CONTINUOUS INTEGRATION IN WORDPRESS PLUGIN DEVELOPMENT

Patric Manner
Student number: 37208

Master of Science (Technology) Thesis
Supervisor: Dragos Truscan
Åbo Akademi University
Faculty of Science and Engineering
Spring 2020

ABSTRACT

Continuous integration is an agile software development practice that enables teams to work efficiently together. It is first and foremost a mindset, which encourages developers to integrate often to transform historically time and resource consuming software integrations into quick and smooth tasks. Over the years, however, tools have been developed that make the integrations even more effective. In this thesis, continuous integration was introduced as a part of the development of a WordPress plugin, Aucor Core, through a pipeline, where many of the tools, such as unit tests, integration tests and a remote integration server, were used to enhance the process by providing automation and fail-safes. A thorough analysis of the subject matter was done in the context of the company producing the plugin. The analysis, then, formed the base for the practical implementation of the pipeline. The implementation was divided into three stages: extending the local development environment to include testing, writing the tests for the plugin's features and setting up a remote integration server using Travis CI. The purpose was to reduced risks during development, increased confidence in the plugin and favor a more active development of it.

Keywords: agile methods, continuous integration, testing, wordpress, plugin

CONTENTS

LIST OF FIGURES

Glossary

CD	Continuous Deployment. A practice in Agile software development expanding CI and CDE to automatically deploy changes when they successfully pass the pipeline.
CDE	Continuous DElivery. A practice in Agile software development expanding CI to always have a stable deliverable ready for deployment.
CI	Continuous Integration. A practice in Agile software development emphasizing frequent integrations.
CMS	Content Management System. An application that provides an interface for creating and managing online content.
CSS	Cascading Style Sheets. A language for styling HTML elements on websites.
GUI	Graphical User Interface. A graphical interface for users to interact with a program.
HTML	HyperText Markup Language. Markup for documents displayed in web browsers.
HTTP	HyperText Transfer Protocol. An application-layer protocol for hypertext distribution.
JS	JavaScript. A scripting language for dynamic and interactive front-end actions, but can also be used as a back-end language.

JSON	JavaScript Object Notation. A data format that stores human-readable text in concise key-value pairs.
MVP	Minimum Viable Product. A concept when creating a product to only include the bare essentials for it to work, which is later fleshed out by incremental improvements.
PHP	PHP: Hypertext Preprosessor. A back-end scripting language used mainly in web development.
SSH	Secure SHell. A network protocol that provides a secure channel for communication over unsecured networks through encryption.
TCP	Transmission Control Protocol. A network protocol that lets parties exchange data, with guaranteed order of delivery of packets.
YAML	YAML Ain't Markup Language. A human readable data serialization standard.

1 INTRODUCTION

People who have taken part in software development know that there are usually multiple ways of achieving the same goal, and among those possibilities there are generally ways that are better, faster or more efficient than the others. These best practices [3], as they are commonly called, range from designing and writing the actual code to the processes of putting the code together to form a cohesive product, such as project management and quality control. In an ideal world, everyone would be using all of the best practices all the time, but in reality, this is rarely achieved. However, the way a development team works can be positively influenced by using just some of the practices and maybe later down the line put the team on the right path to adopt more of them. The drawback with best practices and why they are most commonly ignored is that they often require some sort of extra setup and work, not directly related to furthering the task at hand. A good example is testing, which in the long term is an undeniable gain for any software project, but requires a developer to do something else than developing new features for the software. In other words, best practices are an important asset to build maintainable, extendable and overall high quality software, but require a fair amount of discipline and energy to introduce in a project [4]. Once in place and a part of one's daily routine, though, they are definitely worth the initial effort.

1.1 Goal of the thesis

In this thesis, a form of best practice, known as continuous integration (CI), will be discussed and implemented in a real working environment, as a proof of concept for the company Aucor Oy [5]. Since its popularization as one of the Agile practices in the late 1990's, CI has been embraced by the software development industry for its proposed ability to increase testing agility, software predictability and communication between team members [6], but requires much more from a company than just adding another tool to their developers' arsenal. The prerequisite for CI to work at a basic level is arguably only that the people involved need to have or are willing to acquire a mindset to integrate often [7]. Of course, with proper tools and systems in place the activity is elevated to a whole other level, but the spirit of continuous improvement and willingness of a team to better themselves and their work, which is the prevalent force in Agile, is why companies can thrive by working with CI and other Agile methods. Before focusing on the technical aspects, a closer look will be taken at Agile development methodologies (from which Aucor has taken inspiration to improve on their operations already), which showcase the correct attitude and mentality needed to make the most out of CI. Although seemingly straight-forward, there is not a singular way of implementing CI [8], and it induces (or requires) significant individual as well as organizational change to successfully be adopted [9].

