Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"

Network Technologies Department of the Faculty of Informatics

Organization and control of continuous code delivery

Text part of course work
in specialty “Computer Science and Information Technologies” 112

Coursework supervisor
associate professor, doctor of technical science
Glybovets A.M.

_____
(signature)

“____” _____ 2020 yr.

Made by student
Ivanov O.A

“____” _____ 2020 yr.

Kyiv 2020

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"
Network Technologies Department of the Faculty of Informatics

APPROVED
Head of Department of Computer Science
associate professor of Computer Science
_____ S.S. Gorokhovsky
(signature)
"____" _____ 2020 yr.

INDIVIDUAL TASK
for course work
For the student of the Faculty of Informatics of 1 course of master program
THEME Organization and control of continuous code delivery

Output data:
Text part content of coursework:
An individual task
Calendar plan
Abstract
Introduction
Part 1: The Problem of Delivering Software
Part 2: Continuous Integration
Part 3: Continuous Deployment
Summary
References

Issue date "____" _____ 20 yr. Supervisor _____
(signature)
Task received _____
(signature)

Theme: Organization and control of continuous code delivery

Calendar plan of coursework execution:

№	Stage name	Deadline	Note
1.	Getting of coursework topic	01.09.2019	
2.	Searching of appropriate literature	10.09.2019	
3.	Part 1: The Problem of Delivering Software	10.12.2019	
4.	Part 2: Continuous Integration	25.02.2020	
5.	Part 3: Continuous Deployment	01.04.2020	
6.	Documentation forming	10.04.2020	
14.	Writing coursework summary	01.05.2020	
15.	Coursework analysis with the supervisor	02.05.2020	
16.	Coursework changing according to the supervisor's remarks	03.05.2020	
17.	Creating of the presentation	05.05.2020	
18.	Defending of the coursework	14.05.2020	

Student Ivanov O.A.
Supervisor Glybovets A.M.
"____" _____

Contents

Abstract

In this coursework will be defined what is: continuous integration (CI), continuous deployment and delivery (CD), branching strategies. Then will be covered common patterns and anti-patterns of implementation mentioned systems.

The second part of the coursework will explore Continuous Integration with focus on it parts. Will be explained importance and ways of optimization of CI process.

The third part will uncover Continuous Deployment process. Will be covered Infrastructure as a Code approach (IaC), release strategies for production, and zero-downtime deployments approaches. The finally: some example of zero-downtime implementation will be presented.

Key words: CI/CD systems, IaC, release strategies, GIT flow, zero-downtime, branching strategies.

INTRODUCTION

An extremely strange, but common, feature of many software projects is that for long periods of time during the development process the application is not in a working state. In fact, most software developed by large teams spends a significant proportion of its development time in an unusable state. The reason for this is easy to understand: Nobody is interested in trying to run the whole application until it is finished. Developers check in changes and might even run automated unit tests, but nobody is trying to actually start the application and use it in a production-like environment. [1]

This is doubly true in projects that use long-lived branches or defer acceptance testing until the end. Many such projects schedule lengthy integration phases at the end of development to allow the development team time to get the branches merged and the application working so it can be acceptance-tested. Even worse, some projects find that when they get to this phase, their software is not fit for purpose. These integration periods can take an extremely long time, and worst of all, nobody has any way to predict how long. [1]

On the other hand, we have seen projects that spend at most a few minutes in a state where their application is not working with the latest changes. The difference is the use of continuous integration. Continuous integration requires that every time somebody commits any change, the entire application is built, and a comprehensive set of automated tests is run against it. Crucially, if the build or test process fails, the development team stops whatever they are doing and fixes the problem immediately. The goal of continuous integration is that the software is in a working state all the time.[1]

The problem with Continuous Integration, Delivery, and Deployment is that it's not at all easy to set up and takes a lot of time, especially when you've never done it before or want to integrate an existing project. However, when done right, it will pay itself back by reducing bugs, making it easier to fix the bugs you find and producing better quality software (which should lead to more satisfied customers). [2]

The importance of such system is critical for successful projects. Creation, configuration, best practices, and common approaches will be described in this work.

Task formulation:

1. Provide definition of Continuous Integration.
2. Describe approaches for Continuous Integration implementation.
3. Describe common mistakes during implementation of CI.
4. Uncover difference between Continuous delivery and deployment.
5. Provide overview of release strategies. Uncover IaC principles.
6. Design Zero-downtime approach using one of cloud providers.

CHAPTER 1: The Problem of Delivering Software

1.1 The delivery pipelines.

Every time that you start project you should know the release date (the point in time then end user will be able to use your product). A lot of estimation, planning and other activities could be done before, but this date will be delayed due to bugs or undetected defects in software.

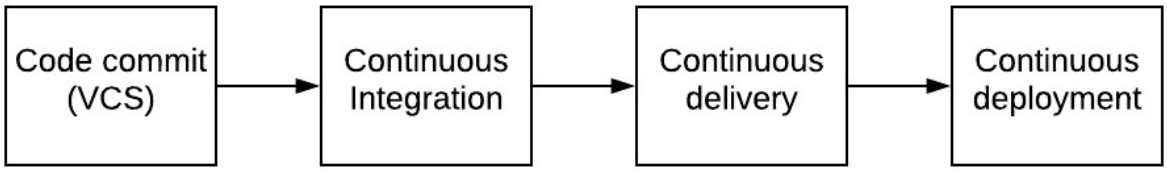

Figure 1.1 The delivery pipeline

On the Figure 1.1 shown general approach of Software delivery. Code, that was written by developer, should pass through all this stages. On the other hands: such structure is not implemented at every project. Common situations that code deployed manually, or test coverage is low and as a result CI system are not informative.

Continuous Integration, Delivery, and Deployment are relatively new development practices that have gained a lot of popularity in the past few years. Continuous Integration is all about validating software as soon as it's checked in to source control, more or less guaranteeing that software works and continues to work after new code has been written. Continuous Delivery succeeds Continuous Integration and makes software just a click away from deployment. Continuous Deployment then succeeds Continuous Delivery and automates the entire process of deploying software to your customers (or your own servers).[2]

If Continuous Integration, Delivery, and Deployment could be summarized with one word, it would be Automation. All three practices are about automating the process of testing and deploying, minimizing (or completely eliminating) the need for human intervention, minimizing the risk of errors, and making building and deploying software

easier up to the point where every developer in the team can do it (so you can still release your software when that one developer is on vacation or crashes into a tree).[2]

1.2 Antipatterns and common mistakes

In many software projects, release is a manually intensive process. The environments that host the software are often crafted individually, usually by an operations or infrastructure team. Third-party software that the application relies on is installed. The software artifacts of the application itself are copied to the production host environments. Configuration information is copied or created through the admin consoles of web servers, applications servers, or other third-party components of the system. Reference data is copied, and finally the application is started, piece by piece if it is a distributed or service-oriented application. There is quite a lot to go wrong in this process. If any step is not perfectly executed, the application won't run properly. And at this point it may not be at all clear where the error is, or which step went wrong.[1] In the book *"Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation"* by David Farley, Jez Humble., Few antipatterns were described. They are below:

The first antipatterns is **"Deploying Software Manually"**. Most modern applications of any size are complex to deploy, involving many moving parts. Many organizations release software manually. By this we mean that the steps required to deploy such an application are treated as separate and atomic, each performed by an individual or team. Judgments must be made within these steps, leaving them prone to human error. Even if this is not the case, differences in the ordering and timing of these steps can lead to different outcomes. These differences are rarely good.

The common signs of this antipattern are:

- Reliance on manual testing to confirm that the application is running correctly
- Frequent corrections to the release process during the course of a release
- Releases that take more than a few minutes to perform

- Dev team are sitting bleary-eyed in front of a monitor at 2 A.M. the day after the release day, trying to figure out how to make it work

If you recognize yourself, we have bad news for you. Such approach will “kill” your project and/or your team. On the other hand, over time, deployments should tend towards being fully automated. There should be two tasks for a human being to perform to deploy software into a development, test, or production environment: to pick the version and environment and to press the “deploy” button. Releasing packaged software should involve a single automated process that creates the installer. The one-button-deploy is the goal that should be achieved.

The second antipattern is **“Deploying to a Production-like Environment Only after Development Is Complete”**. This pattern could be described in few words: “If you have deployed to production-like environment at last sprint you are at middle sprint”. The aim is deploying to production-like environment as sooner as possible.

Common signs of this antipattern are:

- Deployment to production-like environment occurs after half time of total project estimation.
- development team meet operation team during first deploy to prod.
- QA team were not involved into testing on production. And many others

All of this will lead to project delay and as a result losing money for business. The remedy is to integrate the testing, deployment, and release activities into the development process. Make them a normal and ongoing part of development so that by the time you are ready to release your system into production there is little to no risk, because you have rehearsed it on many different occasions in a progressively more production-like sequence of test environments. Make sure everybody involved in the software delivery process, from the build and release team to testers to developers, work together from the start of the project. We are test addicts, and the extensive use of continuous integration and continuous deployment, as a means of testing both our software and our deployment process, is a cornerstone of the approach that we describe.

