scanner plugin** for Jenkins must be installed. Then, by finding inside the Configure System page the "SonarQube Servers" tab, an instance of "SonarQube installations" has to be created, with a new instance of "Secret Text" credentials as seen in other cases.

However, with the current procedure the integration is not complete: what is missing is a tool that actually performs the analysis and that sends the results to the SonarQube server. That is the role that a **SonarScanner** holds in this architecture. Apart from the fact that multiple scanners may run on the CI server, there are multiple versions of these scanners that may be introduced: the one adopted in this project will be the **SonarScanner for Gradle**, which will be in charge of making the analysis after the Jenkins server triggers it.

3.2.6 Install miscellaneous Jenkins plugins

All of the instruments needed have been integrated with Jenkins, so it is now possible to improve the capabilities of the Continuous Integration server itself. The last plugins introduced for the sake of this project are the following:

- **Pipeline plugin**: a core aspect of the project, it is usually already present in the set of recommended plugins that are installed along with Jenkins. In reality, it is a suite of plugins that provides support in implementing and integrating **Continuous Delivery pipelines** into Jenkins.

- **Delivery Pipeline plugin**: a utility plugin, it introduces the visualization of Continuous Delivery pipelines inside the page of a Jenkins build.

- **HTML Publisher plugin**: a plugin that allows to visualize HTML reports, such as those produced during testing tasks, inside the page of a Jenkins build.

- **Git Parameter plugin**: it introduces the capability of making git branch, tag, pull request or revision number as parameters inside a Jenkins build.

- **Parametrized Scheduler**: a plugin that allows to configure a cron-style timer scheduler for parametrized builds. Inside them, it can specify triggers to make them run with different parameters at different times.

With these plugins, the toolchain is set up and ready to be used. It must be noted that all these plugins introduce features that must be discussed, in order to comprehend how Jenkins can be exploited in all of its entirety.

3.3 Jenkins: Core Elements

Jenkins is based on the concept of **Project** (or **Job**), which can be defined as a set of runnable tasks that can be controlled and monitored from the Jenkins UI. Each job can be configured as the user pleases and can be executed as many times as he wants to: the result of each project execution is called **Build**. When a build is run, a directory called **Workspace** is created inside all those computers that act as Master or Slave (which are called **Nodes**) and it will be indeed the working directory used for building. A build is the main point of attention during the whole process, as it can be considered the main unit of work of a Continuous Integration or Delivery pipeline.

In the beginning, the idea of performing sequential tasks was not present in the minds of Jenkins developers, so the main type of job available was the **Freestyle Job**. The word “freestyle” was used in the sense of “improvised” or “unrestricted”, meaning that the order of the tasks was not properly set or defined. Also, Jenkins provided a default interaction model heavily based on user interactions: from the web browser, the user had to manually create jobs and manually configure them through the Jenkins UI. The result of these two elements was extra effort to manage multiple jobs (even in scenarios where they had same configurations) and a separation between the configuration of a project and the actual code. This last point is crucial, as it was impossible to apply CI/CD best practices to the job configuration itself [19]. To improve what was the current Jenkins behavior, the concept of **Pipeline Job** was introduced in order to give a proper shape to these unrestricted tasks and, over the course of time, even the idea of integrating the configuration part into the automation flow became more and more concrete, culminating in the definition of the *“Pipeline as Code”* philosophy.

How could this feature be implemented? The answer was the Pipeline plugin previously mentioned, which has granted the capability to define entire jobs as pipelines and also to store and version them inside a VCS. In this way, the entire Continuous Delivery pipeline can be considered part of the application, therefore being subject to versioning and renewing like the rest of the code. For this purpose, a **Pipeline DSL** (Domain-Specific Language) based on the Groovy programming language was developed, in such a way that its syntax can be used to describe a Jenkins Pipeline inside a file (commonly called a **Jenkinsfile)**, which must be part of the source directory of the project it refers to.

To write a Jenkinsfile, two types of syntax can be used:

- **Scripted**: it is the traditional way of writing the code, as it encourages a more imperative programming model. Being the first one developed, this method is based broadly on Groovy syntax, therefore it is harder for new users to understand and to implement, while on the other hand it provides higher control over the script and higher power of code manipulation.

- **Declarative**: a relatively new feature, it encourages a more declarative programming model. This method imposes limitations to the users by allowing only for a stricter, pre-defined pool of Groovy syntax elements. However, this comes with the advantage of having a richer syntax over the Scripted one (with key elements ready to use) and of making the act of writing and reading Pipeline code easier, saving a huge amount of time.

In both cases, the Pipeline is defined as a set of *blocks*. The most important ones are:

- **Pipeline block**: it describes a user-defined model of a Continuous Delivery pipeline whose code defines the entire build process. This block is the key element of the Declarative syntax and inside of it there are typically stages for building, testing and delivering the application.

- **Node block**: it describes a Node that will execute the pipeline. This block is the key element of the Scripted syntax.

- **Stage block**: it defines a subset of tasks that belong to a specific portion of the Pipeline (for example “Build”, “Test”, “Deploy”). Each stage will be represented by the Delivery Pipeline plugin, allowing to visualize the pipeline in a clearer way.

- **Steps block**: it defines each single task performed inside a Stage. Conceptually, inside this block, there are one or more **Step blocks** in charge of telling Jenkins what to do at a particular point in time, for example the execution of a shell command.

Apart from the previously mentioned differences, many individual syntactical components are shared between the two, so writing in one style or another is mostly a matter of which value is preferred, if flexibility or ease-of-use.

3.4 Creation of a Continuous Delivery pipeline

The subsequent steps will be to create a Jenkins job and to define inside the root directory of TrustBank Mobile a Jenkinsfile. However, before that, there is the need to apply modifications to the TrustBank Mobile project itself.

The previous state of the art of the TrustBank Mobile app was the following: an enterprise Java/Kotlin application for Android OS, which already used Gradle for building and its JUnit plugin for testing. To make use of this aspect, Gradle and JUnit are being incorporated in the DevOps toolchain, thus avoiding undesired changes. It must be noted that Gradle incorporates the concept of **Plugin** too, defined as an element that extends the capabilities of a project by handling a useful set of tasks, such as compiling tasks, setting domain objects or source files and so on. Therefore, by exploiting this feature, it is possible to enhance the TrustBank Mobile project with these two plugins:

- **JaCoCo plugin**: a Gradle plugin that provides code coverage metrics for Java code via integration with JaCoCo. It adds a project extension named *jacoco* of type *JacocoPluginExtension*, which allows to configure JaCoCo usage inside the build.

- **SonarQube plugin**: a Gradle plugin that allows to start a SonarQube analysis of a Gradle project. Thanks to it, the SonarScanner in charge of running the analysis is installed automatically.

The addition of these two plugins is simple: inside the `build.gradle` file regarding the app directory, the following lines were added:

```
apply plugin: ‘jacoco’
apply plugin: ‘org.sonarqube’
```

The syntax used refers to the plugin DSL used by Gradle, exploited in Groovy language (as the `build.gradle` file can also be written in Kotlin), as opposed to the legacy method to apply plugin.

To integrate JaCoCo completely, other actions are required. The first is to introduce inside the same file the following lines, which specify the version of JaCoCo that will be used:

```
jacoco{
      toolVersion = “0.8.5”
}
```

Another property that can be set is “reportsDirectory”, so that the location of the report that will be generated can be customized. However, in this case its default value will be used, that is `$buildDir/reports/jacoco/test`, where *$buildDir* is a variable that automatically translates to the build output directory. These steps already allow to implement a **Gradle Task** (that is, an atomic piece of work which a build performs) named **JacocoReport**: its main duty is to generate indeed the code coverage reports in different formats (XML, CSV and HTML). Nonetheless, TrustBank Mobile source code is not suited to be analyzed directly by that task, therefore a custom task must be created, which will be named **jacocoTestReport**.

Without focusing too much on the complete source code (which can be fully seen in Appendix A), let us analyze what this task accomplishes. First and foremost, it is defined as dependent on the Gradle task that will be used for testing, as obviously the report cannot be generated if tests are not executed beforehand. Once the testing task is executed, a *.exec* file will be created inside the jacoco directory: this is directly referenced in the “executionData” property. The “reports” property is configured in order to obtain an HTML report and an XML report (also renamed as `jacocoTestReport.xml)`. Being careful to exclude files and classes not useful for the coverage report, it is important to define in the “classDirectories” properties both Java and Kotlin classes created by the testing task.