Aucor Oy is a digital agency that specializes in creating custom websites and digital services with the open-source content management system (CMS) WordPress [10]. As such, they develop and maintain customized themes and plugins, many of which are used on multiple websites. For everything to run smoothly for the company and its customers, the products need to be dependable and satisfy their intended purpose, which in turn means that they need to be thoroughly tested. While testing cannot prove the absence of bugs [11], it can increase confidence in the product and the integration of its different parts. As it stands now, most of the testing during the products' development is done manually. This may be viable in smaller, simpler software, but as the size of the products and the amount of them increases, it also becomes increasingly difficult to make sure that everything is being tested and working properly, not to mention the time it takes to do it all by hand. This is where CI comes in, and the goal is to improve the current development process in the following two ways:

- Creating a test suite
- Configuring a third party CI environment for test automation

For the scope of this thesis, creating a test suite means writing unit and integration tests. As will become apparent in later chapters, in WordPress theme and plugin development the distinct line between the two types of tests is more blurred than usual, but in general terms that is the nature of the tests to be written. The tests will create a safety net moving forward in the development process by providing regression testing, ensuring that everything still works as it did before implementing any new features. The automation of those tests (or any part of development that allows it) will be an integral part of reducing unnecessary errors caused by manual mistakes (which is very human) during the testing and speeding up the process. Configuring a third party CI environment is mainly to enable the automatic testing, but also to remove an all too prevalent phenomenon in development, which is software working on one developer's machine, but not on another.

The pipeline will be introduced for the development process of one of Aucor's products, the Aucor Core plugin [12], which is a WordPress plugin that brings with it critical site functionality. This plugin was chosen as the target of the thesis because of its free availability on GitHub [13], its wide usage and critical nature. The last part is especially important, as it is potentially a single point of failure for sites using it. With the pipeline in place, the developers should feel more at ease to more actively develop the product, without the fear of making changes that potentially break something and noticing it too late due to lack of awareness and improper testing. In summary, the stop criteria for the thesis will be a test suite and complete path coverage of the Aucor Core plugin's features, an automated testing environment using a third party CI server provided by Travis CI [14] and a passing build.

1.2 Thesis structure

The thesis is divided into chapters so that the background of the relevant terminology and concepts involved are discussed first: the philosophy of Agile and CI, basic theory about testing and web applications, and finally more concrete and more domain-specific subjects, such as WordPress and plugin development, in chapter 2. Chapter 3 will consist of preliminary analysis and assessment of how to introduce a CI pipeline in the context of Aucor as a company and producer of software, which mirrors Aucor's current development practices to conducted studies on CI to form an initial structure and base for the implementation. Chapter 4 will describe the steps taken, tools used and the thoughts behind making the decisions to implement the pipeline. A closer look at the Aucor Core plugin will also be taken in this chapter. The remaining chapters 5 and 6 will contain the analysis of the work done, results achieved, evaluation of the approach and a discussion on where to continue from here.

2 BACKGROUND

As a company working in the highly competitive industry that is web development, Aucor needs to be constantly improving and upholding a level of quality to be able to keep growing and gaining customers. With websites and services created (and maintained) now counted in the hundreds, it becomes increasingly important to have systems in place that focus the energy of the developers on the right matters, i.e. activities that create value for the customers and keep them satisfied. To help achieve this, Aucor has streamlined their daily work by adopting techniques from the Lean philosophy [15] as well as several Agile principles. While Lean is more about end-to-end management, the Agile principles already in use in Aucor have laid a good foundation for which to continue expanding with other Agile practices and disciplines, such as CI.

2.1 Agile methodology

At the turn of the millennium, a new software development approach, called Agile, was put together by a team of industry experts [16]. It has gained popularity ever since for its ability to rapidly respond to changes and succeed in the uncertain and constantly evolving environment that is software development [17].

2.1.1 Origin

Up until the mid 1990's, software development had been treated as any other branch of engineering, but after decades of building expensive, unwanted and unreliable software, people were starting to realize that changes were needed [18]. Developers were growing tired of corporate bureaucracy and processes for processes' sake, when they really just wanted to make good products for customers and work in environments that did not just talk about people being important, but actually acted like people are important [19]. Various practitioners, from developers inside software companies to outside consultants, started experimenting and combining new ideas to form methodologies that worked in different development situations. Realizing that others may also be interested in such methodologies, frameworks were created based on the working combination of ideas to be spread to teams in other organizations and contexts [17]. Frameworks such as Extreme Programming, Adaptive Software Development, Crystal and SCRUM arose as a result [19]. What most of the frameworks encapsulate is essentially the idea that "teamwork makes the dream work" and that no hardwired processes are superior to that of a well-functioning team [18]. This was put into words on a February weekend in 2001, in a ski lodge in Utah, where 17 proponents of these new ways of developing software met to discuss the values and goals they commonly shared [19]. The result was the Agile Manifesto.