The third antipattern is **"Manual Configuration Management of Production Environments"**

Many organizations manage the configuration of their production environments through a team of operations people. If a change is needed, such as a change to database connection setting or an increase in the number of threads in a thread pool on an application server, then it is carried out manually on the production servers. If a record is kept of such a change, it is probably an entry in a change management database.

Signs of this antipattern are:

- Having deployed successfully many times to staging, the deployment into production fails.
- Different members of a cluster behave differently—for example, one node sustaining less load or taking longer to process requests than another.
- The operations team take a long time to prepare an environment for a release.
- You cannot step back to an earlier configuration of your system, which may include operating system, application server, web server, RDBMS, or other infrastructural settings.
- Servers in clusters have, unintentionally, different versions of operating systems, third-party infrastructure, libraries, or patch levels.
- Configuration of the system is carried out by modifying the configuration directly on production systems.

All aspects of each of your testing, staging, and production environments, specifically the configuration of any third-party elements of your system, should be applied from version control through an automated process.

One of the key practices is configuration management, part of which means being able to repeatably re-create every piece of infrastructure used by your application. That means operating systems, patch levels, OS configuration, your application stack, its configuration, infrastructure configuration, and so forth should all be managed. You

should be able to recreate your production environment exactly, preferably in an automated fashion. IaC can help you get started with this.

You should know exactly what is in production. That means that every change made to production should be recorded and auditable. Often, deployments fail because somebody patched the production environment last time they deployed, but the change was not recorded. Indeed it should not be possible to make manual changes to testing, staging, and production environments. The only way to make changes to these environments should be through an automated process.

1.3 Conclusion

Based on the previous two sections and the industry best practices we can tell that we need to make frequent, automated releases of software.

Automated. If the build, deploy, test, and release process is not automated, it is not repeatable. Every time it is done, it will be different, because of changes in the software, the configuration of the system, the environments, and the release process. Since the steps are manual, they are error-prone, and there is no way to review exactly what was done. This means there is no way to gain control over the release process, and hence to ensure high quality. Releasing software is too often an art; it should be an engineering discipline.

Frequent. If releases are frequent, the delta between releases will be small. This significantly reduces the risk associated with releasing and makes it much easier to roll back. Frequent releases also lead to faster feedback—indeed, they require it. The idea of CI/CD system is concentrates on getting feedback on changes to your application and its associated configuration (including its environment, deployment process, and data) as quickly as possible.

Feedback is essential to frequent, automated releases. There are three criteria for feedback to be useful:

- Any change, of whatever kind, needs to trigger the feedback process.
- The feedback must be delivered as soon as possible.

- The delivery team must receive feedback and then act on it.

CHAPTER 2: Continuous Integration

2.1 Continuous Integration overview

Continuous integration was first written about in Kent Beck's book *Extreme Programming Explained* (first published in 1999). As with other Extreme Programming practices, the idea behind continuous integration was that, if regular integration of your codebase is good, why not do it all the time? In the context of integration, "all the time" means every single time somebody commits any change to the version control system. As one of our colleagues, Mike Roberts, says, "Continuously is more often than you think" [1].

Continuous integration (CI) is the process of integrating new code written by developers with a mainline or "master" branch frequently throughout the day. This contrasts with having developers working on independent feature branches for weeks or months at a time, merging their code back to the master branch only when it is completely finished. Long periods of time in between merges means that much more has been changed, increasing the likelihood of some of those changes being breaking ones. With bigger changesets, it is much more difficult to isolate and identify what caused something to break. With small, frequently merged changesets, finding the specific change that caused a regression is much easier. The goal is to avoid the kinds of integration problems that come from large, infrequent merges. [3]

In order to make sure that the integrations were successful, CI systems will usually run a series of tests automatically upon merging in new changes. When these changes are committed and merged, the tests automatically start running to avoid the overhead of people having to remember to run them—the more overhead an activity requires, the less likely it is that it will get done, especially when people are in a hurry.[1]

The outcome of these tests is often visualized, where "green" means the tests passed and the newly integrated build is considered clean and failing or "red" tests means the

build is broken and needs to be fixed. With this kind of workflow, problems can be identified and fixed much more quickly.[3]

Continuous integration represents a paradigm shift. Without continuous integration, your software is broken until somebody proves it works, usually during a testing or integration stage. With continuous integration, your software is proven to work (assuming a sufficiently comprehensive set of automated tests) with every new change—and you know the moment it breaks and can fix it immediately. The teams that use continuous integration effectively can deliver software much faster, and with fewer bugs, than teams that do not. Bugs are caught much earlier in the delivery process when they are cheaper to fix, providing significant cost and time savings. Hence, CI an essential practice for professional teams, perhaps as important as using version control.[1]

Here are few benefits that have made continuous integration essential to any software development lifecycle.[4]

Early Bug Detection: If there is an error in the local version of the code that has not been checked previously, a build failure occurs at an early stage. Before proceeding further, the developer will be required to fix the error. This also benefits the QA team since they will mostly work on builds that are stable and error-free.

Reduces Bug Count: In any application development lifecycle, bugs are likely to occur. However, with Continuous Integration and Continuous Delivery being used, the number of bugs is reduced a lot. Although it depends on the effectiveness of the automated testing scripts. Overall, the risk is reduced a lot since bugs are now easier to detect and fix early.

Automating the Process: The Manual effort is reduced a lot since CI automates build, sanity, and a few other tests. This makes sure that the path is clear for a successful continuous delivery process.

The Process Becomes Transparent: A great level of transparency is brought in the overall quality analysis and development process. The team gets a clear idea when a test fails, what is causing the failure and whether there are any significant defects. This

enables the team to make a real-time decision on where and how the efficiency can be improved.

Cost-Effective Process: Since the bug count is low, manual testing time is greatly reduced and the clarity increases on the overall system, it optimizes the budget of the project.

2.2 Continuous Integration best practices

Before starting to build new system you'd better ask yourself about existing one. Implementation of industry best practices will allow you to reduce time and avoid errors. Martin Fowler in his article *Continuous Integration* (First published in 2006) mentioned eleven practices, and they are still relevant.

Maintain a Single Source Repository. This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimized. Instead, it is preferred for changes to be integrated rather than for multiple versions of the software to be maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.[5]

The original principle recommends NOT to use branching in the version control system. Instead, it recommends that only a single branch of the project be under development all the time. However, in most organizations, it is necessary to have many branches where development is ongoing in parallel. Often, companies need to support the previous releases of the product, fix bugs in them, while other team members start working on the next release. This requires multiple branches in the code base.[6]

Automate the Build A single command should have the capability of building the system. Many build tools, such as make, have existed for many years. Other more recent

tools are frequently used in continuous integration environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Debian DEB, Red Hat RPM or Windows MSI files).[5]

Automation of the build should include steps such as compiling the code, executing unit tests and integration tests. They may also include a number of other tools as described in the previous post - code quality checks, semantic checks, measuring technical debt etc. Most of the modern build tools support these additional integrations and should be used in developing continuous integration environments.

In the real world projects, different teams might be responsible for developing different parts of the system, each having its own repository. In such a case, it is almost impossible (without significant work) and quite unnecessary to automate the build across the entire product. It is generally sufficient if the build automation is developed for each individual part of the system.[6]

Make the Build Self-Testing Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave. [4]

The code should contain unit tests at the minimum. Frameworks such as JUnit can be used to easily mock dependencies. The interactions of a component with other modules should be mocked. This ensures that a module can be tested independently of others.

Unit tests should test behavior, not implementation details. What is the difference? Let’s take an example to figure it out: Testing for behavior: “I don’t care how you calculate the speed of the car, just make sure that the answer is correct” When I test for implementation: “I don’t care what the answer is, just make sure you use the equation: Speed = Distance/Time” Testing for behavior is the correct approach because we need to verify the results, not how the solution was achieved.[5]

Many testing frameworks allow us to Assert the mock objects i.e. test whether the mock object was called, whether it was passed in a certain argument etc. These assets should be minimized unless testing for the implementation itself is the primary concern.[6]

Everyone Commits to the Baseline Every Day. By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making. Committing all changes at least once a day (once per feature built) is generally considered part of the definition of Continuous Integration. In addition performing a nightly build is generally recommended. [5]

These are lower bounds; the typical frequency is expected to be much higher.

The code should contain unit tests at the minimum. Frameworks such as JUnit can be used to easily mock dependencies.

The interactions of any component with other modules should be mocked.

No in-progress work should be committed into the master branch. The master branch should always be working software.