Now, let us shift focus onto SonarQube. The ability to execute a SonarQube analysis via a regular Gradle task makes it available anywhere Gradle is available, without the need for manually setting up a SonarScanner in the local computer. In fact, the Gradle build already has much of the information needed for SonarQube to analyze a project, therefore modifications applied to the existing files are highly reduced. In fact, it is enough to add the following lines to the same `gradle.build` file mentioned before:

```
sonarqube{
 properties{
  property “sonar.projectName”, “TrustBank Mobile”
  property “sonar.host.url”, “http://localhost:9000”
```

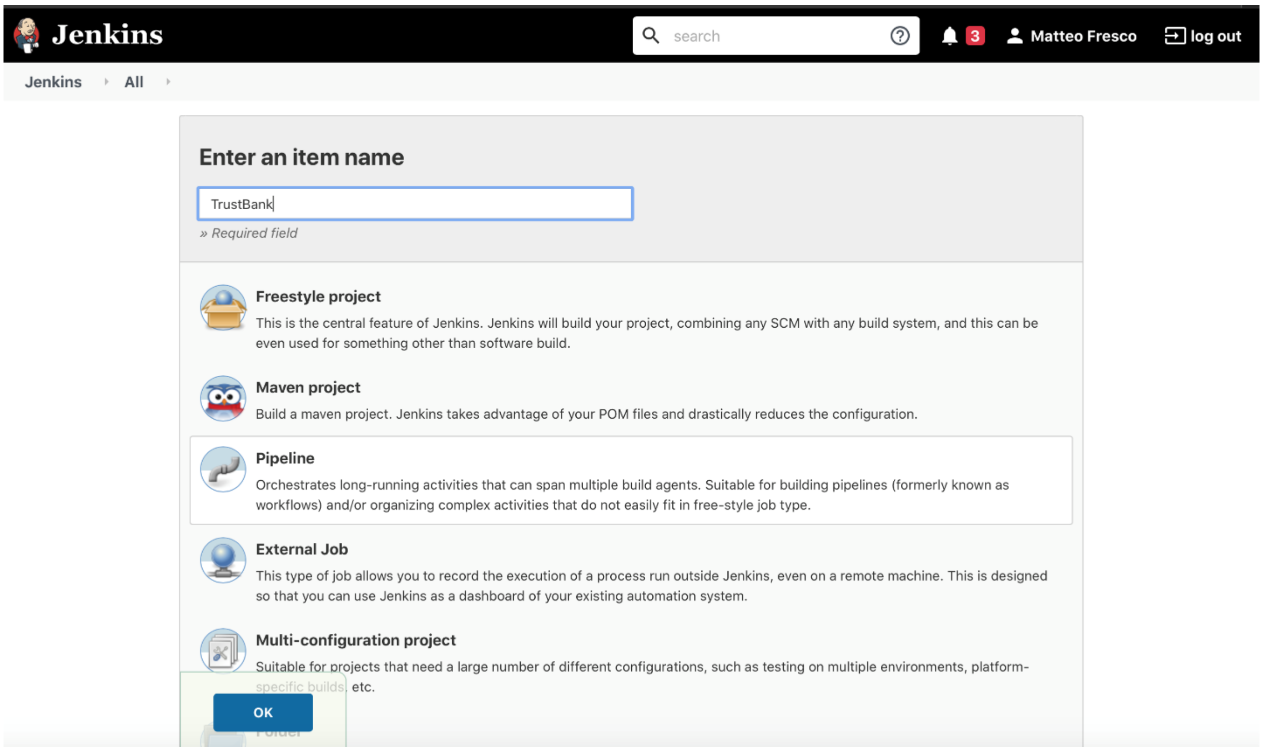
```
   property “sonar.host.login”, “MY_SONAR_TOKEN”
   property “sonar.exclusions”, “MY_FILES_TO_EXCLUDE”
   property “sonar.java.coveragePlugin”, “jacoco”
   property “sonar.coverage.jacoco.xmlReportPaths”, “MY_REPORT_PATH”
  }
}
```

While the addition of the plugin already allows to use a Gradle task called **sonarqube**, these properties will define the name of the SonarQube project that will be created and which files are to be excluded from the static code analysis. There is also the need to properly define which type of coverage plugin Sonar should check for: being in this case JaCoCo, another property is needed to define the exact location of the XML report produced. It is important to also declare the location of the SonarQube server and the token created beforehand for the integration. As it can be seen, the token will be inserted in plain along with the source code, which may be considered a drawback, however the security measures used by the company to keep the source code of the project private will also conceal the content of the token.

Now that both JaCoCo and SonarQube are integrated in the toolchain, it is time to finally create the Jenkins job. Taking into consideration the concepts introduced in the previous section, a decision has to be made about which shape should be given to the project. Exploiting the recently introduced Pipeline feature instead of the definition of a Freestyle job is a natural decision.

To create it, the user can simply select in the left sidebar the tab “New Item”: as seen in Figure 6, a new page will be opened, so that the user can insert the name of the project and select “Pipeline” as job typology.

Now, the doubt remains on which syntax must be chosen. Even though the Scripted syntax offers higher capabilities, for a user lacking experience with the Groovy language ease-of-use is deemed as imperative, therefore the Declarative style is preferred. The first step is the creation of a Jenkinsfile inside the project directory, which must be committed and pushed to the proper GitHub repository. The entire content of the file can be seen in Appendix B, but it will still be dissected and analyzed, in order to fully comprehend each step.



Enter an item name

TrustBank

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

External Job
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

FIGURE 6 - CREATION OF TRUSTBANK JENKINS JOB

Since the Jenkinsfile is written through the Declarative style, the first block is of course the Pipeline block, inside of which the first line of code is the following:

```
agent any
```

indicating that the Jenkins job will be run on any available Node. Then, the following:

```
parameters {
 booleanParam(name: ‘PERFORM_STATIC_CODE_ANALYSIS’, …)
 booleanParam(name: ‘DELIVER_TO_PRODUCTION, …)
}
```

It defines that this job can be run as a **Parametrized Build**. This feature is extremely useful, introducing a dynamic component that can decide how the build should be performed. For instance, the software company may decide to run experimental builds in order to test some new features or how the pipeline behaves if the execution of certain steps is omitted. In this case, a consideration is made regarding the exclusion of static code analysis or build output delivery. These parameters, however, are set by default as true, since the main build includes these steps.

After that, there is the possibility to define a **Triggers block**, which defines certain conditions under which the execution of a Jenkins build is started without the need for manual input. In conjunction with the Parametrized Scheduler plugin, the result consists in the following lines:

```
triggers{
  parametrizedCron(‘‘‘
     H/30 18 * * *
  ‘‘‘)
}
```

In this case, the **ParametrizedCron block** allows the user to define the following: each day, at 6:30 pm, a build of the corresponding job must be executed automatically. This is due to the built-in cron trigger and the corresponding syntax used in the cron lines. For instance, the notation defines 5 variables that represent, in order, minute (0-59), hour (0-23), day of the month (1-31), month (1-12) and day of the week (0-6, Sunday to Saturday) of the time they describe. The value of each variable can be an appropriate number, therefore not exceeding the ranges defined above, or the wildcard ‘*’ that means “any”. These parameters will also be accessible through the “Configure” tab in the page of the TrustBank project we have created inside the Jenkins IDE.

After having defined the preliminary components, it is time to create the actual blocks of our pipeline. Therefore, inside a Stages block we can define each and every stage, starting from:

```
stage(‘Slack – Notify Start Build’){
 steps{
  slackSend channel: ‘#trustbankbuilds’,
   message: ‘Started ${JOB_NAME}’ with build #${BUILD_NUMBER}'
 }
}
```

It must be noted that a Stage block has an argument inside the round brackets, that represents indeed its name. This first stage consists in making the Jenkins server send a custom message to a specific Slack channel, allowing to have a quick notification when a build is started. This behavior is exploited due to the keyword *slackSend*: it represents the corresponding step that will perform the aforementioned action and it is available in the Declarative Pipeline syntax through the introduction of the Slack Notification plugin used for the integration, a mechanism that will be used by other plugins. The arguments are self-explanatory, although they are not the only ones available.

