Lappeenranta-Lahti University of Technology LUT

School of Business and Management

Degree Program in Computer Science

Eemeli Manninen

Implementing a Continuous Integration and Delivery Pipeline for a Multitenant Software Application

Examiners: Professor Kari Smolander

M.Sc. (Tech.) Timo Storhammar

Instructors: Professor Kari Smolander

M.Sc. (Tech.) Timo Storhammar

ABSTRACT

Lappeenranta-Lahti University of Technology LUT

School of Business and Management

Degree Program in Computer Science

Eemeli Manninen

Implementing a Continuous Integration and Delivery Pipeline for a Multitenant Software Application

Master's thesis

41 pages, 4 figures, 2 tables

Examiners: Professor Kari Smolander

M.Sc. (Tech.) Timo Storhammar

Keywords: Continuous Integration, Continuous Deployment, Continuous Delivery, Pipeline, Multitenancy, Implementation, Design

Continuous integration and delivery are processes which allow software companies to automate the building, testing and installation of software solutions. This thesis designs and documents a CI/CD pipeline for Enerity Solutions Oy with the intent of allowing the company to automate its manual deployment processes in a multitenant environment. The pipeline was created as a design science artifact based on company interviews and the current processes of the field. The feature development and integration stages of the pipeline automatically build and test source code changes while the delivery stage enables a scheduled delivery process. The pipeline reduces risk of defects by introducing performance testing, database schema validation and environment replicability but requires upfront technical investment, may cause tenant system availability to suffer during implementation and may introduce over reliance on single pipeline components.

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT

School of Business and Management

Tietotekniikan koulutusohjelma

Eemeli Manninen

Jatkuvan integraatio- ja toimitusprosessin käyttöönotto moniasiakasympäristössä toimivaan ohjelmistoon

Diplomityö

41 sivua, 4 kuvaa, 2 taulukkoa

Työn tarkastajat: Professori Kari Smolander

Diplomi-insinööri Timo Storhammar

Hakusanat: Jatkuva Integraatio, Jatkuva toimitus, Julkaisuputki, Moniasiakasympäristö, Käyttönotto, Suunnittelu

Jatkuva integraatio ja -käyttöönotto ovat prosesseja jotka mahdollistavat ohjelmistokoodin automatisoidun kääntämisen, testaamisen ja julkaisun ohjelmistoyrityksissä. Tässä tutkielmassa suunnitellaan jatkuva integraatio ja -toimitusprosessi Enerity Solutions Oy:lle tarkoituksenaan mahdollistaa yrityksen ohjelmistotuotteiden käsin toimituksen automatisointi. Suunnitelma kehitettiin suunnittelutieteen teennöksenä jossa yhdistettiin yrityshaastatteluja ja nykyisiä alan lähestymistapoja. Tuotannon ja integraation vaiheet kääntävät ja testaavat ohjelmakoodia automaattisesti ja toimitusvaihe koostuu ajastetusta toimitusprosessista. Suunnitelma vähentää ohjelmistovirheiden riskiä mahdollistamalla suorituskykytestauksen, tietomallien validaation ja ympäristöjen kahdentamisen, mutta vaatii teknistä investointia, voi aiheuttaa ohjelmiston toimimaattomuutta käyttöönoton aikana ja voi johtaa yliriippuvuuteen yksittäisistä prosessin komponenteista.

ABBRIVIATIONS

SaaS – Software as a Service

B2B – Business to Business

B2C – Business to Customer

CI – Continuous Integration

CD – Continuous Deployment/Delivery

Table of contents

1 INTRODUCTION

Enerity Solutions Oy is a Finnish software company developing and hosting multiple multitenant electricity trade focused software-as-a-service (SaaS) web applications as the market leader of its field in Finland. The company is aiming to switch its software deployment processes from a manual process to the utilization of an automated deployment pipeline due. The change has been proposed due to the increase in the frequency of required software deployments, the number of tenants and new features between its multiple products. The proposed change should help control the software deployment process and increase the quality of software installations in the future.

This thesis focuses on documenting the industry practices of continuous integration (CI) and continuous deployment (CD), which are used as a part of a software lifecycle to handle frequent software releases, as well as design, document and evaluate one such process fit for the existing workflows and technical infrastructure of Enerity Solutions Oy. This thesis also takes into account the possible changes in day to day work practices when implementing a CI/CD solution as well as possible problem areas of such implementation both now and in the future.

The design is created in cooperation with various personnel from Enerity Solutions Oy to both collect requirements for the technical aspects of the design but also to help guide the design to make use of the existing technical infrastructure and work practices of the company. In addition to cooperation with the company personnel, the general industry frameworks, component interactions and technical considerations were gathered from current literature of the research field.

1.1 Thesis background

DevOps is an overarching set of software development principles aimed at speeding up the frequency of software feature integration, minimizing the risk of large scale conflicts during the merge process, keeping the release version of the software bug free and to automate the deployment of the software. These principles consist of regular software development, quality assurance and the deployment process of the software. The end goal of these principles is to have an always deployable, tested and working version of the software ready to be deployed whenever necessary. (Virmani, 2015) This can be achieved

by utilizing a process which automates and documents the flow of actions from a change in source code to a deployed software, a pipeline.

A software company, such as Enerity Solutions Oy planning to start utilizing DevOps principles in practice should have an existing software development process with support for not only the specification, design and development phases of a software project, but also allocated resources for evaluation and evolution of the process and the projects finished using these principles. (Samarawickrama, 2017)

1.2 Goals

This thesis sets up three goals. These goals are to:

1. Describe the environment and requirements of a CI/CD pipeline for a multitenant SaaS ASP.NET web application
2. Design and document a CI/CD pipeline for Enerity Solutions Oy
3. Evaluate the design against the initial requirements and industry standards

These goals were chosen with an intent of offering a concrete design for Enerity Solutions Oy with its decisions justified by both the current industry standards of the field and the current company environment. The reasoning of dividing the thesis into three distinct goals is to allow for a logical flow of design from industry standards and techniques to environmental requirements and core design into evaluation, reasoning and future proofing of the design. The first two goals are required to create the initial pipeline design for the company and the third goal adds conclusions to the concrete examples and allows for a more academic outlook of the problems stated in the rest of the thesis.

1.3 Structure of thesis

The remaining chapters of the thesis are divided into three parts. Firstly, the research methodology and data gathering methods of the thesis are introduced, and a theoretical look at current frameworks, methods and definitions of CI/CD pipelines are detailed. The second part focuses on documenting the business environment and requirements of a CI/CD pipeline for Enerity Solutions Oy, documents a reasoned design for such a process and evaluates it against the initial requirements of the company as well as against the best practices of the industry. Lastly, the future of both the pipeline design and multitenant environments in general are discussed.

2 RESEARCH METHODOLOGY

This chapter goes over the research methodology used in this thesis, the reasoning of why it was chosen, the data gathering methods used within the research methodology and the application plans for the gathered data.

2.1 Design science

Design science is a problem-solving paradigm which aims to satisfy organizational informational system needs by combining foundational theories, frameworks, instruments, constructs, models and instantiations to organizational environment problems with the aim of creating various artifacts to describe or solve these problems. These artifacts can range from completed software to formal logic, mathematical models and descriptions which allows them to fit the problem environment in the best possible way. (Hevner, 2004) These artifacts can afterwards be evaluated and modified further in order to react to possible changes in the original environment.

This thesis was chosen to utilize design science as its research methodology since it is answering a concrete business need for a real organization and the creation of a documented design artifact was one of the wanted outcomes. What sets this approach apart from regular software design, which could have been used instead to create the design in a purely practical organizational approach is that design science uses previous informational system theories in addition to existing technical knowledge of the developers to justify its design choices, allowing for a more objective solution to be created.