Do not commit to any branch if you see that the build is broken. You should verify what caused the error and try to solve it as soon as possible instead of committing your code. [6]

Every Commit (to Baseline) Should be Built. The system should build commits to the current working version to verify that they integrate correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. For many, continuous integration is synonymous with using Automated Continuous Integration where a continuous integration server or daemon monitors the revision control system for changes, then automatically runs the build process.[5]

You should separate the CI workflows for the master branch and other branches. This includes the steps from compilation right up to packaging and testing. The build on

the master branch should generally include more tests. The build on the master branch might also need different scripts to run because the application might need to be packaged differently for different deployment platforms. A build running on the other branches might not need a packaging step at all, or it is generally limited to packaging for the same platform as that of the developer.

A “nightly build” should also execute at a scheduled time every night. This build should include more verifications than the ones on other branches. It takes longer to run and is executed less frequently.

There should be no commented-out tests in mainline branch. By commenting out tests, we get an incorrect indication of the status of the build.

Introduce coding standard checks as part of CI process. The code must be reviewed using the automated tools as well as by the team members before check-in to mainline.[6]

Keep the Build Fast The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.[4]

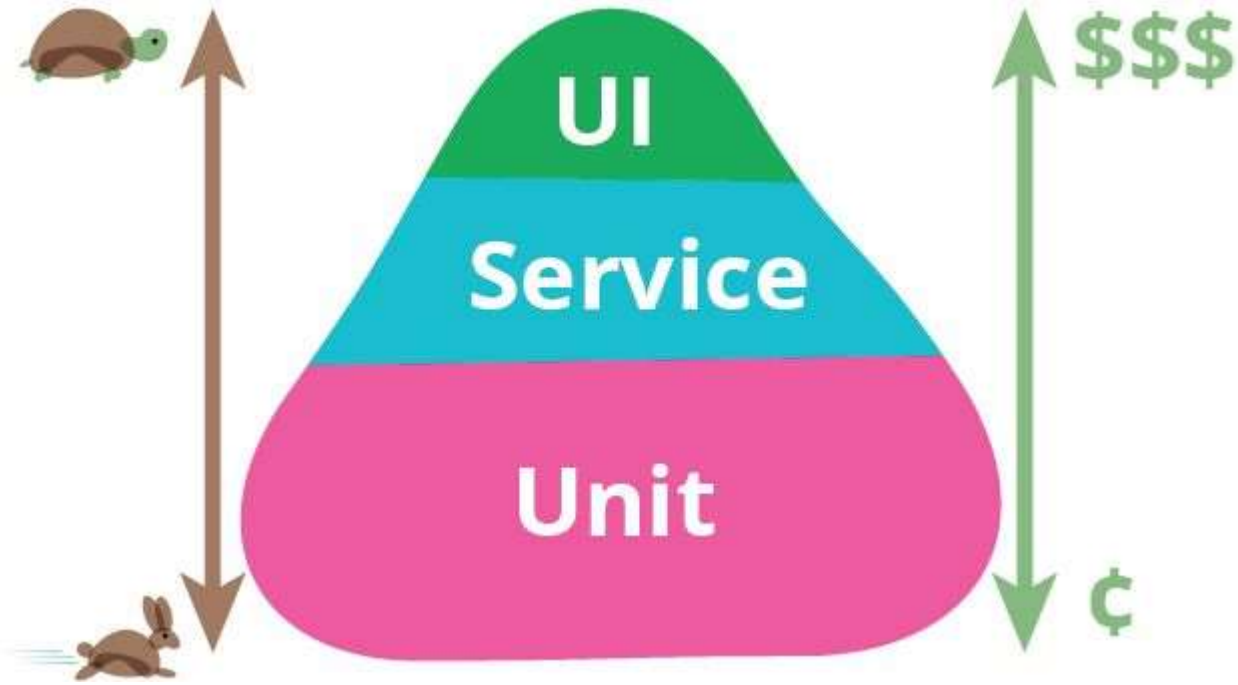

Figure 2.1 Test pyramid by Martin Fowler

The test pyramid, as explained by Martin Fowler on Figure 2.1. You should aim to have more fast-executing tests than slow tests. This would mean that you need to have more unit tests than other types of tests.[4]

Avoid using a database in unit tests. If possible, avoid using it for integration tests too. This usually needs the integration tests to use an alternative data source, usually pointing to an in-memory database. If it’s not possible to avoid using a real database for tests, you need to make sure to refresh the database before each test, so as to ensure that the data is in a known state, and tests don't start with inconsistent data.[5]

Do not depend on many UI tests, UI tests are brittle i.e. they frequently change and need a lot of maintenance. I recommend you use UI test frameworks like Selenium to alleviate some of the problems of UI testing such as the location of the UI element changing on the screen, handling UI events, and other UI parameters.[6]

Test in a Clone of the Production Environment Having a test environment can lead to failures in tested systems when they deploy in the production environment because the production environment may differ from the test environment in a significant way. However, building a replica of a production environment is cost prohibitive. Instead, the test environment, or a separate pre-production environment ("staging") should be built to be a scalable version of the actual production environment to both alleviate costs while maintaining technology stack composition and nuances. Within these test environments, service virtualization is commonly used to obtain on-demand access to dependencies (e.g., APIs, third-party applications, services, mainframes, etc.) that are beyond the team's control, still evolving, or too complex to configure in a virtual test lab. [4]

This is the hardest principle to put into practice in real world development. This needs the build automation system to create and deploy the packages into a staging environment that reflects the real production environment. Unless your application is self-sufficient with no external dependencies, achieving this is difficult because of the sheer complexity of the production environment. My suggestion for complex products is to

invest time and effort into using a virtualization platform or a container platform such as Docker to replicate the production environment. Continuous delivery pipelines can be used to deploy the build into these environments.

Make It Easy to Get the Latest Deliverables Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding errors earlier also, in some cases, reduces the amount of work necessary to resolve them. All programmers should start the day by updating the project from the repository. That way, they will all stay up to date.[4]

It is recommended to use an artifact repository such as Nexus or Nuget server to store the packages from the latest build. Usually, the packages stored in such an artifact repository are also version controlled with a build number. This makes it easy for everyone involved to get any artifacts, current or past.[6]

Only the build packages from the mainline branch should be stored in the artifact repository. If not, it will result in overwriting of the existing packages each time someone builds a work in progress branch.[6]

Everyone Can See the Results of the Latest Build. It should be easy to find out whether the build breaks and, if so, who made the relevant change.[4]

All modern CI servers have the capability to display a dashboard containing the status of the builds. They can also be configured to show other metrics as described in the previous post.All CI servers can also be configured to send email notifications when the build completes. It is recommended that the emails are sent to the whole team when the build fails so tat it can be fixed as soon as possible.[2]

A failing build is not a hall of shame. Everyone makes mistakes and developers are not immune to it. When the build is failing, it should be considered as a welcome result because the problem was identified early. Failing early and fixing problems early is the key aim of CI.[5]

CI is not only for developers. The various metrics that can be derived using the CI system by installing extensions can be made use of to improve not just the quality of the software, but also the quality of the development practices.[1]

Automate Deployment. Most CI systems allow the running of scripts after a build finishes. In most situations, it is possible to write a script to deploy the application to a live test server that everyone can look at. A further advance in this way of thinking is continuous deployment, which calls for the software to be deployed directly into production, often with additional automation to prevent defects or regressions.[4]

Not all projects need automated deployment, especially if the enterprise servers are on customer site. The project plan determines when the upgrade to the latest build happens on the customer site; usually, this is planned months in advance. If the production sites are self-hosted by the same company that is developing the software, it is much more beneficial to invest in Continuous deployment systems. Continuous deployment is the next logical step after the process for Continuous integration is in place and working well.[6]

Not all commits result in a shippable product. It is a common misconception in the Agile community that each build is a shippable product. A shippable product is entirely different than working software.

The described best practices should be implemented in every project, however we should keep in mind the limitation in resources, such as:

- Limited budget,
- Not enough expertise inside team (example no automation QA),
- Time limitation,
- Security restriction. (GDPR)

In this case we should optimize process. But we should not skip any of practices mentioned before. The solution might be found in using open-source software like Jenkins or TeamCity for CI. Jenkins are totally free, and TeamCity contains professional version

for free. One we can reduce, but not skip it, is automation testing. On the other hand, we should keep unit test and extend they coverage.

2.3 Branching strategies

Every piece of code is stored in version control system. The way how we store, update code base and manage development process will have significant impact to our project. One of most critical and important decision in building CI system is branching strategies. Depends on it we can predict behavior and load of CI. In this section we will cover the most common of them.

Trunk-based Development Workflow or simply called trunk is the strategy of one branch, usually it is master, and tagging release branches. The visualization of trunk provided on figure 2.2.