An interesting factor is the presence of the two variables ${JOB_NAME} and ${BUILD_NUMBER}, whose values are self-explanatory: these are **environment variables** and originate from the **Jenkins Environment Variable**, a global variable exposed through the *env* variable and accessible anywhere in the Jenkinsfile. More specifically, the *env* variable can store inside of it values of type String, which are indeed the environment variables (so, the JOB_NAME variable represents, in reality, the variable env.JOB_NAME). There is a predefined set of them which can be found, in this case, at the location localhost:8080/env-vars.html, but there is also the possibility to define custom environment variables inside the Jenkinsfile: this can be done globally, inside an **Environment block**, or locally per stage.

The next three stages are described as following:

```
stage(‘Gradle – Clean Project’){
 steps{
  sh ‘./gradlew MY_CLEAN_TASK’
 }
}

stage(‘Gradle – Build Project’){
 steps{
  sh ‘./gradlew MY_BUILD_TASK’
 }
}

stage(‘Gradle – Perform Testing’){
 steps{
  sh ‘./gradlew MY_TEST_TASK’
 }
}
```

Each one executes a shell command that starts one of the usual tasks that were performed by the company during the previous cycle of “development-building-testing”. Taking into consideration that the project refers to an enterprise application, the usual Gradle tasks for these steps are not used, but they are instead replaced by custom versions developed by the company itself. A consideration must be made regarding the presence of the clean task: the workspace of the TrustBank Jenkins job is reused for each successive build, therefore build and test outputs are saved and need to be addressed. Also, it should be noted that the MY_TEST_TASK is the same task as the one the jacocoTestReport task depends on. Another key aspect is the fact that these tasks are executed through a **Gradle Wrapper**, a script that invokes a declared version of Gradle and that downloads it, if necessary. The advantage of using it is that it is located in the root directory of the project, avoiding the need of setting up an instance of Gradle inside the Global Tool Configuration page. If a user prefers to perform the latter action, it would still be possible, however the Gradle version of Jenkins should be manually updated each time the Gradle version used for the TrustBank project is changed: it is clear that using the Wrapper avoids this tedious task.

At this point, the project is able to produce build and test outputs, so it is time to perform JaCoCo report creation and SonarQube analysis. This is done in the following steps:

```
stage(‘JaCoCo – Create Code Coverage Report ’){
 when{
  expression { params.PERFORM_STATIC_CODE_ANALYSIS }
 }
 steps{
  sh ‘./gradlew jacocoTestReport’
 }
}

stage(‘SonarQube – Static Code Analysis’){
 when{
  expression { params.PERFORM_STATIC_CODE_ANALYSIS }
 }
 steps{
  sh ‘./gradlew sonarqube’
 }
}
```

The Gradle tasks performed are the ones defined previously. What must be noted is the presence of the **When block**: it allows the Pipeline to determine if a stage should be executed or not depending on a given condition. There are several built-in conditions (for example, some depend on the git branch which the job refers to, others depend on the environment and so on), but the one used in this case is the **Expression** one: it executes the stage if the specified Groovy expression evaluates to true. Here is the place where the parameters defined before are used: they can be accessed through the *params* variable, which differs from the *env* variable, since it contains all build parameters, instead of environment variables.

Now, the last remaining stage is defined as follows:

```
stage(‘App Center – Publish Build){
when{
  expression { params.DELIVER_TO_PRODUCTION }
 }
 steps{
  appCenter apiToken: ‘MY_APPCENTER_TOKEN’, …
 }
}
```

Thanks to the App Center plugin, the keyword *appCenter* is available to start the homonymous step. The parameters used are mandatory, although there are also another set of optional ones. First and foremost, the App Center API Token is required: by writing it in the Jenkinsfile, it would result as plaintext, however the same security considerations made previously for the SonarQube token are valid. Then, other parameters define the name of the owner and of the corresponding application created in App Center, the location of build output and the name of the Distribution Groups where it will be shipped.

With the Stages block ending, the flow of the Pipeline could end as well. However, there is the chance to exploit post-build actions through the **Post block**: inside of it, it is possible to introduce actions depending on the build result, or actions that must be performed regardless of the end result. This behavior is exploited thanks to three blocks named **Always**, **Success** and **Failure**, which are used as following:

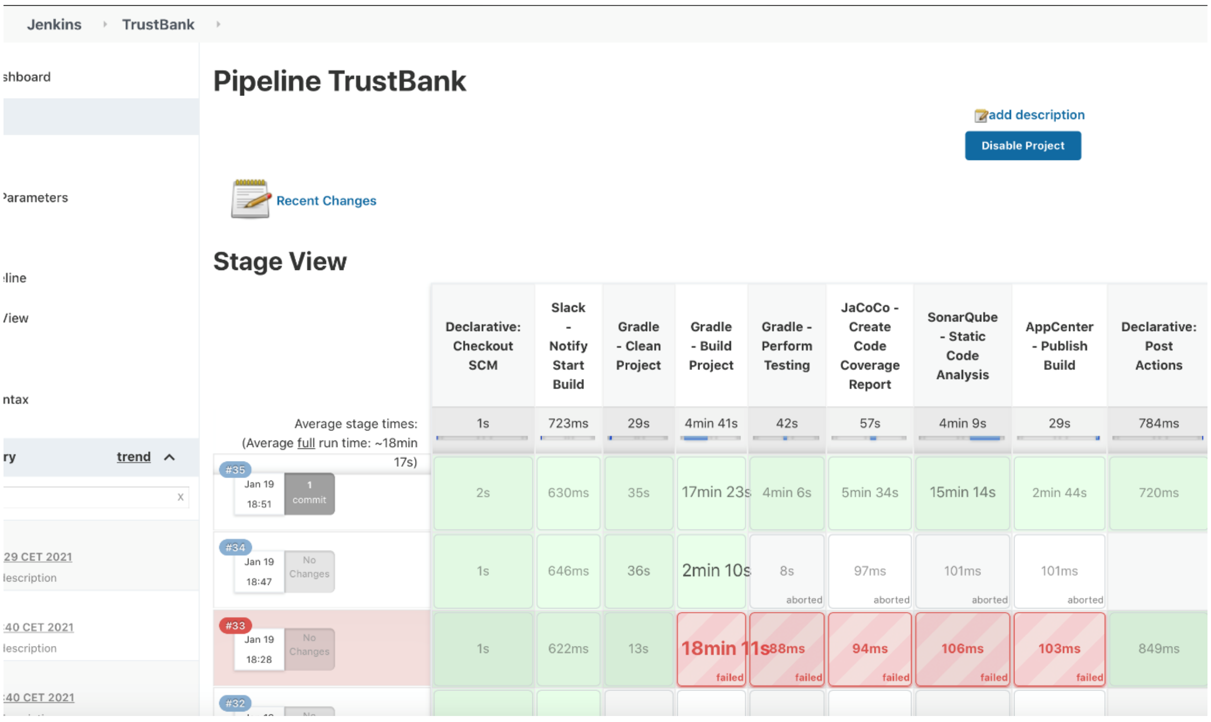
```
always{
 publishHTML (target: …) …
}
success{
 slackSend channel: ‘#trustbankbuilds’, …
}
failure{
 slackSend channel: ‘#trustbankbuilds’, …
}
```

In this case, Slack is used once again to deliver messages, this time regarding the final result of the build. Independently of the result, however, the HTML Publisher plugin is used in order to have an HTML report of the tests that were performed through the MY_TEST_TASK Gradle task inside the corresponding page of the Jenkins job. In fact, this plugin allows to use the *publishHTML* keyword to start the corresponding task, which however requires a mandatory set of parameters: aside from the ones related to the name of the page and the title of the report that will be published inside the Jenkins server, the two parameters “reportFile” and “reportDir” involve respectively the name of the produced HTML file and the directory where it is located. The last three parameters refer to the possibility of making the build fail if the report is not found inside the directory (“allowMissing”), to the feature of archiving reports for all successive builds (“keepAll”) and to the possibility of being linked to the report of the last build, regardless of the build final status (“alwaysLinkToLastBuild”).

This marks the end of the Jenkinsfile, which now completely describes the desired Pipeline. Committing and pushing it to the GitHub repository of the TrustBank project is the next natural step: from this moment, every change or update to the Continuous Delivery pipeline will simply be performed by modifying the Jenkinsfile accordingly. The last remaining phase will be to properly configure the TrustBank job inside the Jenkins IDE, the only user interaction required: starting from the homepage, the job can be selected and, in the left sidebar of the redirected page, the “Configure” tab is available.