2.2 Research environment

The implementation artifact produced as a product of this thesis is created for Enerity Solutions Oy, which has multiple software products already in production. As a consequence of this, the company has already established most of the needed software development architecture such as source control, development frameworks, database implementations and cloud service architecture. This means that the design of the solution must to be designed around these environmental variables instead of expecting it to realistically change or redesign the existing company structure to create the most efficient, scalable and most technologically advanced theoretical solution.

2.3 Data gathering

The data used to design and implement the CI/CD pipeline was collected from two main sources: the existing foundational theories and models of the field via design science, and the requirements gathered from technology lead developers of Enerity Solutions Oy. This allowed the design creation to benefit from both the technical frameworks of the industry and the expertise, insight and strategic goals of the company to create the best possible design. The interviews conducted were structured to firstly describe the wanted outcome of the implementation project, giving an early direction for the design. Secondly, the interviews were used to accumulate technical and nontechnical requirements for the design as well as the reasoning behind these requirements. Thirdly, the interviews provided a list of possible technical pitfalls and possible future prospects for the software product lifecycle, which were important to keep in mind during the development of the design.

The first interview was conducted during the beginning of the thesis with the software development manager of Enerity Solutions Oy. This interview consisted of going over the technical details which should be taken into consideration as a part of the pipeline design. The main requirements concerned the performance of the automated build process in the context of current master commit frequency and size and the database update management as well as the handling of tenant-specific configurations in a multitenant environment as a part of the deployment process.

After the first interview during the design phase, a workshop was created to go help guide the selection of the CI platform. The workshop format was chosen because it allowed the technical knowledge of software engineers with practical knowledge of the software solutions to be used to spot and evaluate possible tool candidates best fit for the company. The workshop participants were selected to be three software engineers from Enerity Solutions Oy and the workshop consisted of two meetings and individual research between these meetings. The purpose of the first meeting was to collectively go over the most common CI platform tools used with SaaS solutions and filter the list roughly to end up with potential candidates worth further investigation. At the end of the first workshop meeting, these candidates were distributed along the participants with the intent of researching them further or even creating small proofs of concept of their properties with the end goal of figuring out their pros and cons.

The second workshop was scheduled a few weeks after the first one and it consisted of going over the potential candidates and figuring out which of them was the most work pursuing as the CI platform tool for the pipeline design.

After the selection of the CI platform tool, a second interview was conducted with the lead software engineer of one of the software solutions intended to be updated via the CI/CD pipeline. The interview consisted of collaboratively discussing the everyday software development workflow practices of Enerity Solutions Oy which would be affected by the implementation of the CI/CD pipeline. This both allowed the lead software engineer to be informed of the possible future changes as well as gave insight to the current workflow practices to be used in the future predictions of the pipeline design.

3 CONTINUOUS INTEGRATION AND DEPLOYMENT

This chapter takes a theoretical look on the different building blocks of a CI/CD pipeline from software development practices to deployment tools as well as researches and describes other software themes and processes which are relevant to the software infrastructure and business environment of Enerity Solutions Oy. These contain subjects such as software multitenancy and automated data structure change management.

3.1 Agile software development

Agile software development is a software development approach conceptualized in the late 1990s and early 2000s, which switched the focus of software development from complex methods to people and the interactions between them (Hoda, Salleh, & Grundy, 2018). In 2001s the Agile Manifesto agile methods were defined to focus on customer satisfaction by shortening the interaction time between versions of a software solution in development and advocate working software to be its primary measure of progress (Manifesto for Agile Software Development, 2019). The increase in customer satisfaction in turn caused the companies to generate more value to their customers and also to themselves making them more desirable for future endeavors. Furthermore the flexible structure allowed the companies to include Lean principles to their software development practices either by noticing probable problem areas faster due to the increased developer interactions and allocating resources there easier or by discontinuing parts of the previous development process which weren't beneficial to the value generated. (Alahyari, 2017)

Agile software development allowed companies to better respond to changing customer demands and help aid communication both internally and externally, since the rapid change management demands active interplay between different parts of a software company, which before might have been separate and had no shared responsibilities at all. (Rauf, 2015) The introduction of agile methodologies to the software business environment were deemed successful and according to the 13th Annual State of Agile Report agile was exercised in some form in 97% of all surveyed software organizations in 2019. (CollabNet, 2019)

In addition to the more flexible workflows and lower response times, agile software deployment practices also allowed companies to switch from project-focused to product-focused business models. This allowed the software lifecycle to be defined more loosely

instead of the previous strictly stage-focused structure. This change also allowed software teams to specialize to be software product-specific instead of specializing in any one process stage in the development of multiple software projects. This allowed for an easier division of responsibility and the application of technical expertise. (Dornenburg, 2018)

3.2 Continuous Integration

The change from infrequent changes of conventional software development methodologies to frequent version changes of agile software development methodology introduces a need for greater attendance of change handling and code testing for new versions of the software. This is to reduce the risk of introducing uncompilable code to the code base, since the increase in the number of changes inherently increases the change of developer errors. CI is a development practice which favors frequent integration of smaller features over large changes to reduce the severity of individual merge conflicts. (Arachichi & Perera, 2018) CI requires several common agile practices to be already implemented, such as single source code repository, revision control, build automation and unit testing. (Lavriv, Buhyl, Klymash, & Grynkevych, 2017)

CI is designed to be an unbiased judge of source code quality, meaning that all changes are examined equally without any temperance from for instance difference between different development workstations or environments. This is made possible by a centralized source code repository, which allows the rebuilding or retesting of any version of the source code whenever and wherever necessary. Depending on the solution, an external build tool might be required to build the source code such as the Microsoft Build Engine for C# applications (Meyer, 2014)

Build waiting time is a large part of a CI process since it defines how many builds generated from new features can be processed in a given time. In CI practices the lowest possible build time is not always the best, since the reduction of build duration at the cost of test coverage might compromise the integrity of the code base. (Erich, 2017) However, if the build takes too long to complete it can cause a bottleneck in the amount of processed changes by the pipeline as well as affect the overall process compliance of developers, since they have to wait for their builds to complete before continuing. Because of this, the final build time of a solution must be assessed by collecting feedback from both the developers and statistical history of failed and successful builds to find a balance between

efficiency and stability. (Laukkanen & Mäntylä, 2015) For a solution built multiple times a day, a build wait time should not exceed 10 minutes or be lower than 5 minutes (Rogers, 2004). In addition to the build time, the over addition of test suites can increase the maintenance requirements of the pipeline, possibly leading to a situation where the CI pipeline can fail for external reasons, halting all progress of pull requests which require a successful build to proceed. (Zampetti, Bavota, Canfora, & Penta, 2019)

3.3 Continuous Deployment and Delivery

The abbreviation CD is used in the other chapters of this thesis to address both Continuous Deployment and Continuous Delivery processes, since their differences are not consequential when describing the overall process flow and component interactions of the whole pipeline. Continuous Deployment and Delivery are both deployment approaches which handle the tested and production ready packages created during the CI phase of the pipeline and deploy them to any chosen environment, be they for testing or production. In addition the CD phase of a pipeline can also run any required installation scrips associated with that environment automatically which allows for the source control of the deployment. (Arachichi & Perera, 2018) The source control of deployment scripts and environmental configurations allow the process to be replicated, duplicated and developed further in a controlled manner.

Continuous Deployment and Delivery both move, unpack and install the built packages similarly, but they differ in the frequency and timing of the deployment. Continuous Deployment deploys the selected version of the software automatically every time a new package is created via the CI stage and pushed to the CD tool. This method is useful in a system where software updates are done multiple times a day, are required to be applyable to all solution users immediately and support constant software updates without risking solution availability for its users. In these environments the usage of a Continuous Deployment process requires an extremely reliable CI platform to ensure the validity of each change in the software.