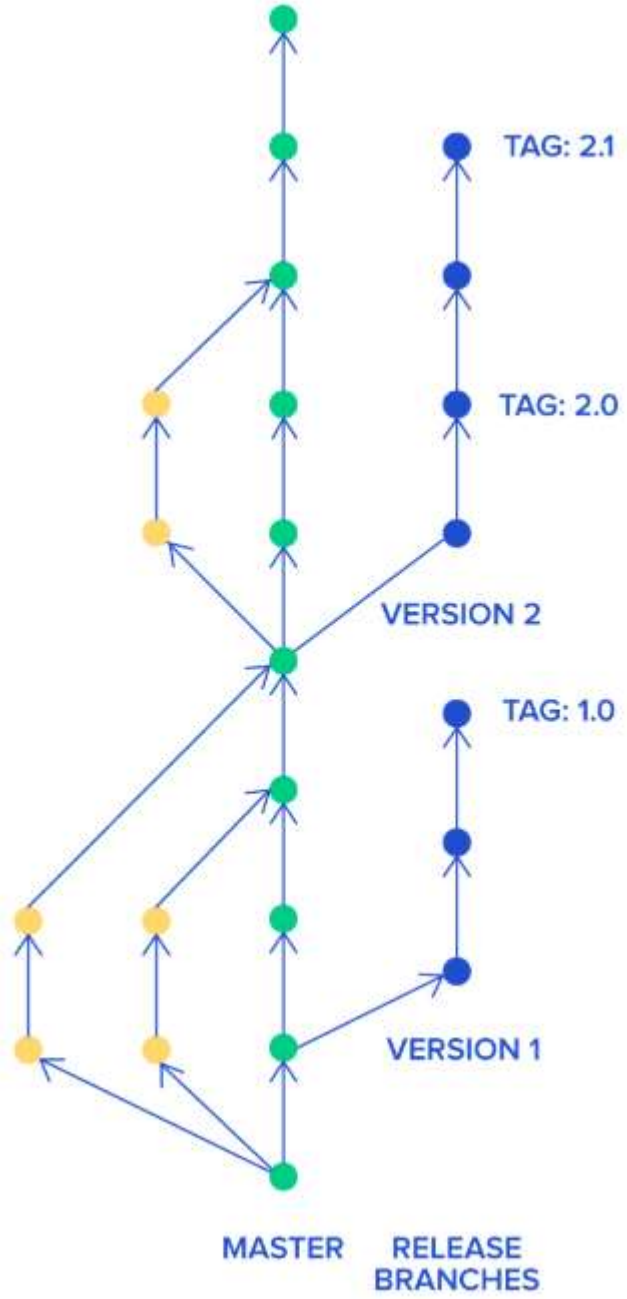

Figure 2.2 Trunk-based Development Workflow

On the figure 2.2 we can see main branch (“master”), marked green, feather branches with yellow dots. And release branches marked blue.

In the trunk-based development model, all developers work on a single branch with open access to it. Often, it’s simply the master branch. They commit code to it and run it.

It’s super simple. In some cases, they create short-lived feature branches. Once code on their branch compiles and passes all tests, they merge it straight to master. It ensures that development is truly continuous and prevents developers from creating merge conflicts that are difficult to resolve.[7]

The trunk-based development model is very simple, but we should keep in mind limitation and recommendation for implementation. We should use this model when we are:[7]

Just starting up. If you are working on your minimum viable product, then this style is perfect for you. It offers maximum development speed with minimum formality. Since there are no pull requests, developers can deliver new functionality at the speed of light. Just be sure to hire experienced programmers.

Need to iterate quickly. Once you reached the first version of your product and you noticed that your customers want something different, then don’t think twice and use this style to pivot into a new direction. You are still in the exploration phase and you need to be able to change your product as fast as possible.

Working with senior developers. If your team consists mainly of senior developers, then you should trust them and let them do their job. This workflow gives them the autonomy that they need and enables them to wield their mastery of their profession. Just give them purpose (tasks to accomplish) and watch how your product grows.

We must think twice before trunk-based development model implementation if we are:[7]

Run an open-source project. If you are running an open-source project, then Git flow is the better option. You need very strict control over changes, and you can’t trust contributors. After all, anyone can contribute. Including online trolls.

The team contains many junior developers. In this scenario trunk-based development model free of protection mechanism and error could be added very easy into master branch.

The second option is **Git Flow model**. The visualization of Git Flow model presented on figure 2.3.

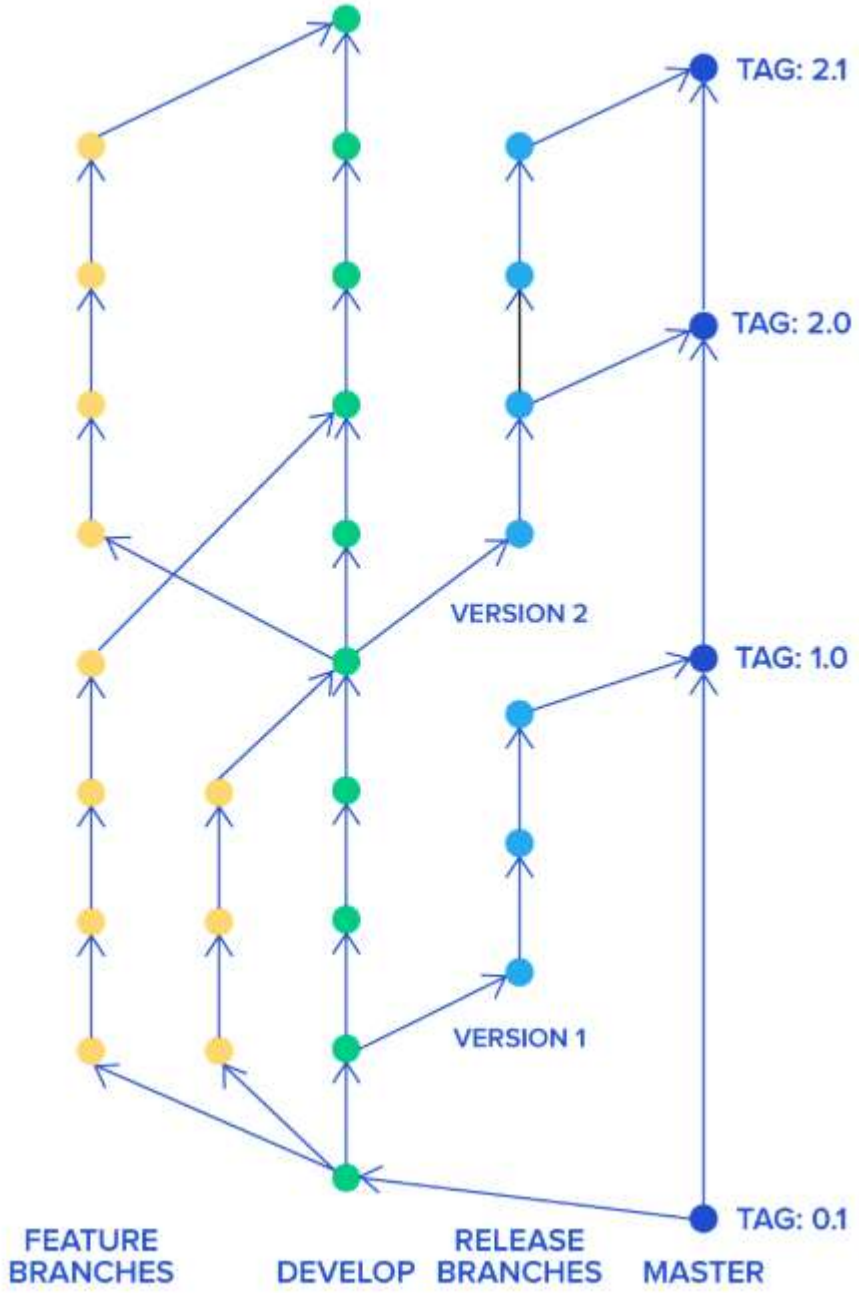

Figure 2.3 Git Flow model

In Git Flow model we have separated master from feather branch. And any changes before reaching master should be merged into release branches (in some scenario release branch are same and have no tags).

One of the advantages of Git flow is strict control. Only authorized developers can approve changes after looking at them closely. It ensures code quality and helps eliminate bugs early. However, you need to remember that it can also be a huge disadvantage. It creates a funnel slowing down software development. If speed is your primary concern, then it might be a serious problem. Features developed separately can create long-living branches that might be hard to combine with the main project.[7]

Like a trunk-base model, Git Flow also has recommendation and limitation. And we should implement it only if we need. Common cases for Git Flow implementation are:[7]

When we run an open-source project. This style comes from the open-source world and it works best there. Since everyone can contribute, you want to have very strict access to all the changes. You want to be able to check every single line of code, because frankly you can't trust people contributing. Usually, those are not commercial projects, so development speed is not a concern [7].