Inside this section, it is important to configure certain elements in the “Pipeline” tab. First of all, the “Definition” parameter defines which will be the source of the Pipeline script, allowing two choices: “Pipeline”, which grants the user the capability of writing the Pipeline script directly in this section, or “Pipeline script from SCM”, which instead exploits even more the Pipeline-as-Code philosophy. Choosing the second option will require to specify which SCM system is being used (in this case, the value will be “Git”), the Repository URL, the Credentials used to access it (the ones defined before during the GitHub integration), which branch must be built and the location of the Jenkinsfile inside said repository.

With this final setup, the build is ready to be run. Even though a cron scheduler has been implemented to automatically execute it, it is also possible to manually run it by selecting the “Build with parameters’ tab inside the current job page. During the execution, thanks to the Delivery Pipeline Plugin, a visual representation of the stages we have defined is shown (as seen in Figure 7), which allows to see the status of the current stage of the build that is being performed: each block representing a stage will ultimately be represented through a color that indicates its final result, so that it is possible to see immediately which stage failed, in case.



Pipeline TrustBank

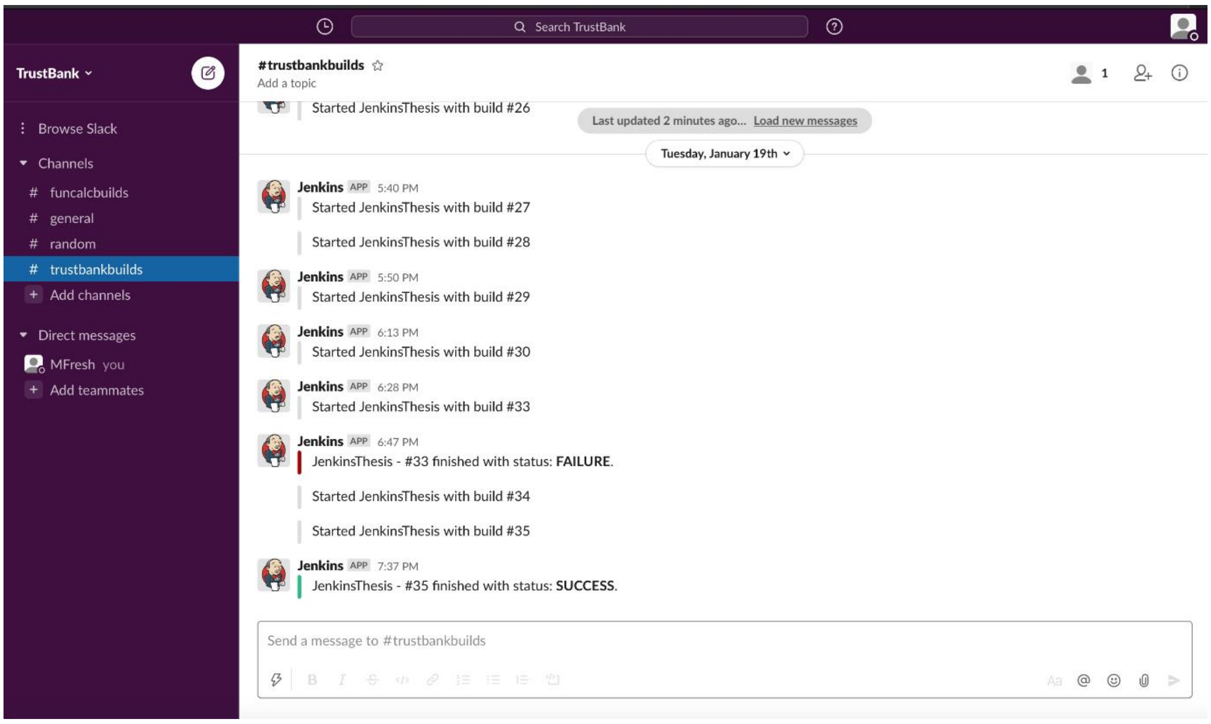
Stage View

	Declarative: Checkout SCM	Slack - Notify Start Build	Gradle - Clean Project	Gradle - Build Project	Gradle - Perform Testing	JaCoCo - Create Code Coverage Report	SonarQube - Static Code Analysis	AppCenter - Publish Build	Declarative: Post Actions
Average stage times: (Average full run time: ~18min 17s)	1s	723ms	29s	4min 41s	42s	57s	4min 9s	29s	784ms
#35 Jan 19 18:51 1 commit	2s	630ms	35s	17min 23s	4min 6s	5min 34s	15min 14s	2min 44s	720ms
#34 Jan 19 18:47 No Changes	1s	646ms	36s	2min 10s	8s aborted	97ms aborted	101ms aborted	101ms aborted	
#33 Jan 19 18:28 No Changes	1s	622ms	13s	18min 11s failed	88ms failed	94ms failed	106ms failed	103ms failed	849ms
#32									

FIGURE 7 - PIPELINE VIEW OF TRUSTBANK JENKINS JOB

The Jenkins IDE allows also to manually abort a build that is currently being executed: this status will also be visually represented. It must be noted that each time a build is performed, a new page related to it is accessible and it generates logs that can be seen in the "Console Output" tab of said page. The overall generated log is divided into portions, each one referring to a single stage, and it is possible to access them when hovering over the visual block of said stage.

In the visual representation, there is the presence of two stages under the "Declarative" name, in particular the first of which is also not defined in the Jenkinsfile. They are both automatically generated by Jenkins, which recognizes that the Declarative syntax is used and therefore updates the pipeline accordingly. For the initial stage, **Checkout SCM**, Jenkins simply adds to the Pipeline a stage made of a single step, described by the keyword *checkout scm*. While by default this operation is carried out as a simple checkout, it is also possible to customize this process when needed. After that, each stage will be carried out as declared, with the addition of the communication mechanism introduced by Slack (Figure 8) to keep the employees updated with the status of the current build.



Started JenkinsThesis with build #26

Jenkins APP 5:40 PM
Started JenkinsThesis with build #27
Started JenkinsThesis with build #28

Jenkins APP 5:50 PM
Started JenkinsThesis with build #29

Jenkins APP 6:13 PM
Started JenkinsThesis with build #30

Jenkins APP 6:28 PM
Started JenkinsThesis with build #33

Jenkins APP 6:47 PM
JenkinsThesis - #33 finished with status: **FAILURE**.
Started JenkinsThesis with build #34
Started JenkinsThesis with build #35

Jenkins APP 7:37 PM
JenkinsThesis - #35 finished with status: **SUCCESS**.

FIGURE 8 - TRUSTBANK SLACK CHANNEL

3.5 Metric Analysis

With the successful establishment of a Continuous Delivery pipeline, the gap between Development and Operations departments will surely be reduced. However, having seen the advantages only from a theoretical point of view, it is now time to analyze in detail what benefits have been gained also from a practical side. Historically, traditional metrics have always emphasized the importance of measurement as a tool for tracking progress and identifying the current status of a project or a particular process. Referring to a subject as wide and complex as DevOps, however, there is no single, universal metric that exists as the sole indicator of success: it is important to analyze the entire process and to find which metrics better suits its analysis. In order to do so, it is mandatory to first define schematically the entire Delivery flow that is being performed, which can be seen in the figure below.