Continuous Delivery focuses on scheduled or manually approved updates to production environments by not automatically deploying a version of software as it finishes the CI phase, but instead by storing the newest package and waiting the installation to be manually confirmed. This approach has the advantage of being viable in software

environments which do not support fluid installations of new versions of the software. To accomplish successful deployments in these types of environments, they can be scheduled during off-business hours for business-to-business (B2B) applications and deployed during scheduled maintenance windows with notice to users in business-to-customer (B2C) applications. The usage of a Continuous Delivery process allows for a more thorough testing of the staged software version by deploying the version to a testing environment and running manual performance and usability tests while waiting for the scheduled installation window for the production environment. Continuous Delivery also allows for the gathering of many small updates or parts of features to release them as one new version with manually created version notes detailing the most important changes in the update, allowing for more info to be passed to the users about things that were changed.

Even though the integration of a Continuous Deployment or Delivery process help ease the manual labor of software updates and introduce the positive requirement of frequent testing of the code base, its integration may also have drawbacks. In a case where the software code base is changed frequently but is not tested thoroughly enough due to old or missing testing practices or when the testing is not fast enough for the requirements of the pipeline, can the end result bring more harm than good to the overall customer experience. This can be especially true when the problems stem from environmental sources such as the production hosting environment configurations or customer data conversions, which cannot be tested by testing the source code. Furthermore, the integration of an automated delivery or deployment process also requires that all parts of the delivery process are done automatically and within the process, such as database scripts and schema updates. Having parts of the pipeline be manually controlled undermines the benefits of having an automated process, and in worst cases can make the deployment of the software even more time consuming since in addition to the manual tasks, the CD process itself requires supervision and maintenance. (Shahin, Ali Babar, & Zhu, 2017)

3.4 Multitenancy in software applications

3.4.1 Software solution multitenancy

Multitenancy is an architectural pattern approach used in software service hosting which allows multiple tenants to share hardware resources and even databases used to host an application with different, tenant-specific configurations (Bezemer & Zaidman, 2010).

These configurations can include service addresses, business logic paths or user interface changes. (Kang, Kang, & Hur, 2011) By customizing the tenant environments separately the software can offer user experiences similar to dedicated, single tenant environments while allowing for easier software upkeep and greater resource efficiency for the offering software company. (Liu, et al., 2014)

The usage of a single shared application instance for multiple tenants has multiple benefits when compared to more traditional approaches where each tenant has been dedicated their own software instance. Firstly, modern cloud-based software service hosting already uses pool-based resource grouping which allows the shared applications to be scaled up or down depending on the amount of active tenants at any time. This allows the hosting company to save on fixed hosting costs on times of low demand by automatically scaling the service down when it is not needed. (Kwok, 2008) The opposite of this would be to use a static amount of resources either in the form of a single virtual machine or a physical server. This decrease in operating costs can then be directly directed towards the cost for the tenant, making the cost-based competition with rival services possible. Secondly, having only one application instance shared between multiple tenants makes updating and maintaining it less time consuming, easier to replicate and manage, when comparing it to a process of having to update and maintain multiple environments, which might differ in configurations or hardware specification.

However, hosting a multitenant application will also introduce some drawbacks, such as the need for a built-in configuration management process for the application itself. This means that the application has to be able to modify its behavior, styles or functionalities depending on the tenant using it in order to offer an experience comparable to a self-hosted service while keeping the tenants separate from each other. The modification of these configurations for one tenant might require interruptions of the service itself while it readjusts to the new settings, potentially disrupting all the tenants using the shared instance of the service. (Bhuvaneswari & Saraswathi, 2014)

3.4.2 Workload distribution

A multitenant software application with multiple different components has to deploy them in a way in which the workload required from these components is distributed throughout its tenants. This is called degrees of isolation of the different software components. The

highest degree of isolation is hosting every tenant as their own instance in their own virtual machine without any resource sharing, and the lowest degree of isolation is having the tenants share all resources with others. (Ochei, Petrovski, & Bass, 2018) The process of finding a working degree of isolation in a multitenant environment requires finding a middle ground between high running costs and possible idling of having few tenants hosted together and individual performance loss or possible security threats of having too many tenants in a shared instance.

3.4.3 Data multitenancy

Another drawback of a multitenancy is that the handling of tenant specific data has to be done properly in an environment where multiple tenants share resources, software components or even databases. This means that different tenant configurations have to be managed and updated properly to make sure that tenants cannot access each other's data, adding to the maintenance costs of the system. The possibility of mixing up tenant data because of a bug in the software increases the operational risk of the system since any problems in the shared application will in the worst case permeate to all tenants using it. (Bhuvaneswari & Saraswathi, 2014) Because of this, special care must be placed on tenant management, such as adding tenants to or removing tenants from an already existing system.

There are three approaches to managing the data in a multitenant software which define how much of the data of the tenants is stored and managed in shared databases. The first approach focuses on isolation and security by storing a tenants data in their own database, avoiding any possible database specific overlap with other tenants. The second approach uses schema separation to store all tenants' data in a common database and focuses on allowing possible data aggregation while still only modifying the schema of a database query between different tenants. The third approach holds the tenants' data in a common database and only one schema, using custom identification to separate the tenant' data. This approach has the most responsibility placed on the application itself, relying on it to only handle the correct data with no internal data segregation from the database itself. (Bhuvaneswari & Saraswathi, 2014)

Data security has an increased role in multitenant software applications since the risk of individual tenants gaining access to other tenant's data by mistake or by exploitation is

increased when they share an environment (Mann & Metzger, 2017). The most effective way to counteract this would be to focus on the data isolation by supplying each tenant with their own database. This however could reduce the system's ability to utilize resource sharing thus leading to potential performance issues in some databases. Furthermore, sometimes a shared database is required in a multitenant software system for system wide configuration- or other non-tenant-specific information management.

3.5 Summary

A CI/CD pipeline builds upon an existing agile software development infrastructure of a software company by automating the frequent source code integration and upping the quality of the created features by systemically testing and validating every new pull request that is introduced to the code base. The integration of a CI process requires the evaluation of the frequency of code changes and the size of the complete solution and test suites to generate a suitable compromise of solution build time and test coverage to both keep the development process fluid and the quality of the created changes high.

The CD process requires the environmental attributes of the software solutions, such as multitenancy both for the deployed application as well as possible database structure changes to be taken into account to ensure that the final design can offer the most value possible. The introduction of a CD process can also affect the availability of the software solution during deployments so the design must be tuned to suit the needs of the tenants either by scheduling orderly maintenance windows for all tenants in a shared environment or using techniques such as rolling deployments to allow solution usability even during deployments.

4 SOLUTION DESIGN

This chapter focuses on the CI/CD pipeline design for Enerity Solutions Oy. The design is created by combining the design requirements generated from interviews and workshops with the industry standard methods, frameworks and suggested processes provided by the recent publications of the field. This chapter goes over the overall design environment of Enerity Solutions Oy, lists the concrete requirements of the design generated from the data gathering phase which guide the direction of the design, propose one solution design for the company and evaluates the created design against the initial requirements and aims to justify any deviations it might have from the literature best practices.

4.1 Design environment

The current workflow practices used at Enerity Solutions Oy focus mostly on the mix of regular small source code changes stemming from bug fixes which need to be deployed to production environments as soon as possible as well as from large feature updates which need to be thoroughly tested and which usually require database schema changes or scripts as a part of the deployment process. Both of these deployment types are currently done manually consisting of building of the solution, possibly cancelling any currently running scheduled tasks active in the deployment environment, running database schema change scripts to all tenant databases and the manual testing of the environments to make sure the update didn't cause any unforeseeable consequences linked to, for instance, tenant data or configurations. In addition these, steps have to be preferably done during office off-hours since updating the solution during the day will directly affect its availability in a negative manner. This style of workflow has worked in the past when the number of tenants and the amount of required fixes and features was relatively low, but as these numbers increase the workload and required frequency of updates rises as well.