If you work mostly with junior developers, then you want to have a way to check their work closely. You can give them multiple hints on how to do things more efficiently and help them improve their skills faster. People who accept pull requests have strict control over recurring changes so they can prevent deteriorating code quality [7].

However, Got Flow has limitation. And we should avoid using it in next scenarios:

If we are just starting up, then Git flow is not for us. Chances are we want to create a minimal viable product quickly. Doing pull requests creates a huge bottleneck that slows the whole team down dramatically. We simply can't afford it. The problem with Git flow is the fact that pull requests can take a lot of time. It's just not possible to provide rapid development that way.[7]

If speed of development impacts us. Multiple branches and pull requests reduce development speed dramatically and are not advised in such cases.

Returning to CI system, in both scenario we should track the master branch changes. And high-quality CI is critical for truck-base workflow. For Git Flow the CI system should be much bigger and it should cover not only master but release and development branches. From configuration perspective it big but profit will be much bigger. High quality CI could train juniors and provide feedback on each branch where core committed.

2.4 Conclusion

Based on previous three section we have covered the best practices for Continuous Integration system. Explored most popular branching strategies and case for them implementation. From previous chapter we know three criteria for feedback to be useful:

- Any change, of whatever kind, needs to trigger the feedback process.
- The feedback must be delivered as soon as possible.
- The delivery team must receive feedback and then act on it.

And the Continuous Integration system meet all of them. For any change in tracking branch the feedback will be provided after build finish (usually it up to 20 min). The feedback will be delivered right in time, for example it might be email or even messenger. The last one could be covered like post action in branching strategies. The example is “no changes can be merged if build failed”.

CHAPER 3: Continuous Deployment

3.1 The deployment pipelines

Continuous integration is an enormous step forward in productivity and quality for most projects that adopt it. It ensures that teams working together to create large and complex systems can do so with a higher level of confidence and control than is achievable without it. CI ensures that the code that we create, as a team, works by providing us with rapid feedback on any problems that we may introduce with the changes we commit. It is primarily focused on asserting that the code compiles successfully and passes a body of unit and acceptance tests. However, CI is not enough.[2]

Continuous delivery is the next step of continuous integration in the software development cycle; it enables rapid and reliable development of software and delivery of product with the least amount of manual effort or overhead. In continuous integration, as we have seen, code is developed incorporating reviews, followed by automated building and testing. In continuous delivery, the product is moved to the preproduction (staging) environment in small frequent units to thoroughly test for user acceptance. The focus is on understanding the performance of the features and functionality related issues of the software. This enables issues related to business logic to be found early in the development cycle, ensuring that these issues are addressed before moving ahead to other phases such as deployment to the production environment or the addition of new features. Continuous delivery provides greater reliability and predictability on the usability of the intended features of the product for the developers. With continuous delivery, your software is always ready to release and the final deployment into production is a manual step as per timings based on a business decision.[8]

The benefits of the continuous delivery process are as follows:

- Developed code is continuously delivered
- Code is constantly and regularly reviewed
- High-quality software is deployed rapidly, reliably, and repeatedly

- Maximum automation and minimal manual overhead

The tools that perform continuous integration do the job of continuous delivery as well.

Continuous deployment is the fully matured and complete process cycle of code change, passing through every phase of the software life cycle to be deployed to production environments. Continuous deployment requires the entire process to be automated--also termed as automated application release--through all stages, such as the packaging of the application, ensuring the dependencies are integrated, deployment testing, and the production of adequate documentation for compliance.[3]

The benefits of continuous deployment and automated application release are as follows:

- Frequent product releases deliver software as fast as possible
- Automated and accelerated product releases with the code change
- Code changes qualify for production both from a technical and quality viewpoint
- The most current version of the product is ready in shippable format
- Deployment modeling reduces errors, resulting in better product quality
- Consolidated access to all tools, process and resource data leads to quicker troubleshooting and time to market
- Effective collaboration between dev, QA, and operation teams leads to higher output and better customer satisfaction
- Facilitates lower audit efforts owing to a centralized view of all phase activities

At an abstract level, a deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users. Every change to your software goes through a complex process on its way to being released. That process involves building the software, followed by the progress of these builds through multiple stages of testing and deployment. This, in turn, requires collaboration between many individuals, and perhaps several teams. The deployment pipeline models this process, and its incarnation in a continuous integration and release management tool

is what allows you to see and control the progress of each change as it moves from version control through various sets of tests and deployments to release to users.[1]

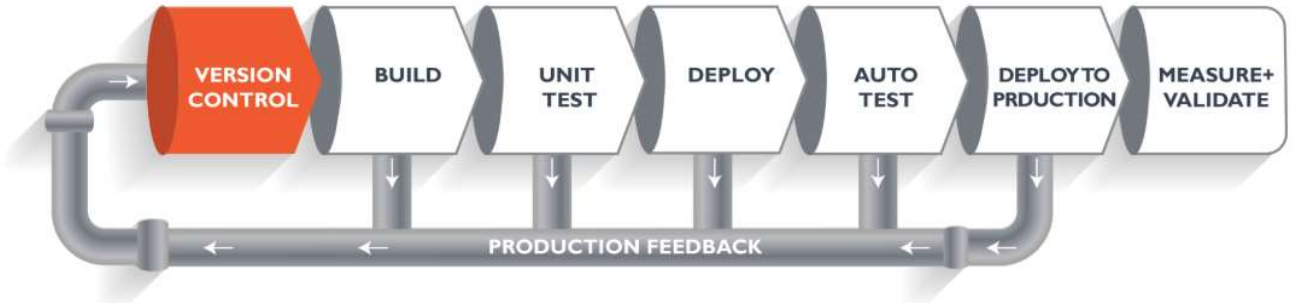

Figure 3.1 Basic deployment pipeline

The on figure 3.1 is a logical demonstration of how software will move along the various stages in this lifecycle before it is delivered to the customer or before it is live in production. The challenge on its way is infrastructure adaptation. This problem could be solved with Infrastructure as a Code approach.

3.2 Infrastructure as a Code

Infrastructure as code is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration. Changes are made to definitions and then rolled out to systems through unattended processes that include thorough validation. The premise is that modern tooling can treat infrastructure as if it were software and data. This allows people to apply software development tools such as version control systems (VCS), automated testing libraries, and deployment orchestration to manage infrastructure. It also opens the door to exploit development practices such as test-driven development (TDD), continuous integration (CI), and continuous delivery (CD). Infrastructure as code has been proven in the most demanding environments. For companies like Amazon, Netflix, Google, Facebook, and Etsy, IT systems are not just business critical; they are the business. There is no tolerance for downtime. Amazon's systems handle hundreds of

millions of dollars in transactions every day. So it's no surprise that organizations like these are pioneering new practices for large scale, highly reliable IT infrastructure. [10]

The Principles of Infrastructure as Code are:

Systems Can Be Easily Reproduced. It should be possible to effortlessly and reliably rebuild any element of an infrastructure. Effortlessly means that there is no need to make any significant decisions about how to rebuild the thing. Decisions about which software and versions to install on a server, how to choose a hostname, and so on should be captured in the scripts and tooling that provision it.

Systems Are Disposable. One of the benefits of dynamic infrastructure is that resources can be easily created, destroyed, replaced, resized, and moved. In order to take advantage of this, systems should be designed to assume that the infrastructure will always be changing. Software should continue running even when servers disappear, appear, and when they are resized.

Systems Are Consistent. Given two infrastructure elements providing a similar service—for example, two application servers in a cluster—the servers should be nearly identical. Their system software and configuration should be the same, except for those bits of configuration that differentiate them, like their IP addresses.

Processes Are Repeatable. Building on the reproducibility principle, any action you carry out on your infrastructure should be repeatable. This is an obvious benefit of using scripts and configuration management tools rather than making changes manually, but it can be hard to stick to doing things this way, especially for experienced system administrators.

Design Is Always Changing. With iron-age IT, making a change to an existing system is difficult and expensive. So, limiting the need to make changes to the system once it's built makes sense. This leads to the need for comprehensive initial designs that take various possible requirements and situations into account. Because it's impossible to

accurately predict how a system will be used in practice, and how its requirements will change over time, this approach naturally creates overly complex systems. Ironically, this complexity makes it more difficult to change and improve the system, which makes it less likely to cope well in the long run.

3.3 Release strategies

Release strategy is responsible for safety and reliable deploy to production environment. In that manner that will not impact end users or impact will be minimized. In modern world systems are running in 24/7 mode. This fact limits us in instrument and tactics for deployment. The solution for such case will be strategies with zero-downtime approach. However, in some situation the product owner could provide you some time frame, where system could be switched off. But it's rather exception than rule.