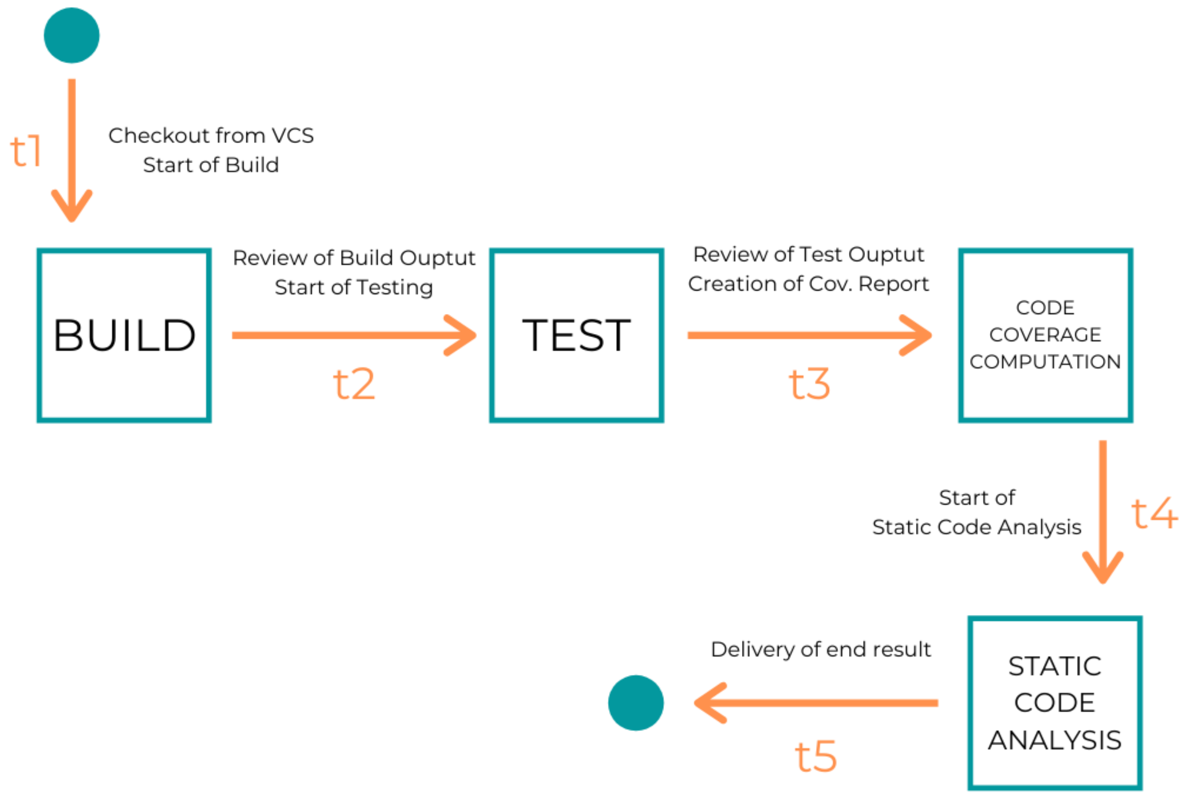


FIGURE 9 - DELIVERY FLOW OF TRUSTBANK MOBILE

It is possible to highlight that the current flow mostly refers to the Development side of the spectrum, which tries to improve its methodologies in order to rapidly deliver products of higher quality to the Operations side. In the current use case, without taking into consideration the Continuous Delivery pipeline and focusing solely on the traditional flow, it possible to detect two types of tasks that are being performed: the **Automated** ones (described inside the teal squares), such as Build, Test, Code Coverage Computation and Static Code Analysis which are executed by the compiler, and **Manual** ones (described through the orange arrows), which mostly consist in output reviews and manual inputs that start the execution of the following automated tasks.

Considering the Manual ones, which are being performed by human employees of the company, it must be noted that they are integral part of the process that has been described. Therefore, two traditional metrics come into mind, which can help analyze the impact of having these tasks performed manually:

- **Calendar Time**: this metric expresses the duration of the project or one of its tasks in the form of days, weeks, months and so on. It can also be defined in relative form (for example, "Month1" since the start of the project) or in absolute form (for example, "September 12"). Relative forms are used typically in the planning phase of the project management process, while absolute ones are used typically in the controlling phase.

- **Effort**: it expresses the time spent by employees working on a project to complete a task. It depends on Calendar Time and on people employed in the company. It is measured in **person-hour** by standard IEEE 1045, but other measures are available (like person-day, person-month, person-year: the length of each effort interval depends on national and corporation parameters). This metric directly translates into **Cost**.

Now, for each Manual task, estimates were performed by the company regarding how much time they required to be performed and how much effort had to be invested into them:

- **t1**, **t4** and **t5**: since there are no reviews involved, the time and effort involved are not relevant.

- **t2**: Calendar Time = 2 hours; Effort = 1 person for 30 minutes = 0.5 person-hour.

- **t3**: Calendar Time = 2 hours; Effort = 1 person for 40 minutes = 0.66 person-hour.

Therefore, the total time spent in manual tasks for each build, which will be called **Cumulative Time**, can be defined as:

$$CT = \sum_{i} CT\mathrm{i} = 4\ hours$$

Next, let us define the total effort spent in manual tasks for each build, which will be called **Cumulative Effort**. It is be defined as follows:

$$CE = \sum_{i} CE\mathrm{i} \cong 1\ person - hour$$

Few considerations must be made: an employee with the role of supervisor is in charge of performing the reviews to see the current status of the code. Although the effective time he spends to perform such tasks ranges from 30 minutes to 1 hour, he does not spend the entirety of his daily working time on them: other unrelated tasks with higher priority may come for him to perform, thus the actual time that passes from the GitHub checkout to the delivery to App Center is about 4 hours a day.

Now, let us take into the account the Continuous Delivery pipeline previously introduced. While Automated tasks are still present, the Manual ones are not considered anymore: the usage of Jenkins guarantees that the delivery process is rapid, but most importantly reliable, therefore there is no more need for constant reviews regarding the build or test output.

Therefore, by implementing the Continuous Delivery pipeline, the company obtains a daily profit in both metrics. However, in order to make a proper confrontation, it must be considered the time and effort that have been invested in order to make the toolchain fully active and to implement the Continuous Delivery mechanism. Therefore, new values must be considered:

- **TimeForToolchain**: 6 hours for 10 working days = 60 hours.

- **EffortForToolchain**: 1 person for 60 hours = 60 person-hour.

At the beginning of the project, when the pipeline was not yet implemented, these values represented initial losses for the company. However, when the mechanism has become fully implemented, time and effort have started to be gained daily. By projecting this confrontation over the course of time:

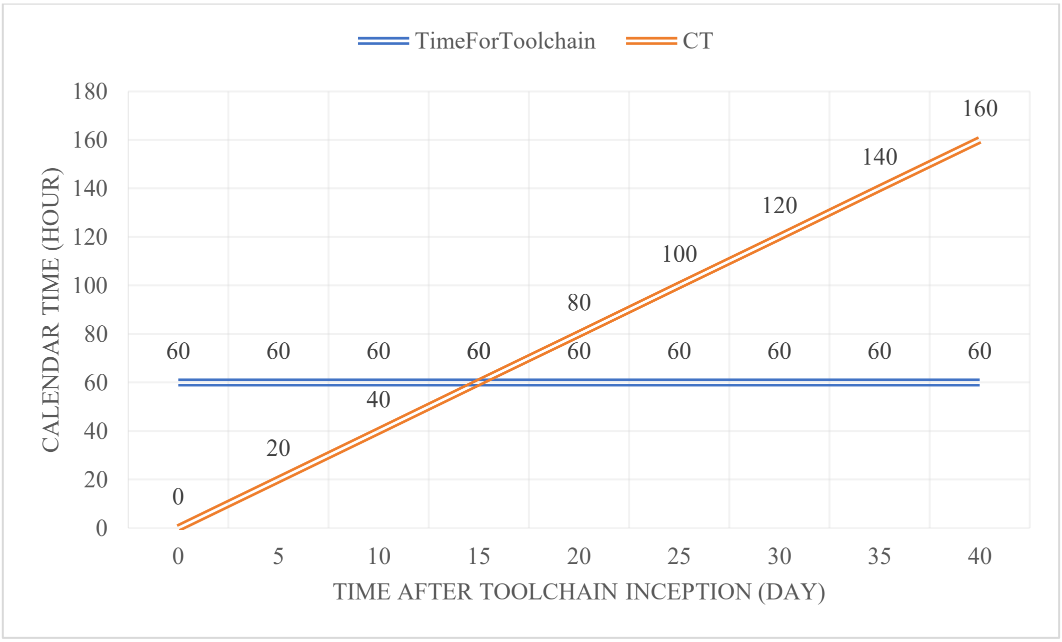


Figure 10 - TrustBank Toolchain Time Confrontation

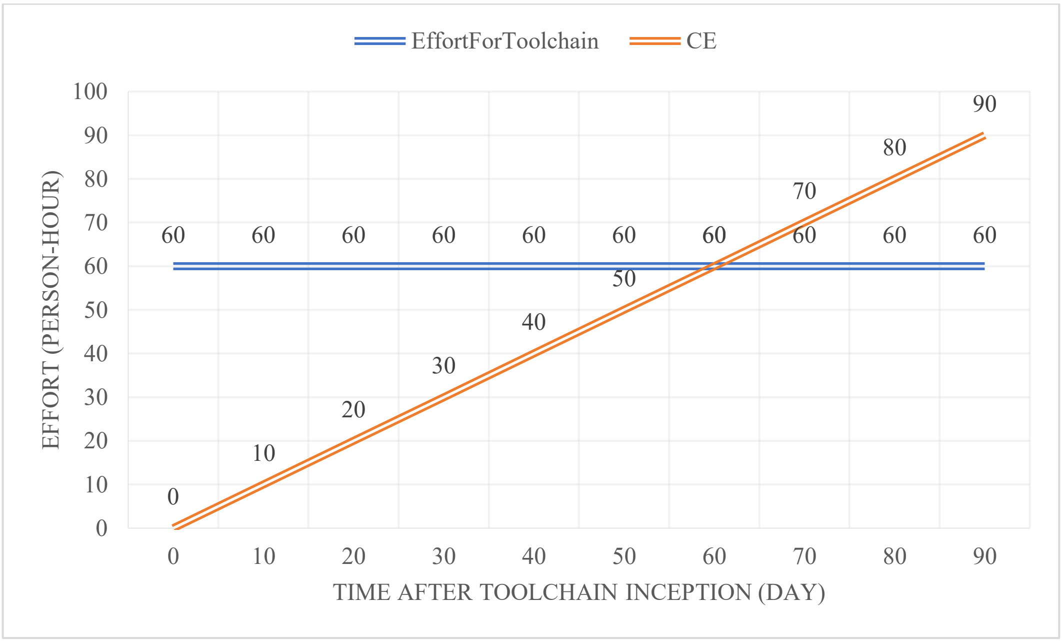


Figure 11 - TrustBank Toolchain Effort Confrontation

Through those graphics, it is possible to see that the company gains a profit from the Effort point of view starting from the 61st day since the implementation of the Continuous Delivery pipeline, while it gains a profit from the Calendar Time point of view immediately from the 16th day.

Another question may arise: why is SonarQube analysis performed, but a corresponding review in the cycle is not made? The answer will be given in the following section.

3.6 Limitations and Future Improvements

Although the software company has now built a fully functional Continuous Delivery toolchain, this mechanism is far from being perfected. Indeed, the decision of using a Custom toolchain can now be revealed as a great choice thanks to its flexibility.

First of all, we must focus on how the Continuous Delivery flow has been executed in Jenkins for this project, which is through a single Master Node running on a local instance. This was feasible, because the project was still not large enough to require higher computational power, but this may still be an issue when the TrustBank Mobile project eventually becomes more complex: it is bound to happen, since the application belongs to a major player in the banking field, which strives to add all the newest functionalities and capabilities available in the market, not even mentioning innovations introduced by internal consultations. Therefore, the idea of enhancing the Jenkins architecture by adding **Slave Nodes** to run **distributed builds** is born: being able to have multiple agents provides advantages regarding the following aspects [20]:

- **Scalability**: this may come as granted, but improving the scalability of the project can also be achieved by adding more resources to the local machine in order to improve the Jenkins cluster. However, the amount of resources that could be added is limited, thus introducing more agents becomes the ideal option.

- **Maintainability**: since most Jenkins jobs require specific third-party plugins, it may happen that some libraries or packages may come in conflict with other software installed in the local instance. By having multiple agents, it is possible to overcome these potential issues.

- **Security**: even though security measures are usually taken by software companies, a single node Jenkins cluster can still be subject to illegal access to production or testing environments from the same node. Instead, for example, by separating the Master from the actual Slaves that will perform the work, attacks on the Slaves will not result in a direct access to the Master.

It must be noted, however, that this solution may also require a great effort in maintaining the agents: installing all required plugins, monitoring their status, keeping them up-to-date and so on. That is why, in recent times, it is preferred to have dynamic agents instead of servers running in static virtual machines, which is possible due to the introduction of **containers** like Docker and Kubernetes: they are a type of operating system virtualization, able to contain all necessary executables, binary code, libraries, configuration files and so on, while avoiding containing the operating system image itself, thus being more lightweight and portable.

Another factor to be considered is the idea to exploit the usage of **Webhooks**, which are a method used to augment or alter the behavior of a web page or application with custom callbacks. A webhook request is performed through an HTTP POST request, with the payload typically in JSON format. Jenkins is able to react to this mechanism provided by many external applications, which may come useful especially for two of the tools used in the toolchain defined for TrustBank:

- **GitHub** allows for integration of webhooks that may be used to update an external issue tracker, update a backup mirror or even trigger CI servers.

- S**onarQube** also provides usage of webhooks, which are made regardless of the status of a Background Task.

In fact, Jenkins may take advantage of the first one in order to make builds start without manual intervention or even without the need to define a scheduler. The GitHub plugin allows to make use of a parameter called "GitHub hook trigger for GITScm polling", available in the configuration tab of the project: when enabled, it triggers the start of a build after receiving a GitHub webhook, which is sent under certain events. A set of these events can be defined when creating the webhook, in the proper section of the GitHub repository that contains the project, and it may include git actions such as pushes, but also branch or tag creation/deletion, issue updates, pull requests, commits with specific comments and so on.

The creation of SonarQube webhooks, instead, can be achieved through the **SonarQube Scanner plugin for Jenkins**, which allows to introduce the *waitForQualityGate* keyword for the Declarative syntax. Before analyzing this plugin, it is mandatory to define what is a SonarQube Quality Gate: this feature consists in a set of Boolean conditions against which projects are measured. Such conditions may be, for instance, that the amount of Bugs has to be less than X percentage, or that the Code Coverage must be higher than Y percentage. Although SonarQube provides a Recommended instance of Quality Gates, it is possible to define customized versions depending on the project. When the full analysis is made, if even a single condition is not met the project is marked as "Failed", otherwise it is marked as "Passed". This mechanism is exploited by the *waitForQualityGate* step, which pauses the execution of the Pipeline, waiting for the previously submitted SonarQube analysis to be completed, and returns the quality gate status through a webhook. This feature can be used in conjunction with the "abortingPipeline" parameter: if set to true, the Pipeline execution is aborted if the Quality Gate status is "Failed".

It is undeniable that webhooks may be extremely useful, so why were they not used in the newly created pipeline? The answer is immediate: running an instance of the Jenkins server in local **denies their implementation**. In fact, an instance of a GitHub or SonarQube server needs a network address reachable from the public internet, in order for the HTTP POST request to be sent. Of course, this is not possible when the server is running on a local machine, a limitation that affects the current implementation of the project and that makes it a target for further improvements.

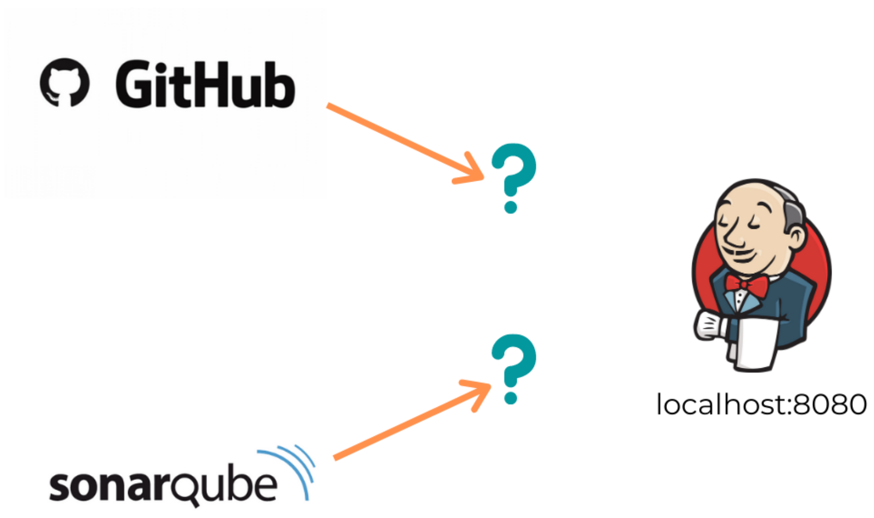


FIGURE 12 - LOCALHOST WEBHOOKS LIMITATIONS

Even after having, in the future, setup the Jenkins server on a public address, how will the company be able to make use of GitHub and SonarQube webhooks? More specifically, how can these webhooks impact the flow of Delivery currently being performed by the company? To answer this, few considerations must be made regarding how the company develops TrustBank Mobile for the banking client, in particular exploring deeply the branching strategy that the software company adopts.