Manual delivery for a commercial software product is one of the hardest parts of a systems development life cycle since it requires constant expertise and knowledge of the contents of every new software release to fix any potential problems during the deployment in addition to placing a lot of responsibility on humans, which are inherently prone to making mistakes. (Arachchi & Perera, 2018) Enerity Solutions Oy already uses most of the required elements for a CI/CD process, such as central code repositories, pull requests for feature updates, test automation and configuration management. This means that the

process of moving from manual updates to automated ones requires mostly linking of the pre-existing components to be a part of a controlled and monitored process.

The trend of incorporating software new tenants to use a shared multitenant cloud instance instead of offering tenant-specific environment hosting, as well as actively aiming to transfer existing tenants from tenant-specific environments to a shared one further adds to the importance and usability of a CI/CD pipeline now and in the future.

4.2 Requirements of the solution

The requirements of the solution listed in this chapter were generated primarily from interviews with personnel of Enerity Solutions Oy which were detailed in chapter 2.3. In addition to this, some of the requirements were generated from technical attributes and current processes of the solution environment.

4.2.1 Source code and CI

The solution design should be able to utilize the existing source code solution Bitbucket to fetch and build the selected solution whenever needed. The build of the solution should be linked to existing code review conventions, such as pull reviews to allow the buildability of the solution to serve as a required step in the merging of a new feature. Since the solution is estimated to be built many times a day by the build server, the individual build time of a single modified or merged pull request should not exceed 10 minutes.

4.2.2 Unit test support

Unit tests are a part of software products by Enerity Solutions Oy, but they are not used actively as a part of the manual deployment process. They are instead utilized during development of new features to ensure the buildability and continuous structure of the software. These tests should be able to be integrated into the new pipeline as a part of the CI build process to allow for real-time information of the performance as well as to serve as a regression test failsafe to ensure that the new feature does not affect other parts of the solution negatively.

4.2.3 Cloud connectivity and multitenancy

Enerity Solutions hosts most of its tenants using a SaaS model, and uses a shared cloud instance situated in Microsoft Azure to provide both the production as well as the testing

environment of the solutions. The design of the pipeline should allow multiple automated deployments to the shared SaaS instance located in Microsoft Azure.

4.2.4 Database updates

Database schema updates offer one of the biggest problems to modern CD processes since the schemas of SQL databases have to be matched to the current version of the software to allow the software to correctly communicate with the databases. This means that often the update of the database is required to be completed as a part of the software update to make sure that the software version and the database schema match. (Jong & Deursen, 2015)

In the case of Enerity Solutions Oy, the solutions are used only by B2B tenants which means that there is no requirement for a zero-downtime updates or undisturbed availability. This allows the software and database updates to be done using maintenance windows outside office hours, which has practically allowed the software to be manually updated in the past. The design of an automated pipeline must retain some of the most important steps done while updating the solution manually, such as monitoring the database update scripts for possible tenant-specific errors and checking the performance after possible changes to the database schemas of the solution. In addition the solution must still be able to be deployed even during business hours to fix immediate bugs which could compromise the data safety or business logic of the solution.

The database management of the multitenant software is currently done with a combination of using shared databases for environment configuration and tenant-specific databases for operational data management. This allows some of the environment-specific tasks and services to manage the tenants effectively but also provide the best possible separation of tenants' business-critical data inside the environment. This requires the pipeline to be able to automatically update both the shared and customer-specific databases with their required configurations.

Furthermore, the database updates must not be excessively time-consuming in a situation where there are no needed changes but the whole database schema is still updated. The implementation of a checksum system or a comparable tenant-specific management tool to keep track of needed database changes could save time in situations where the deployment version does not modify the database schema or add new data. Since many tenant database

updates are run sequentially using a shared tool, unnecessarily run database updates can add up especially as the number of tenants increases in the future.

Automated database updates have to also take into account the long-running tasks that are required for the functionality of the software which require an active database connection to function correctly. These include for instance daily measurement readings and business calculations run for all tenants of the software. The problem with these tasks is making sure that none of them are running while a update to the database is done since mismatching the executing code with the database schema mid calculation might lead to unwanted consequences such as damage to the tenant data in an event of an premature task interruption. For this reason a maintenance window which makes sure all running tasks have completed successfully should be used as a part of the update of the solution. This window must be sufficiently long to allow for all updates to finish but must also not interfere with the consistent competition of the rest of the tasks.

The validity of database updates for a given deployment version can be tested in a test environment which replicates the production environment as closely as possible. This environment can also be used to run performance tests to make sure that any changes which pass all unit tests and the review process of source control are not slowing down the solution when tested on a production-grade load and with production-style cloud database infrastructure which might function differently to local test databases.

4.2.5 Logging

Logging of the stages of the automated deployment process is paramount for the monitoring of the pipeline, especially in the early stages of implementation. The required levels of reporting generated by the pipeline should contain at least the integration build status and duration for each build, the deployment status and duration to the shared cloud instance and lastly the information from the performance tests run in the test environment. The amount of logging can be incrementally tuned down to only reports of possible performance decreases between builds or failures during automated deployments when the process is fully implemented.

4.2.6 Ease of use

Lastly the functionality of the automated pipeline must be accessible to the person creating a release. Since Enerity Solutions Oy must handle their deployment scheduling and feature

bundling manually in an organized manner, requiring maintenance windows whenever a new version is ready for installation, it cannot use Continuous Deployment as its installation strategy, which makes Continuous Delivery the preferred choice. This allows an version composed of many feature updates of fixes to be deployed whenever a release is deemed to be ready or when a critical bug has to be fixed immediately. The person creating the release must be able to choose the time the solution will be tested and updated with the wanted features and freeze the selected version, excluding all sequential changes from affecting the scheduled release version before it has been successfully delivered.

4.3 Implementation

This chapter details the process of creating a CI/CD pipeline process design for Enerity Solutions Oy from the choosing of the CI platform for the pipeline, to the description and integration strategy of all required pipeline components and finally detailing the process of the pipeline from a source code change to a software delivery.

4.3.1 CI platform requirements

Enerity Solutions Oy bases most of its software products on the Microsoft Azure cloud platform, which sets some restrictions on the choices of further technologies especially those related or designed for Amazon Web Services. One of the main goals for the design is the ability to integrate as many already existing systems as possible to the pipeline thus reducing the overall changes to the day-to-day workflow of the software developers.

The first step of selecting the pipeline components is to choose the CI platform to perform the core functionality of fetching source code, building and testing the solution and packaging the versions for deployment as well as to integrate to other tools readily used for the whole CI/CD pipeline process. The integration platform had several key requirements which are detailed in table 1.

Requirement	Priority
Integration to Bitbucket	Mandatory
MSBuild support	Mandatory
xUnit support	Mandatory
Octopus Deploy support	Preferred
Microsoft Teams support	Preferred
Email reporting	Preferred
Cloud hosting	Preferred

Table 1: CI platform requirements for Enerity Solutions Oy

Enerity Solutions Oy hosts its source code in a Git repository in Bitbucket, which makes integration between the CI platform and Bitbucket mandatory for the pipeline design. The solution deployed using the pipeline is a ASP.NET MVC web application, which makes both MSBuild, Microsoft's C# application build platform and xUnit, an unit testing tool for .NET applications the preferred tools for building and testing the software. The integrationability of these tools is mandatory for the pipeline.

Octopus Deploy is a CD platform which has previously been used in Enerity Solutions Oy during manual software deployments of the software solution to allow multiple distinct packages to be deployed to predefined locations at the same time as well as to automatically run the necessary deployment scripts as a part of the deployment process and manage the configuration management between different environments. Due to the existing experience with the tool and the already established deployment scripts, the integration of Octopus Deploy would be preferred for the pipeline. This is however not a mandatory requirement since the equivalent functionalities could be recreated in other delivery tools if nescessary.