The blue green deployment One of the most effective ways to achieve a zero-downtime deployment is the Blue-green deployment technique. Blue-green deployment refers to a deployment solution that operates two identical hardware environments that are configured precisely the same way. One is referenced as “Blue” and the other as “Green”. One environment is active, the other remains idle. The basic idea in blue-green deployment is that you operate two versions of the website side by side one with the old build and other with the latest build. This technique is crucial for ecommerce businesses with critical uptime requirements. So how does it work? There is a load balancer, which forwards requests to the corresponding environment: production or pre-production. When an update is needed, you deploy it to the pre-production environment. Then you test it and switch the load balancer so that pre-production becomes the new production environment, and vice versa. [11]

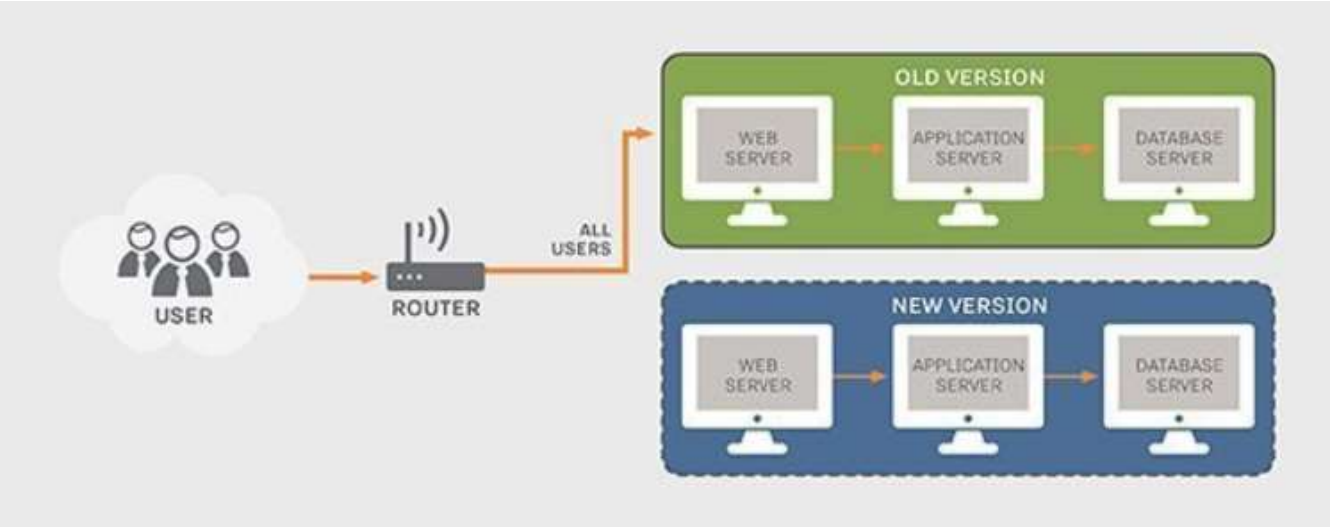

Figure 3.2 Blue Green deployment schema

Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody.[12]

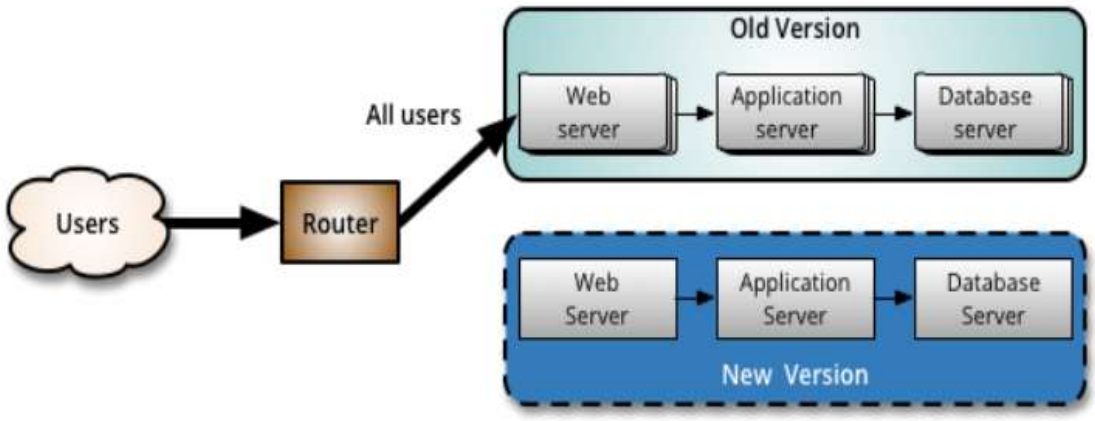

Figure 3.3 Initial state of Canary release

Like a Blue Green Deployment, you start by deploying the new version of your software to a subset of your infrastructure, to which no users are routed, as described at figure 3.3.

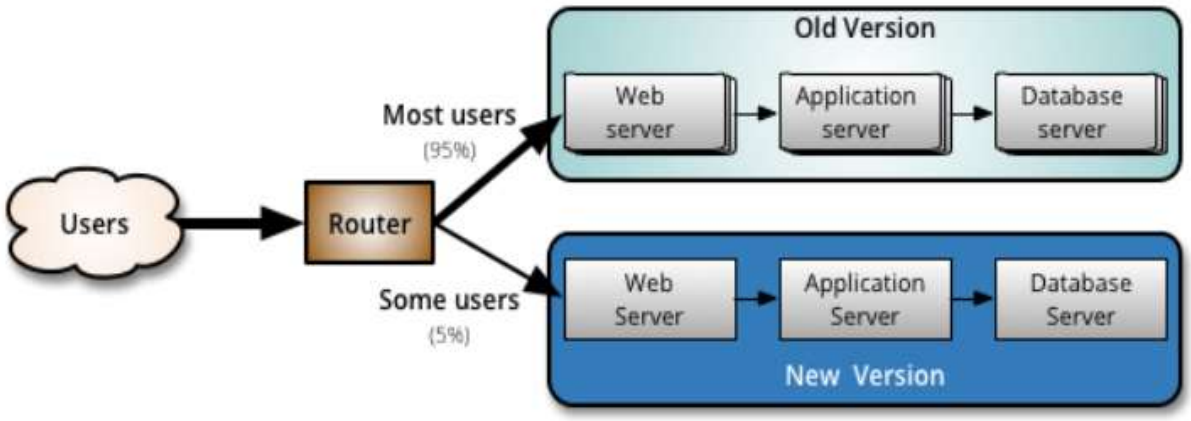

Figure 3.4. Middle state of Canary release

When you are happy with the new version, you can start routing a few selected users to it (Figure 3.4). There are different strategies to choose which users will see the new version: a simple strategy is to use a random sample; some companies choose to release the new version to their internal users and employees before releasing to the world; another more sophisticated approach is to choose users based on their profile and other demographics.[12]

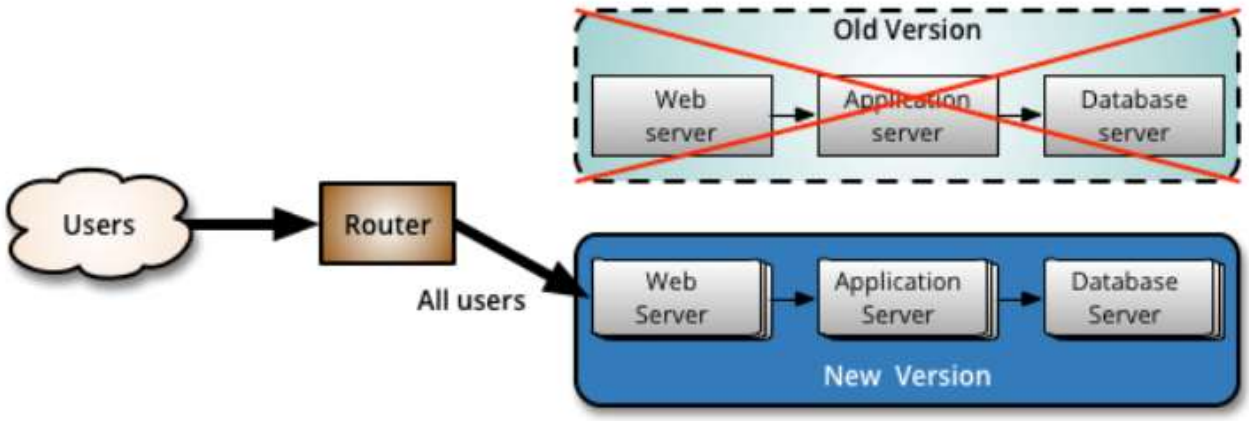

Figure 3.5 last stage of Canary release

After test finished the old version should be deleted as described at figure3.5.

Both the Blue Green deployment and Canary release provide zero-downtime for end users. And widely used in e-commerce and other 24/7 system. That approaches provide option to fast roll back in case of any errors. In Blue Green deployment we can skip switching if have problem, and in Canary release we can simply delete new version.

3.4 Implementation zero-downtime in Azure PaaS infrastructure.

In this section, we will cover simple implementation for zero-downtime deployment to Azure PaaS intrastation. This solution is operable for the last 2 years. However, the author is limited by NDA so much information (endpoint technologies customer sensitive information) will be not covered.