The toolchain described previously works for the main flow involving the main version of the product, which is held in a branch called "Release". Each time a release cycle is concluded (the duration of which depends on the TrustBank needs), with the deployment to the production environment, the focus shifts onto the next Release branch. However all developers do not push their work directly to such branch. In fact, due to both the enterprise nature of the application and the microservices architecture it presents, developers are divided in teams that operate on a single, specific feature of the application (services like bank transfer and bank withdrawal, or even UI refactoring and so on): therefore, there are N "Feature" branches, created from the current Release branch, on which developers commit their local changes (as seen in Figure 13).

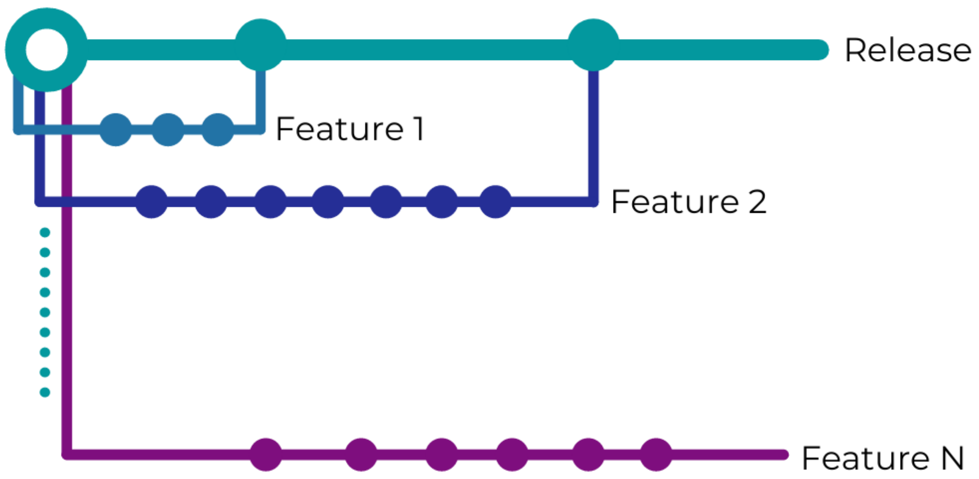


FIGURE 13 - SOFTGENIUS BRANCHING STRATEGY

Let us define the status of the application present in the Release branch as **"Release State"**. This version is, of course, not the final one of a release cycle: it will undergo through changes, internal testing on safe environments performed by the Operations team and, only when TrustBank will deem it as ready to be deployed (depending on the status of the main version and their business needs), it will be available to the end-users. About the Feature branches, when the work on a certain feature is completed, all the changes implemented undergo through several checks, then they are merged to the Release branch: this concludes the existence of the corresponding Feature branch, which is then closed. This mechanism is the one currently adopted by the company, through manual tasks. So, how can this be improved? The successful result of this project has made the company confident enough to invest in a more ambitious idea involving the aforementioned branches: **extending the automation flow also to the Feature branches**.

This can be exploited as following: each time a developer commits its changes to the Feature branch he or his team are working on, a GitHub webhook is used to automatically trigger the start of a build. The source code that undergoes this process consists in the current Release State of the application plus the changes introduced by the team that works on that particular feature: the immediate advantage is that the build involves only the changes of a single feature, ignoring all other changes that are being developed by other teams on other features. This separation is crucial, as it allows to keep the work on all microservices separated while still introducing automation in the process. Then, when building, testing and static code analysis are performed, the build output is delivered to App Center.

Through this method, several advantages may be reached:

- Through the usage of GitHub webhook, the builds become flexible: the result is a higher efficiency than the one provided by a scheduler, since a developer may start multiple builds on a single day, or even zero if no changes were introduced.

- By exploiting the distributed builds feature of Jenkins, it is possible to apply this mechanism on multiple Feature branches in parallel: this will avoid putting an excessive amount of load on the single Node that currently executes the pipeline. In particular, it also provides the capability to perform static code analysis in parallel for every single Feature branch.

- The Operations team, handled by TrustBank, will now receive, along with the daily APK of the main version on the Release branch, multiple APKs containing "partial" versions, each one with a specific new feature implemented. This furtherly reduces the gap between Developers and Operations: the latter team can now test the new features, independently one from the another, as they are being developed step by step.

- Through the implementation of the *waitForQualityGate* pipeline step, the Delivery phase is not performed if the newest changes defined by a developer introduce bugs, vulnerabilities or if the code is not covered enough by tests. This last aspect is perhaps the most important one: since every single change implemented on a Feature branch undergoes static code analysis so, through the aforementioned plugin it is guaranteed that the Operations team will receive only changes that satisfy Quality Gates, therefore the code shipped is high-quality and reliable.

The introduction of the *waitForQualityGate* step may exploit a limitation to the local execution of the Release Pipeline: if Quality Gates are not met, are the changes still being delivered to App Center and available for the Operations department of TrustBank? Would it not be necessary the presence of a review of SonarQube report? Theoretically, the answer would be yes, however, manual revisions performed on each Feature branch before merging guarantee that the Release version of the product still meets quality standards. The generated reports are still useful, since they are being used by the Operations team to perform an extra check for builds that must go to production. The advantage that will be achieved with the aforementioned improvements is that these manual tasks will be no more, being replaced by automatic execution instead.

Another crucial point is the merging step of a Feature branch into the Release one, which happens only once, when the work on that specific feature is done. This task is usually tedious and requires great effort from multiple employees and supervisors, due to its importance. However, the company has made the following consideration: if the entire development of a Feature is automated and meets quality standards, then it is guaranteed that the changes that will be merged to the Release branch will not negatively affect the quality of the main version. Thus, it would also be possible to automate this task by modifying the Pipeline accordingly: an initial step may be introduced to check for conflicts between the Feature branch in consideration and the Release one and, if there are no issues, then the next steps will be performed as usual, with the final one being the act of merging into Release.

To conclude this discussion, let us recap the types of pipeline that were discussed:

- **Release Pipeline**: it is the one currently available for use by the company and executed in local. It operates on the main version of the TrustBank Mobile application (Release branch) and a build is performed once a day due to business agreement between the two companies.

- **Feature Pipelines**: they are the next ones the company aims to introduce. Each one operates on a partial version of the application (Feature branch) and a build is performed whenever a developer performs a commit (ranging from zero to multiple ones on the same day).

- **Merge Pipelines**: another type that the company strives to implement. Each one operates on a partial version of the application (Feature branch) and a build is performed only once, when no more work on the corresponding Feature branch has to be performed and the merge into Release branch must be performed.

Chapter 4

Conclusions

This project was born with an idea in mind: to reduce the bridge that normally affects Development and Operations teams in a concrete corporate context such as the relationship between SoftGenius and TrustBank. Culture, Processes and Tools are already present in the SoftGenius flow of work, but the mindset of the company is focused on continuous improvement and research of more efficient and reliable mechanisms, Continuous Integration and Continuous Delivery above all.

The analysis of the current state of the DevOps market, of many categories of instruments that facilitate the flows of development and delivery, has culminated in the integration of several tools, some already in use by the company, others new which have been previously seen lightly or just mentioned. Through all of them, the most interesting and flexible one has been found in Jenkins, a Continuous Integration server that over the course of the years has been improved through the implementation of plugins by a large community that backs it up. The capabilities of such instrument are huge, some of which resulted extremely useful in integrating third-party tools such as GitHub, Slack and SonarQube, and in helping orchestrate a Continuous Delivery mechanism in an easily accessible and rapid way, thanks also to the presence of a Domain Specific Language such as the Declarative Pipeline one and the possibility of defining the entirety of the pipeline through code inside a Jenkinsfile, versioned and fully made part of the project itself.

Successfully implementing a DevOps toolchain to exploit Continuous Delivery has undoubtedly shown its benefits: the introduction of automation has transformed the delivery process into a robust and errorless one, which is crucial in such a field as the banking one, with the TrustBank Mobile case. Not only that, but as it was seen in Paragraph 3.5, a great long-term profit regarding effort and time spent by the company has also been gained: tedious manual tasks have been eliminated from the flow, due to the security and reliability that only automation can guarantee.