All tools of the pipeline are preferred to be hosted off premises in a cloud instance, since the company does not host its software locally either. In addition to the mandatory integrations, reporting features to both email and Microsoft Teams, Microsoft's communications platform, are preferred to ease the tracking of the pipeline. Email can be

used to alert the service manager or their team of possible problems during deployment, and Microsoft Teams can be used to keep developers informed on the progress of their pull requests during feature development.

4.3.2 CI platform selection

Stemming from the gathered technical and non-technical requirements, a workshop consisting of three developers was formed with the intent of going over possible solutions and cataloging their pros and cons for further analysis. This research found four possible pipeline configurations for Enerity Solution Oy: Jenkins, Shippable, AppVeyor and Azure Pipelines. In addition to these choices, the possibility of the refusal of any automated deployment tool was also considered to ground the different choises to the existing environment more clearly. The positive and negative aspects of each choice are listed in table 2.

Tool	***Positive aspects***	***Negative aspects***
Manual deployment	Customizable for each deployment	Requires a person to supervise and deploy the software
	Human supervision for fast response for deployment errors	Deployment for multiple environments is time-consuming
		No automated builds to validate version stability
Jenkins	Support for Windows platform, Bitbucket and Octopus Deploy	Locally hosted
Shippable	Support for xUnit and Bitbucket	Requires additional configuration to support C#

	Cloud hosting	
AppVeyor	Support for Windows platform, Bitbucket and Octopus Deploy	
	Cloud hosting	
Azure Pipelines	Support for Windows platform, Bitbucket and Octopus Deploy	
	Cloud hosting	
	Included in Azure service already in use	

Table 2: Automated deployment pipeline tool comparisons for Enerity Solutions Oy

From the considered alternatives both AppVeyor and Azure Pipelines fulfilled all the given mandatory and preferred requirements with very little to no drawbacks while both Jenkins and Shippable had at least one missing mandatory or preferred requirement. This meant that only AppVeyor and Azure Pipelines were considered further.

The possibility of rejecting a CI/CD pipeline entirely and continuing to deploy the software solutions manually was also considered. Continuing to use a manual deployment process would save the company resources and development time which otherwise would have been used in the implementation of the pipeline and also allows for hands-on quality control of each deployment by a human which allows for fast response times for any problems during the deployment. However, a manual deployment process lacks the

scalability of an automated one and as the number of environments increases the replicability of the pipeline would allow for a much more standardised and stable deployment process for every environment. Furthermore, the decrease in introduced bugs and the early discovery of runtime errors via reviewed configuration changes will decrease the failure rate of deployments in the future which is much more favourable than being able to fix many more problems faster. For these reasons a manual deployment process was ruled out of the considered options.

The choice between the two remaining CI platforms was decided by the fact that Azure cloud services are already used by Enerity Solutions Oy to host its software products which would be affected by the introduction of a automated deployment pipeline. Therefore the introduction of a new Azure service in parallel to the already existing ones would be easier than establishing a new service. For these reasons Azure Pipelines was chosen to be the CI tool for the CI/CD pipeline.

4.3.3 Pipeline structure

The pipeline consists of three distinct phases relying on different tools: source control including code hosting and pull request management using Bitbucket Cloud, software builds and unit testing using Azure Pipelines and software installation and script execution using Octopus Deploy. The different parts of the pipeline are activated either automatically depending on the state of the code repository or manually during a deployment. The overall pipeline flow is illustrated in figure 1. The process starts with feature integration between the source control tool, Bitbucket and the CI platform, Azure Pipelines where an pull request update or merge triggers a new CI event. After a successful CI merge event the CD platform Octopus Deploy is provided with the new version of the packaged software. Both the CI and CD events are logged via Microsoft Teams. The deployment event to a Microsoft Azure hosted environment through Octopus Deploy is manually scheduled and is not automatically triggered by any action done by the CI platform.

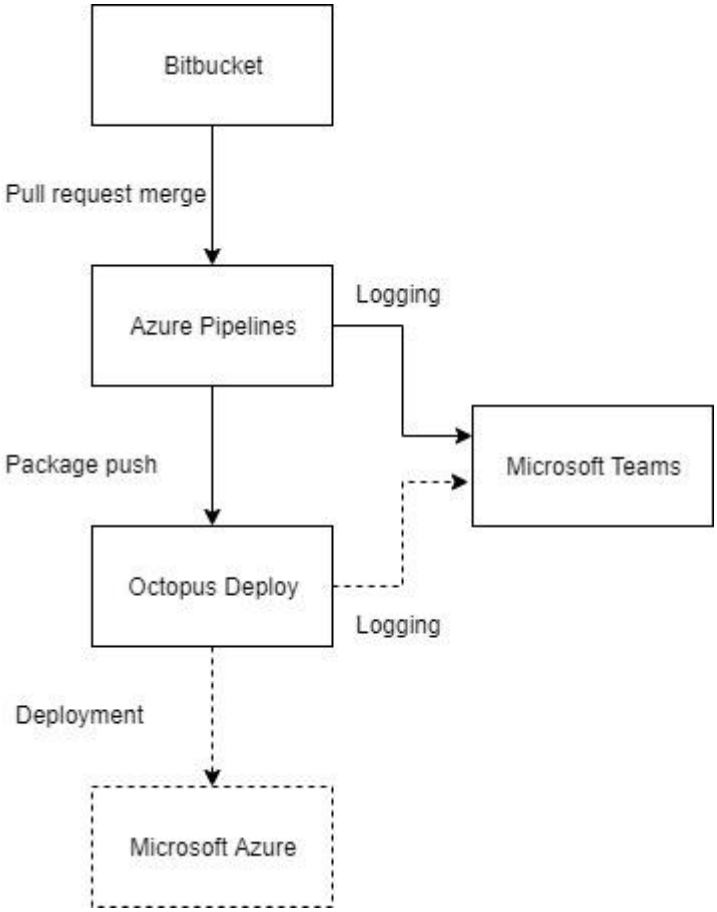

Figure 1: Pipeline components and process flow

4.3.3.1 Feature integration

The first phase of the CI/CD pipeline is feature integration which occurs when a new feature branch is created, the feature is developed and a pull request is created. The already existing development and testing phases as well as repository control of Enerity Solutions Oy do not need to change for this step to be implemented, only a connection from Bitbucket to Azure Pipelines is required to be created. The feature integration flow between source control and CI platform is pictured in figure 2. After the creation or modification of a pull request, Azure Pipelines is triggered and fetches the selected feature branch source code from the code repository, builds the solution, runs any unit tests associated with it and marks the pull request ready be merged if the build is successful. The merging process of successfully tested pull requests is not automated, since that would undermine the already existing code review conventions used at Enerity Solutions Oy which state that every new feature introduced to the code base must have at least two

manual approvals from other developers. In addition to this the build validation procedure makes sure that all manually reviewed changes are also a part of still compliable software solution. In an event where the pull request is modified after its creation the change triggers a new build and test process for that feature to make sure the changes do not affect the compatibility of the solution.