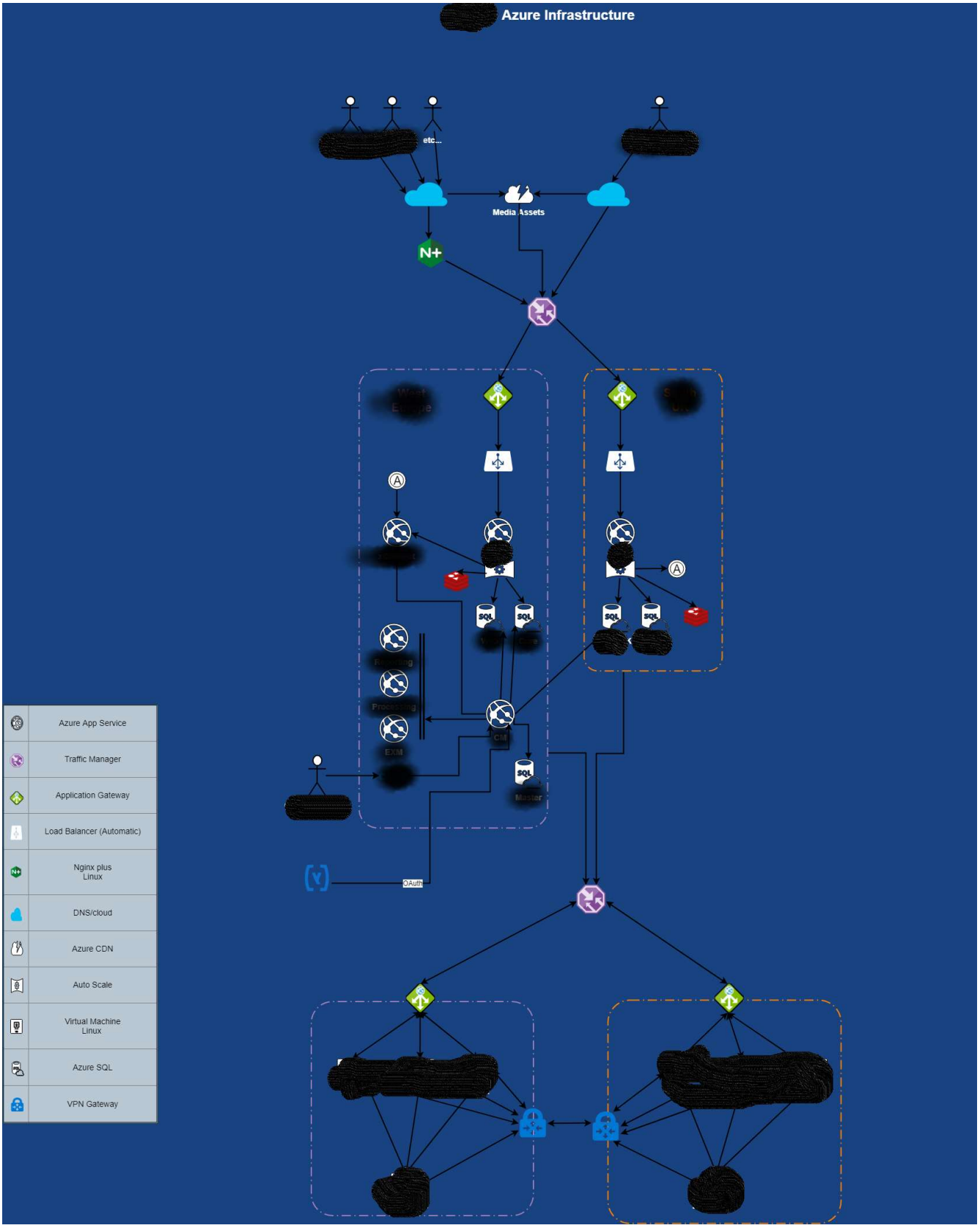
Azure Infrastructure
etc...
Media Assets
N+
SQL
OAuth
Azure App Service
Traffic Manager
Application Gateway
Load Balancer (Automatic)
Nginx plus Linux
DNS/cloud
Azure CDN
Auto Scale
Virtual Machine Linux
Azure SQL
VPN Gateway

Figure 3.6 Project infrastructure

As described on figure 3.6 project infrastructure are complicated and separated to two parallel parts with load balancing between. Application and presentation level aggregated inside Azure Web App. And database level present by Azure SQL. The project used .NET Core for back end and Node.js for frontend implementation.

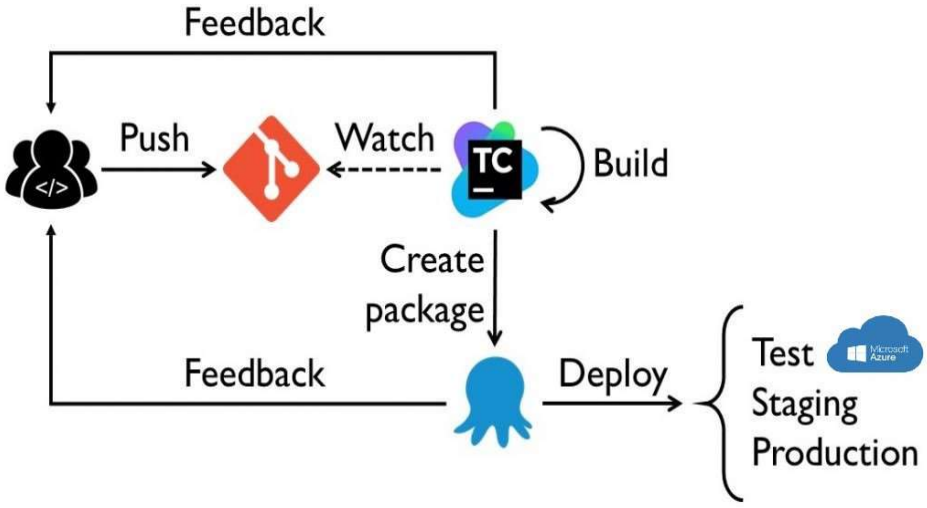

Figure 3.7 General overview of CI/CD system

Figure 3.7 describe key point of CI/CD system with corresponding tools and endpoints in cloud.

The CI system for this project were created on TeamCity and initial part for it was classic git flow with 3 important branches – dev, release, master.

The build process was organized in next order:

1. Build and run unit test for front end part. (command implementation *npm i -g & npm build& npm run test;*)
2. Run code check using SonarQube
3. Build and run unit test for back end part. (was implemented via Xunit framework and MsBuild.)
4. Run code check using SonarQube
5. Package artifact. (This task was done by TeamCity nuget integration.)

After the CI finished created artifact stored inside nugget server at Octopus Deploy

For deploy proposes were used Octopus Deploy solution. This choice was made by architecture team as Octopus Deploy easy to manage and interact with Azure PaaS elements such as Web App and Azure SQL.

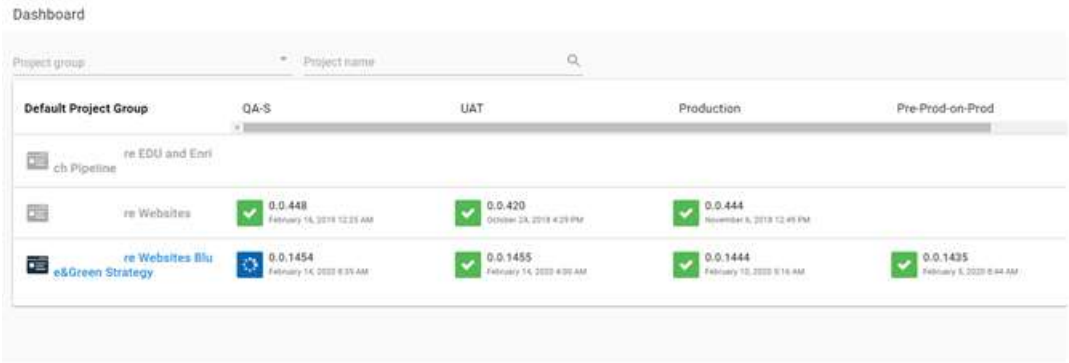

Figure 3.8 Octopus Deploy dashboard

Figure 3.8 described the general view of deploy instruments. When configuration finished the "deploy button" was delivered to the QA team and they launch deploy where they want.