It is important to note that, thanks to this experimental project, the software company has also gathered important information regarding several aspects of the DevOps world, such as the panorama of instruments available on the market, the steps required to build up a Custom toolchain, the understanding of mostly all features that Jenkins has to offer and the potential of this tool. This knowledge will also make SoftGenius improve their cultural and processual DevOps elements, allowing it to become an even greater competitor in the counseling sector: when a new client will approach this software company, it will be assured by the deep DevOps knowledge that SoftGenius brings to the table.

Furthermore, even after this experience, the company will not simply be satisfied by the current Continuous Delivery implementation regarding TrustBank Mobile. In fact, as seen in Paragraph 3.6, SoftGenius is ready to push their knowledge even further by making use of the Continuous Delivery paradigm also in their branching strategy. The next goal has been declared: to introduce the highest degree of automation possible to the development and delivery process of each new Feature implemented in the application, allowing for multiple Jenkins builds to be performed in parallel and exploiting the great potential that webhooks have to offer.

Appendix A

Source code of jacocoTestReport task

```
task jacocoTestReport(type: JacocoReport, dependsOn: 'MY_TEST_TASK') {

    reports {
        xml.enabled(true)
        xml.destination(file("${buildDir}/reports/
                         jacoco/test/jacocoTestReport.xml"))
        html.enabled(true)
    }

    def fileFilter = [
            "**/R.class",
            "**/BuildConfig.*",
            "**/Manifest*.*",
            "**/*Test*.*",
            "android/**/*.*",
            "**/*Fragment.*",
            "**/*Activity.*"
    ]
    def kotlinTreeClasses = fileTree(dir: "MY_KOTLIN_CLASSES",
                                      excludes: fileFilter)
    def javaTreeClasses = fileTree(dir: "MY_JAVA_CLASSES",
                                    excludes: fileFilter)
    def mainSrc = "${project.projectDir}/src/main/java"

    sourceDirectories.from(files([mainSrc]))
    classDirectories.from = files([kotlinTreeClasses, javaTreeClasses])
    executionData.from(files("${buildDir}/jacoco/MY_TEST_EXEC"))
}
```

Appendix B

Source code of Jenkinsfile

```
pipeline {

    agent any

    parameters {

        booleanParam(name: 'PERFORM_STATIC_CODE_ANALYSIS',
                     defaultValue: true, description: 'This parameter
                         specifies if static code analysis must be performed')

        booleanParam(name: 'DELIVER_TO_PRODUCTION',
                     defaultValue: true, description: 'This parameter
                         specifies if the build result must be published')
    }

    triggers {
        parametrizedCron('''
                 H/30 18 * * *
         ''')
    }


    stages {

        stage('Slack - Notify Start Build'){
            steps {
                slackSend channel: '#trustbankbuilds',
                        message: 'Started ${JOB_NAME} with build #${BUILD_NUMBER}'
            }
        }

        stage('Gradle - Clean Project'){
            steps {
                sh './gradlew MY_CLEAN_TASK'
            }
        }

        stage('Gradle - Build Project'){
            steps {
                sh './gradlew MY_BUILD_TASK'
            }
        }

        stage('Gradle - Perform Testing'){
            steps {
                sh './gradlew MY_TEST_TASK'
            }
        }
        stage('JaCoCo - Create Code Coverage Report'){
            when {
                expression { params.PERFORM_STATIC_CODE_ANALYSIS }
            }
            steps {
                sh './gradlew jacocoTestReport'
            }
        }

        stage('SonarQube - Static Code Analysis'){
            when {
                expression { params.PERFORM_STATIC_CODE_ANALYSIS }
            }
            steps {
                sh './gradlew sonarqube'
            }
        }

        stage('AppCenter - Publish Build){
            when {
                expression { params.DELIVER_TO_PRODUCTION }
            }
            steps {
                appCenter apiToken: 'MY_APPCENTER_TOKEN',
                        appName: 'DevOps2021',
                        distributionGroups: 'DevOpsTester',
                        ownerName: 'MY_OWNER_NAME',
                        pathToApp: 'MY_APK_PATH'
            }
        }

    }


    post{

        always{
            publishHTML(target: [allowMissing: true,
                 alwaysLinkToLastBuild: true, keepAll: false,
                 reportDir: 'MY_REPORT_DIR', reportFile: 'index.html',
                 reportName: 'Test Report', reportTitles: 'Test Report'])
        }
        success{
            slackSend channel: '#trustbankbuilds',
                    message: '${env.JOB_NAME} - #${BUILD_NUMBER} finished
                      with status: *${currentBuild.currentResult}*.',
                    color: 'good'
        }
        failure{
            slackSend channel: '#trustbankbuilds',
                    message: '${env.JOB_NAME} - #${BUILD_NUMBER} finished
                      with status: *${currentBuild.currentResult}*.',
                    color: 'danger'
        }
    }


}
```

References

[1] S. Mezak, 25 January 2018. [Online]. Available: https://devops.com/the-origins-of-devops-whats-in-a-name/.

[2] M. Huttermann, in *DevOps for Developers*, 2012.

[3] S. Linnanvuo, "Agile & Lean Metrics: Cycle Time," 28 October 2015. [Online]. Available: https://screenful.com/blog/software-development-metrics-cycle-time.

[4] M. Rikmus, "What is DevOps? An Intersection of Culture, Processes and Tools," 10 October 2019. [Online]. Available: https://dzone.com/articles/what-is-devops-an-intersection-of-culture-processe.

[5] Anonymous, "DevOps: Shift Left to Reduce Failure," 2 June 2016. [Online]. Available: https://devops.com/devops-shift-left-avoid-failure.

[6] JakobTheDev, "The Eight Phases of a DevOps Pipeline," 18 June 2019. [Online]. Available: https://medium.com/taptuit/the-eight-phases-of-a-devops-pipeline-fda53ec9bba.

[7] AWS, "What is Continuous Integration?," [Online]. Available: https://aws.amazon.com/devops/continuous-integration/.

[8] AWS, "What is Continuous Delivery?," [Online]. Available: https://aws.amazon.com/devops/continuous-delivery/.

[9] Markets and Markets, "DevOps Market by Type (Solutions and Services), Deployment Model (Public, Private, and Hybrid), Organization Size, Vertical (BFSI, Healthcare, Telecommunications & ITES, Manufacturing), and Region - Global Forecast to 2023," 10 Sep 2019. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/devops-824.html.

[10] S. Watts, "What is a DevOps Toolchain?," 4 Mar 2019. [Online]. Available: https://www.bmc.com/blogs/devops-toolchain/.

[11] A. T. Tunggal, "Bitbucket vs GitHub," 27 Nov 2020. [Online]. Available: https://www.upguard.com/blog/bitbucket-vs-github.

[12] A. Stringfellow, "Gradle vs. Maven," 30 Jun 2017. [Online]. Available: https://dzone.com/articles/gradle-vs-maven.

[13] Gradle, "Gradle vs Maven Comparison," [Online]. Available: https://gradle.org/maven-vs-gradle/.

[14] R. Jain, "Jenkins vs GitLab CI: Battle of CI/CD Tools," 11 Aug 2020. [Online]. Available: https://www.lambdatest.com/blog/jenkins-vs-gitlab-ci-battle-of-ci-cd-tools/.

[15] P. Mishra, "CircleCI vs Jenkins: Choosing The Right CI CD Tool," 8 Jul 2020. [Online]. Available: https://www.lambdatest.com/blog/circleci-vs-jenkins-choosing-the-right-ci-cd-tool/.

[16] B. Doerrfeld, D. MacVittie and O. Frank, "The State of DevOps Tools 2019," DevOps.com, 2019.

[17] Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/appcenter/distribution/.

[18] A. Khan, "AWS Public Sector Summit," Washington D.C., 2018.

[19] "Pipeline as Code with Jenkins," [Online]. Available: https://www.jenkins.io/solutions/pipeline/.

[20] A.-W. Shihadeh, "5 Tips for Hosting Your Own Jenkins Instance," 13 Feb 2020. [Online]. Available: https://medium.com/better-programming/5-tips-for-hosting-your-own-jenkins-instance-64b058ad2c2a.