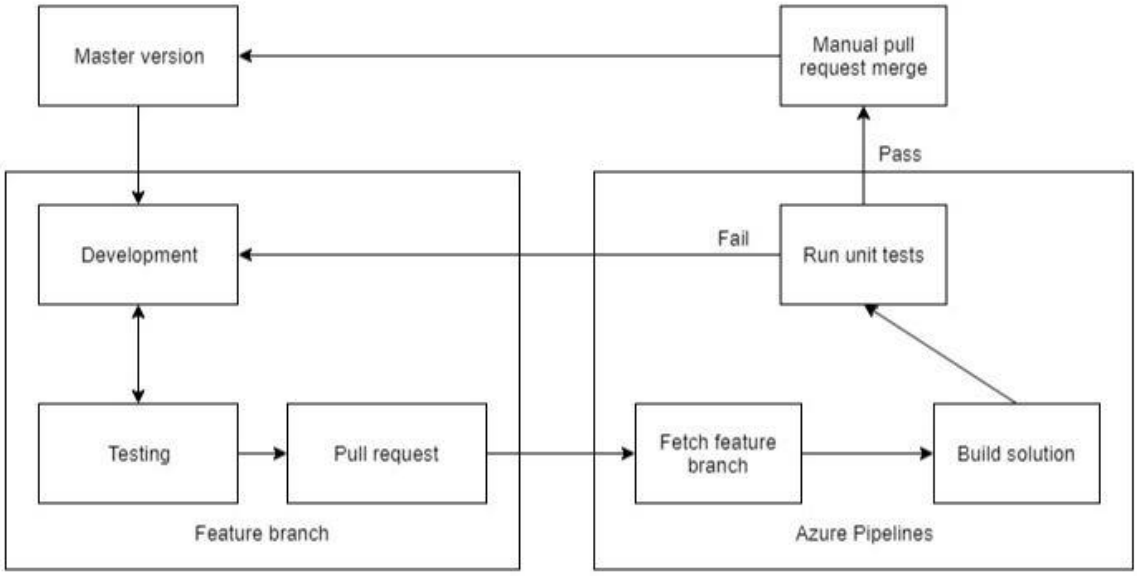

Figure 2: Feature integration flow between Bitbucket and Azure Pipelines

4.3.3.2 Feature merges

After a pull request created from a new feature branch has been approved by code reviewers and the build and test procedure has been successfully completed, the feature is ready to be merged to the master version of the code solution. After the merge has been made, another trigger for the build and test pipeline is made to make sure the merge was successful. The flow of master version build process is pictured in figure 3. This procedure is identical to the pull request creation and modification one, but it contains the stage to package up the application and push it to the CD tool Octopus Deploy to be used in the later stages of the pipeline. In a situation where the build and test procedures fail due to unseen conflicts in the merge process, the pull request can be manually reverted to return the master version of the source code back to a deliverable state.

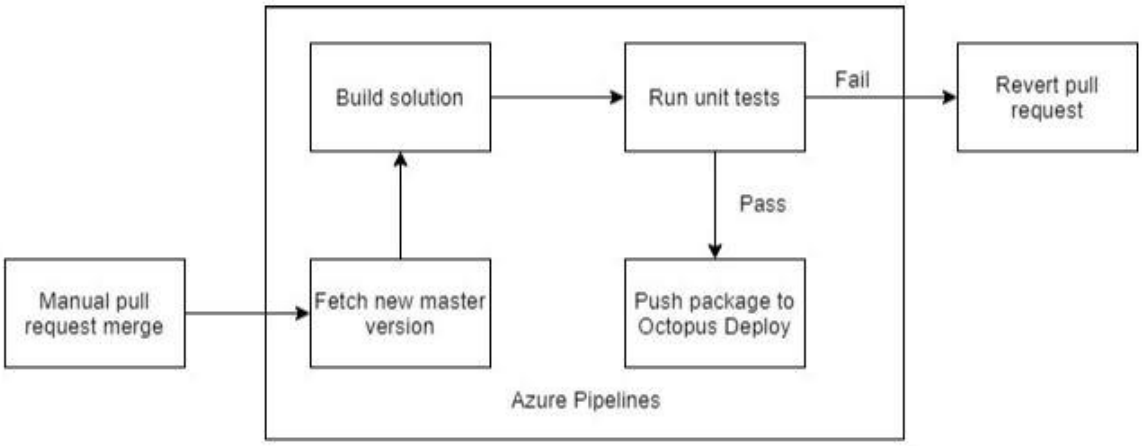

Figure 3: Master deliverability check and package delivery process

4.3.3.3 Version delivery

The final stage of the pipeline focuses on the deployment of the software to a multitenant Microsoft Azure environment and consists of deployments to both test and production environments. The solution deployment is a two-staged process meaning that in order to release a version to a production environment it must first be successfully deployed to a testing environment which has a configuration and database structure comparable to the production environment. In addition to just testing the validity of the solution as well as the build, test and packaging processes of the CI tool, the test environment can also be used to run tests to validate the performance of for instance some data-reliant tasks which cannot be reliably tested during development. In addition to performance testing, the testing environment also allows the testing of the installation scripts, database updates and internal setup changes of the delivery tool itself before installation to the production environment.

After the software is successfully been deployed to the testing environment, the software can be scheduled to be deployed to the corresponding production environment. This allows flexibility in the software deployment timeframes since the action doesn't require developers to oversee its progress, allowing for more solution availability for tenants during business hours. In an event of an unforeseeable error in the production deployment phase, the pipeline reports the incident both via email and Microsoft Teams. This allows for fast responses to the problems either by redeploying the software automatically using

the pipeline or in a worst case scenario reverting back to manual update process in the case where the pipeline itself is causing the installation problem. This delivery flow is pictured in figure 4.

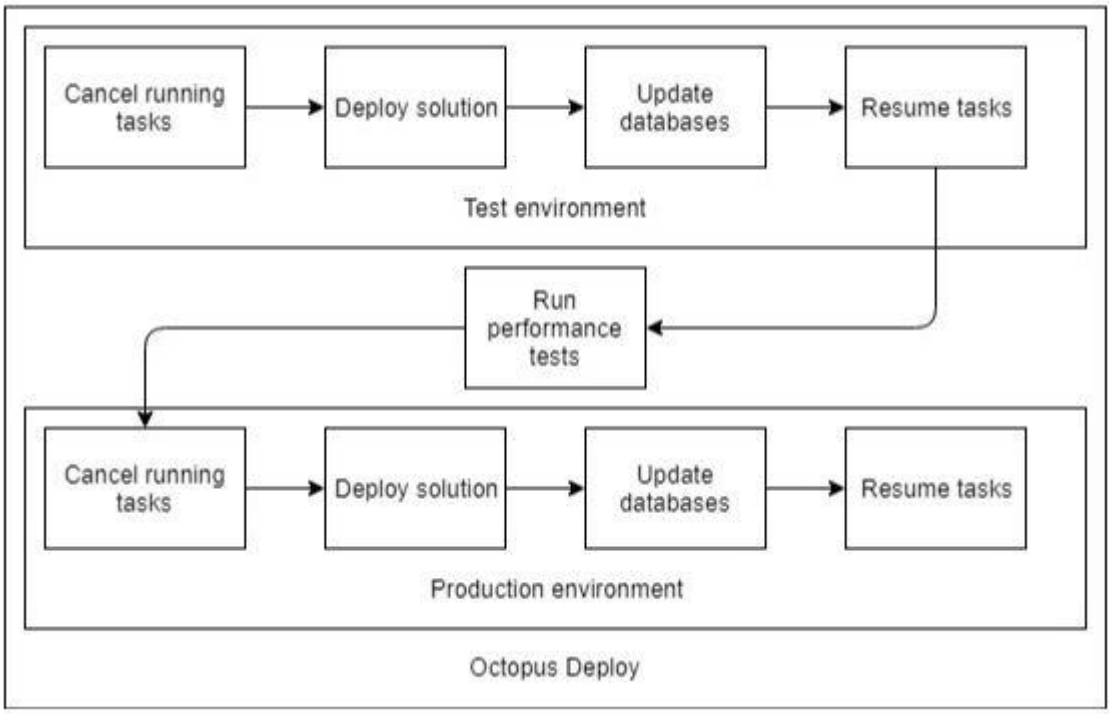

Figure 4: Version testing and delivery

4.4 Evaluation

This chapter evaluates the proposed design through the viewpoints of initial requirement coverage, the changes in day to day work practices of the development process and the possible negative side effects of the implementation of a CI/CD pipeline.