The Blue Green deployment was implemented in that project. And it was done by using Azure web app feather – deployment slot. From deployment perspective the deployment slot it is the same web application but with trailing slot name in the end. For example, we have web application with name "*test-aplication*" and this application has the deployment slot called "*staging*". By default, Azure infrastructure will provide 2 URL: *test-aplication.azerewebservices.net* for web application and *test-aplication-staging.azerewebservices.net* for deployment slot. The one different between them will be application parameter, that translated to web app. It wasn't enough to make correct solution, as we should split the database. To achieve that connection string should be the source of information. For example, the main web app connected to *test-aplication-db_b* and staging slot has connection string to *test-aplication-db_g.* To get this information we should pars the connection string for active web app and for it slot to determine current DB steps. Script for this task below:

```
Function Get-Active-DB($ResourceGroupName, $WebAppName, $webAppSlot) {
```

```
$credentials = Get-AzureRmWebAppPublishingCredentials -ResourceGroupName $ResourceGroupName -WebAppName
$WebAppName -webAppSlot $webAppSlot
   $username = $credentials.UserName
   $password = $credentials.Password
   if(!$webAppSlot){
      $apiUrl =
“https://$WebAppName.scm.azurewebsites.net/api/vfs/site/wwwroot/App_Config/ConnectionStrings.config";
   }
   else{
       $apiUrl = "https://$WebAppName-
$WebAppSlot.scm.azurewebsites.net/api/vfs/site/wwwroot/App_Config/ConnectionStrings.config";
   }
   $base64AuthInfo = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{0}:{1}" -f $username,$password)))
   [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12

   Invoke-RestMethod -Uri $apiUrl -Headers @{Authorization=("Basic {0}" -f $base64AuthInfo);'If-Match' = '*'}  -Method
GET -OutFile ".\config_from_target.config" -ContentType "multipart/form-data";
   $connection = [xml](Get-Content .\config_from_target.config)
   foreach ($node in $connection.connectionStrings.add){

   if ($node.name -eq "$DbName"){
      $target_DB=$node.connectionString.split(";")[3].split("=")[1]
      }
   }
   return $target_DB
}
```

The next difficult was to automate DB replication from, active slot to staging one. The PowerShell script, for this task is below.

```
Function Replace-Green-With-Blue-DB {
  Param(
      [Parameter(Mandatory=$True)]
      [string]$ResourceGroupName,
      [Parameter(Mandatory=$True)]
      [string]$ServerName,
      [Parameter(Mandatory=$True)]
      [string]$GreenDbName,
      [Parameter(Mandatory=$True)]
      [string]$BlueDbName,
      [Parameter(Mandatory=$True)]
      [string]$PoolName
   )
$Database = Get-AzureRmSqlDatabase  -ResourceGroupName $ResourceGroupName -ServerName $ServerName -
DatabaseName $BlueDbName
   try{
      write-host "Clean up on SQL server old DBs, if present, will be deleeted"
      Remove-AzureRmSqlDatabase -ResourceGroupName $ResourceGroupName -ServerName $ServerName -
DatabaseName Old_$GreenDbName -ErrorAction Stop
      Write-Host "Sleep for 100 sec until DB remowed"
      Start-Sleep -s 100
   }
   catch {
      write-host "Already removed Temp DB for $GreenDbName"
   }
   try{
```

```
    write-host "Rename old DB"
    Set-AzureRmSqlDatabase -ResourceGroupName $ResourceGroupName -ServerName $ServerName -DatabaseName
$GreenDbName -NewName Old_$GreenDbName
  }
  catch{
    Write-host "OOOPS Green DB missing but it's not a problem "
  }
  try{
    Remove-AzureRmSqlDatabase -ResourceGroupName $ResourceGroupName -ServerName $ServerName -
DatabaseName Temp_$GreenDbName -ErrorAction Stop
  }
  catch {
    write-host "Already removed Temp DB for $GreenDbName"
  }
  Write-Host "Sleep for 100 sec until DB renamed"
  Start-Sleep -s 100
  Write-Host "Restore to Temp_$GreenDbName :"
Restore-AzureRmSqlDatabase -FromPointInTimeBackup -PointInTime (Get-Date).AddMinutes(-10) -ResourceGroupName
$ResourceGroupName -ServerName $ServerName -TargetDatabaseName Temp_$GreenDbName -ResourceId
$Database.ResourceID -ElasticPoolName $PoolName  -ErrorAction Stop
Write-Host "Renaming DB"
  try{
        Set-AzureRmSqlDatabase -ResourceGroupName $ResourceGroupName -ServerName $ServerName -
DatabaseName Temp_$GreenDbName -NewName $GreenDbName
  }
  catch {
    write-host "Something wrong another try to rename"
    Write-Host "But first sleep for 200 sec just to be on a safe side"
        Start-Sleep -s 200
    Set-AzureRmSqlDatabase -ResourceGroupName $ResourceGroupName -ServerName $ServerName -DatabaseName
Temp_$GreenDbName -NewName $GreenDbName
  }
  Write-Host "Deletting old DB"
  try{
    Remove-AzureRmSqlDatabase -ResourceGroupName $ResourceGroupName -ServerName $ServerName -
DatabaseName Old_$GreenDbName
  }
  catch {
    write-host "Already removed Old DB for $GreenDbName"
  }
  }
```

Call for this function is next format:

```
Write-Host "Started" (Get-Date)
$BlueDbName=Get-Active-DB -ResourceGroupName $ResourceGroupName -WebAppName $WebAppName -webAppSlot
''
write-host "Active DB is $BlueDbName"
$GreenDbName=Get-Active-DB -ResourceGroupName $ResourceGroupName -WebAppName $WebAppName -
webAppSlot $webAppSlot
write-host "Target DB is $GreenDbName"
Replace-Green-With-Blue-DB -ResourceGroupName $ResourceGroupName -ServerName $ServerName -GreenDbName
$GreenDbName -BlueDbName $BlueDbName -PoolName $PoolName
```

The Kudu console were used to interact with Web Application and corresponding slot. As a result, the reliable system was delivered to client and last deploy to production environment has number 245 as described at figure 3.9.

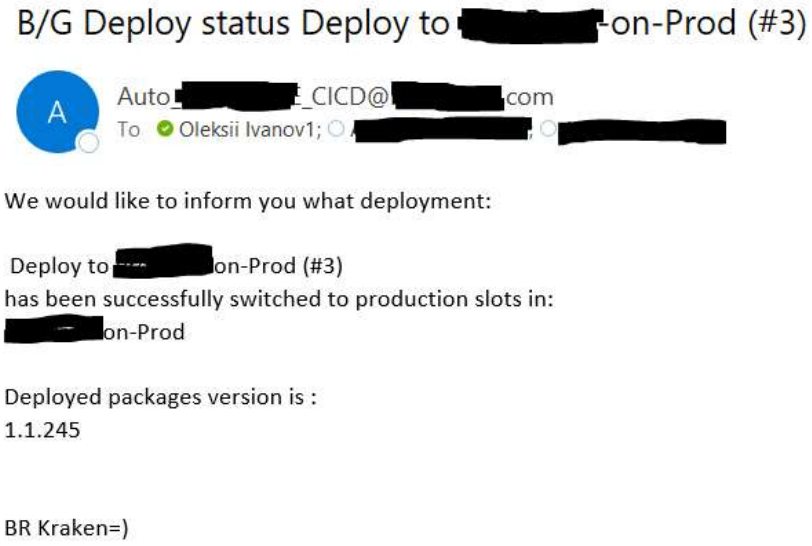

Figure 3.9 Sample mail notification after successful deploy

3.5 Conclusion

Based on previous four section we have uncovered difference between Continuous deployment and delivery. Covered most popular release strategies. Described main principles of IaC approach. Also were provided part of zero-downtime implementation on Azure PaaS infrastructure.

Summary

In the presented course work were described main approach for code delivering system, more famous like CI/CD system. Were described popular pattern and antipattern in CI implementation. Defined branching strategy as start point for CI systems. Covered Continuous Delivery and Deployment. Described Infrastructure as a Code (IaC) approach. Described most popular zero-downtime release strategies. And finally provided part of real life CD system and scripts for automation.

References

1. “Continuous delivery: reliable software releases through build, test, and deployment automation” / Jez Humble, David Farley/RR Donnelley in Crawfordsville, Indiana./ August 2010
2. “Continuous Integration, Delivery, and Deployment” / by Sander Rossel / Packt Publishing / 2018
3. “Effective DevOps” / by Jennifer Davis, Ryn Daniels / Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472./ May 2018
4. https://dzone.com/articles/what-is-continuous-integration-andontinuous-delive
5. https://www.martinfowler.com/articles/continuousIntegration.html
6. https://dzone.com/articles/continuous-integration-part-3-best-practices
7. https://www.toptal.com/software/trunk-based-development-git-flow
8. “DevOps: Continuous Delivery, Integration, and Deployment with DevOps” / by by Sricharan Vadapalli / Packt Publishing / March 2018
9. https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch
10. “Infrastructure as Code” / by Kief Morris/ O'Reilly Media, Inc./ September 2015
11. https://elogic.co/blog/how-to-achieve-zero-downtime-deployment-with-magento/
12. https://martinfowler.com/bliki/CanaryRelease.html