4.4.1 Initial requirement coverage

The previous manual process of deploying new versions of software focused on building the solution once before every deployment and manually testing its performance, which was both time consuming and prone to human errors. The proposed pipeline solution was designed to primarily ease this burden on deployers by allowing a working version of the software to be deliverable on a scheduled time and minimizing the risk of deployment errors or performance problems in production environments while utilizing already

existing source control, delivery and communication tools of Enerity Solutions Oy. These requirements have been fulfilled by the pipeline design since the cyclic nature of pull request testing and verification and the two-staged delivery process offer flexibility and security in all stages of the pipeline.

4.4.2 Changes in work culture and practices

The new pipeline design has some required changes on the daily development practices but the bulk of the changes are centered around software deployments. One of the concrete changes for developers is the addition of environment spesific configurations used by the CI/CD pipeline to the code repository to. This change allows the software to be always deliverable to a new environment if something would to happen to already existing production environments since the configurations are not stored within the environments itself. Additionally, forcing any changes to enviroment spesific configurations to be done as part of the normal development process allows them to be reviewed like any other changes, decreasing the risk of an environment braking because of a faulty configuration file. The passive changes to a developer workflow include the addition of automated builds for every pull request a developer creates. This assures the quality and buildability of the new feature with every small change to the source code which might otherwise be too small to be tested manually and could slip past the review process. This addition can also be seen to passively push the quality of the code changes up since every addition will trigger the testing of the whole solution which gives instant feedback to the developers and encourages them to double check their changes next time.

The largest change to work practices caused by the addition of a new pipeline is targeted towards the delivery process and the engineers responsible for them. Firstly, the new pipeline frees engineers from manually installing the software during off-business hours and running and monitoring all deployment scripts individually tenant by tenant. Secondly, the pipeline documents each delivery individually for future reference and also can spot differences in performance during the delivery by running the version first through a test environment designed to resemble the production environment as closely as possible. Thirdly, the design of the pipeline allows the environmental configuration changes to be monitored like any other source code change which in turn decreases the risk of them causing problems in the deployment process.

4.4.3 Negatives of implementation

The introduction of new technologies can cause negative consequences to a normal workflow of a company. Firstly, time and resources must be spent to design, develop and implement the pipeline solution and integrate it as a part of the companys daily practices. This includes moving resources away from normal software development work which could generate more instant revenue for the company, whereas the benefits of the pipeline might not be so obvious and increase productivity and quality only in the long run.

Secondly, the introduction of the pipeline to development or production environments might cause interruptions of their normal operations during the initial implementation phase of the pipeline where any environmental difficulties get resolved. This might result in multiple failed deployments while the correct environmental settings get development through trial and error. This problem is further aggrevated by the fact that all the changes required for testing the validity of the pipeline must be processed through the normal code change management process, since they are modified with the rest of the solution code base. This problem could be mitigated by creating additional pipelines for testing purposes, but these might be only partially successful since existing production environments might still have differences when compared to new test environments which can not be reasonably accounted for.

Thirdly, the developers of the company have to take into account the added steps to the development process, such as the requirement of making sure all created pull requests are compatible to the pipeline process and utilize it to its full potential. This can cause some loss of efficiency for the software development process in the very beginning of the introduction of the pipeline while developers readjust to the new conventions. In addition to the additional steps required and the learning nescessary, the actual build times added to the pull requests might slow the previously instantanious merges down, especially when large amount of small incremental changes are made to existing pull requests, each triggering a new build of the solution. This can however be countered by allowing multiple builds per pipeline to be run sequentially as long as they are not the ones required for the merge of the pull requests, though they might drive the cost of pipeline maintenance up along with its performance.

Fouthly, the over reliance of external automated systems can cause problems in cases where the pipeline system or some of its components malfunction or otherwise lose their ability to provide their required service. In these situations, if a version update is absolutely still required, a manual process might still be needed.

4.5 Summary

The proposed CI/CD pipeline design aims to replace manual deployments in Enerity Solutions Oy by offering a structured and monitored way of making source code changes, building and testing them with both unit and performance tests suites and installing a working version of the software whenever needed with attention paid to the existing environmental requirements of the company. This design requires work, resources, work culture changes and possibly risky software solution desing changes to implement and further constant care and maintenance to be able to provide a stable and reliable tool for deployment engineers. The pipeline is designed to provide long term value for both the developers and the company by easing the day-to-day operations of the deployment engineers as well as lowering the risk of errors in both the development and delivery phases of the pipeline. The addition of the pipeline also allows the company to expand and add new tenants to its multitenant environments easier when the process is partly or completely automated.

5 DISCUSSION

Just as the software products which can be improved through incremental change via the CI/CD pipeline, the pipeline itself must be able to be improved upon. These changes can occur as a consequence of modifications to software architecture, the different tools used as a part of the pipeline or external requirements from the company or its developers. This chapter focuses on the potential future changes for both Enerity Solutions Oy and its products, as well as the proposed CI/CD pipeline and its role in the process of change.

5.1 Multitenancy in the future

After the implementation and introduction of a CI/CD pipeline, it has to be maintained and ready to be developed further along with the increase of tenants and the introduction of new technologies and company trends and needs. One of the most likely developments would be to introduce the CI/CD pipeline designed for B2B tenants to the software solutions which serve B2C tenants as well. This would require a much greater potential for system availability meaning that the automated software updates must not disturb the functionality of the solution in any way. One of the potential solutions is to host the solution on multiple mirror servers which can be used interchangeably. In addition of balancing the load of users on multiple servers, this can be used as a platform for a rolling software update. A rolling update allows multiple versions of a software to coexist, and the tenants can be redirected to use the updated version once it has been successfully updated on at least one server. (Sun, et al., 2014)

This however can introduce problems to the database update process, since the databases must remain accessible and functional during potentially large schema updates to work with both versions of the software across all the staging servers. These schema changes, such as adding new columns to a table or updating column default values can cause the whole table to refuse any new statements during the schema update, resulting in wait times and in worst cases inoperative software for the tenants. One of the solutions for dealing with table locking schema changes is to develop the software to support mixed-state databases, which allow it to operate on multiple database schemas at once. (Jong, Deursen, & Cleve 2017) This however would require large changes in the software components which communicate with databases to allow support to many different concurrent schemas of the databases.

Another challenge of a continuously evolving database schema is the ability to rewind possible database schema changes along with the software version without loss of possible user made changes to the database data after the intial update. This might be required in an event of a faulty deployment process which prevents the partial usage of the software. This can however be extremely difficult since the user changes to a database might not be able to be converted back to the previous schema of the database leading to a loss of data which in a business-critical software system might be unacceptable.

5.2 Future development of pipeline

The future developments of the pipeline depend mostly on the future of the software solutions it is used to deploy. If the software solution integration and delivery process works flawlessly there will probably be no external motivation to change it outside of quality of life changes, such as the introduction of a more detailed reporting suite. However, there are a few additions to the pipeline which could be considered in the future.

The first possible addition is the utilization of automated version notes. The current practice for informing solution tenants of new features is to manually collect all merged features from the latest release and to manually compile a list of their descriptions to be sent to the tenants. This process could be automated by adding the step of creating version notes for the current feature as a part of the pull request process. This way a descriptive note of the merged feature will always be added, and the version release notes can now be compiled from these.

Secondly, the process of user interface testing could be integrated alongside performance testing to the CD process of the pipeline. This would allow the testing of user interface performance as well as changes in used client-side library versions and their effect on the functionality of the solution. The addition of client-side testing would round up the coverage of the testing suites of the solution and allow the validity of the user interface to be checked without the need of deployment engineers and possible defects to be fixed before any tenants are affected.

5.3 Thesis retrospective

The goals of the thesis were firstly to describe the environment and requirements of a CI/CD pipeline the context of a software company specializing in multitenant cloud

applications. The environmental context and requirements were gathered via interviews and a workshop conducted at Enerity Solutions Oy and also by surveying and researching the relevant and recent publications of the field. The interviews and workshop were successful and provided plenty of requirements and discussion to use in the design of the pipeline. The literature research provided ample theory on the fundamental building blocks of a CI/CD pipeline in such an environment such as agile development, multitenant software architecture and database schema validation, which were used to create and future proof the pipeline design.

The selection of the different components of the pipeline were a combination of the existing technical infrastructure of Enerity Solutions Oy and the new tools introduced and researched during the organized workshop. After the selection of Azure Pipelines as the CI tool, the rest of the work was focused around getting the already existing tools integrated to form a complete pipeline process which could be used to practically complete software deliveries in a way which provides value to the company and the users of the pipeline.

This thesis could have been expanded further by addressing the implications of a CI/CD pipeline on an enterprise level, meaning the interconnectivity between different software products which all utilise the same delivery process. The possible resource sharing benefits, but also the security drawbacks could have been examined to figure out if there is a balance between solution-spesific deliveries and the entanglement of different software solutions.

6 CONCLUSIONS

CI and CD are processes which allow the automatization of source code testing and validation in addition to the installation or the software. This thesis created a pipeline design for Enerity Solutions Oy which was created by gathering requirements stemming from the existing solution environment, used components and services, and the future vision of the delivery process and the software products. The pipeline design in composed of Bitbucket Cloud, Azure Pipelines, Octopus Deploy and Microsoft Teams which allow support for both the solution build tool, Microsoft Build Engine, as well as the unit test tool, xUnit, and also offer adequate information communication in all phases of the pipeline. The design of the pipeline is divided into three distinct phases which are the feature integration-, feature merge- and version delivery processes. These processes together allow the automatization of pull request validation via solution building and unit testing as well as provides a platform to schedule software version updates whenever they are deemed most fitting for the solution.

The proposed pipeline design supports the continuous testing and validation of the code base health by running at least two automated build and test suites on every new feature to the software code base to ensure the buildability of the solution during the automated process. Furthermore, the delivery phase of the pipeline design also supports dedicated testing environments which mimic the production environments of the solution as closely as possible. The testing environments allow the validation of database schema and environmental configuration changes as well as the running of performance tests to spot any defects which could cause loss of availability in the production environments.

The drawbacks of the pipeline design are mostly focused on its workload requirements, since the pipeline requires multiple new and existing processes to be integrated to a single process with support for possibly multiple software products. Moreover, the introduction of a new pipeline design can cause tenant availability and the success rate of deployments to lower during the integration stages since the deployment processes have to be verified in each environment separately.

7 REFERENCES

Alahyari, H. (2017). A study of value in agile software development organizations. *The Journal of Systems & Software, 125*, 271-288.

Arachchi, S., & Perera, I. (2018). Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. *2018 Moratuwa Engineering Research Conference (MERCon)* (pp. 156-161). Moratuwa: IEEE.

Arachichi, S. A., & Perera, I. (2018). Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. *2018 Moratuwa Engineering Research Conference (MERCon)*, 156-161.

Bezemer, C., & Zaidman, A. (2010). Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)* (pp. 88-92). Antwerp: ACM.

Bhuvaneswari, T., & Saraswathi, M. (2014). Multitenant SaaS model of cloud computing: Issues and solutions. *2014 International Conference on Communication and Network Technologies* (pp. 27-32). Sivakasi: ICCNT.

CollabNet. (2019). *13th Annual State Of Agile Report.*

Dornenburg, E. (2018). The Path to DevOps. *IEEE Software Vol 35(5)*, 71-75.

Erich, F. (2017). A qualitative study of DevOps usage in practice. *Journal of software: Evolution and Process 29, no. 6.*

Hevner, A. R. (2004). Design Science in Information Systems Research. *MIS Quarterly, 28*, 75-105.

Hoda, R., Salleh, N., & Grundy, J. (2018). The Rise and Evolution of Agile Software Development. *IEEE Software, vol. 35, no. 5*, 58-63.

Jong, M., & Deursen, A. (2015). Continuous Deployment and Schema Evolution in SQL Databases. *2015 IEEE/ACM 3rd International Workshop on Release Engineering* (pp. 16-19). Florence: IEEE.

Jong, M., Deursen, A., & Cleve, A. (2017). Zero-Downtime SQL Database Schema Evolution for Continuous Deployment. *39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)* (pp. 143-152). Buenos Aires: IEEE.

Kang, S., Kang, S., & Hur, S. (2011). A Design of the Conceptual Architecture for a Multitenant SaaS Application Platform. *2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering* (pp. 462-467). Jeju Island: IEEE.

Kwok, T. (2008). A Software as a Service with Multi-tenancy Support for an Electronic Contract Management Application. *2008 IEEE International Conference on Services Computing*, 179-186.

Laukkanen, M., & Mäntylä, E. (2015). Build Waiting Time in Continuous Integration -- An Initial Interdisciplinary Literature Review. *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering* (pp. 1-4). Florence: IEEE.

Lavriv, O., Buhyl, B., Klymash, M., & Grynkevych, G. (2017). Services continuous integration based on modern free infrastructure. *2017 2nd International Conference on Advanced Information and Communication Technologies (AICT)* (pp. 150-153). Lviv: IEEE.

Liu, J., Di, Z., Liu, S., Pu, C., Wu, L., & Pan, L. (2014). Finding Optimized Deployment Strategy for Multitenant Services by Iterative Staging. *2014 Asia-Pacific Services Computing Conference* (pp. 129-135). Fuzhou: IEEE.

Manifesto for Agile Software Development. (2019, 6 19). Retrieved from https://agilemanifesto.org/

Mann, Z. Á., & Metzger, A. (2017). Optimized Cloud Deployment of Multi-tenant Software Considering Data Protection Concerns. *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (pp. 609-618). Madrid: IEEE.

Meyer, M. (2014). Continuous Integration and Its Tools. *IEEE Software, vol. 31, no. 3*, 14-16.

Ochei, L. C., Petrovski, A., & Bass, J. M. (2018). Evolutionary Computation for Optimal Component Deployment with Multitenancy Isolation in Cloud-hosted Applications. *2018 Innovations in Intelligent Systems and Applications (INISTA)*, 1-7.

Rauf, A. (2015). Gap Analysis between State of Practice and State of Art Practices in Agile Software Development. *2015* (pp. 102-106). Washington, DC: IEEE.

Rogers, R. (2004). Scaling continuous integration. *5th Internationl Conference Extreme Programming and Agile Processes in Softwave Engineering* (pp. 68-76). Garmisch-Partenkirchen: SPRINGER-VERLAG BERLIN.

Samarawickrama, S. S. (2017). Continuous scrum: A framework to enhance scrum with DevOps. *Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)* (pp. 1-7). Colombo: IEEE.

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access, vol. 5*, 3909-3943.

Sun, D., Bass, L., Fekete, A., Gramoli, V., Tran, A. B., Xu, S., & Zhu, L. (2014). Quantifying Failure Risk of Version Switch for Rolling Upgrade on Clouds. *2014 IEEE Fourth International Conference on Big Data and Cloud Computing* (pp. 175-182). Sydney: IEEE.

Virmani, M. (2015). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, 78-82.

Zampetti, F., Bavota, G., Canfora, G., & Penta, M. D. (2019). A Study on the Interplay between Pull Request Review and Continuous Integration Builds. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 38-48). Hangzhou: IEEE.