2.1.2 Principles of the Agile methodology

The Agile Manifesto is short and sweet:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more." [16]

The manifest was later expanded upon with the 12 principles of Agile [20]. Looking at the manifesto (and the principles that followed) one can see that Agile is first and foremost a mindset. It is the abstraction of all the more practical concepts people usually associate with Agile: frameworks such as Scrum and Extreme Programming, and practices such as test-driven development and pair programming [17]. The concepts and tools all channel essential parts of Agile in their own way and provide good platforms from where to start working more "Agile", but they are only useful if one understands the thought processes behind them. Organizations blindly following agile practices too literally, instead of living up to the values, rarely succeed in bettering their ways [17]. The essence of Agile is creating value for the customer with iterative and fast-phased workflow and focusing on the people behind the work [20].

The value of a software company to a customer is working software. "Working" can mean different things to different stakeholders, but to customers working software needs to satisfy all of their constantly evolving needs. This is where Agile shines compared to a more classical approach, such as the waterfall model [21]. Change is welcomed at every stage of development, because the stages consist of short development cycles that produce small pieces of working software at a time [20]. The short cycles, called sprints, force developers to focus on a smaller number of tasks (ideally only one) at a time, and delivering the work in small batches enables faster feedback from the customers. In addition to simply being able to supply the customers with the changes that they want, both of these factors contribute to minimizing unnecessary or unwanted work (non-valuable work for the customers) during development [18]. Despite being keen on speed and rapid iterations, Agile does not compromise on quality, because quality is value and actually enhances agility [20].

The cornerstones of Agile are the people, which the manifesto makes clear by claiming "individuals and interactions over processes and tools" as the first statement. A big part of software development is problem solving and according to the principles of Agile, problems are best solved by self-organizing teams and motivated individuals, who are trusted to handle the different situations [20]. The people are allowed to express themselves through innovations and reflect on the work done, which creates an environment of continuous improvement in the processes and the products. Interactions in the teams and with the customers are also of upmost importance, as it allows for transparency and up-to-dateness regarding the progress and requirements of a project [18].

2.1.3 Agile software development cycle

In the context of the software development life cycle, the workflow of Agile would look approximately like in figure 2.1. Differences and divergences on a detail level on how to achieve the next step can be seen in the variety of frameworks applying the methodology, but they all follow the basic ideas of what the steps are [22].

Agile works according to a minimum viable product (MVP) development strategy, which means that at first the software only contains the bare minimum it needs to func-

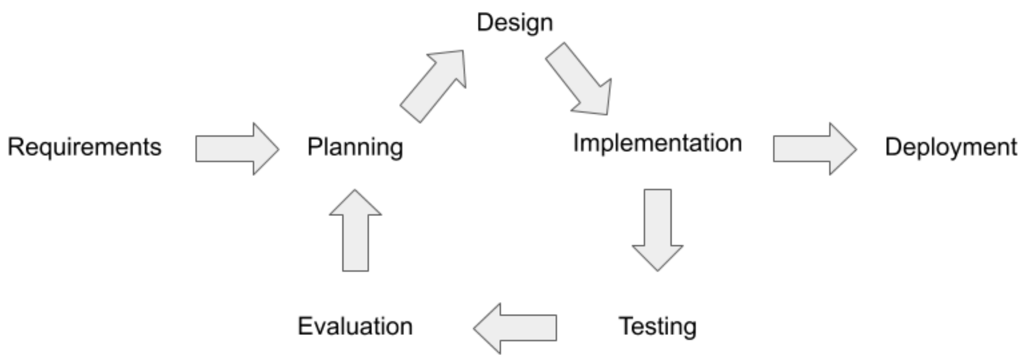


Figure 2.1: Agile workflow

tion and nothing more [23]. The minimal core is then iteratively expanded upon, each iteration only adding the least amount of code needed to enable the new functionality of that iteration. The short iterations are a response and a sign of acceptance of the fact that in a real-world setting requirements change constantly, and trying to define, specify, design and plan every little detail at the beginning of a project is wasted work. Much of it will most likely undergo changes at some point during the development cycles [18]. It is generally considered better to have simple, working code that is incrementally expanded upon than gigantic, complex systems that do not work until every piece is in place.

The Agile process begins with the requirements. These are in Agile often formulated as user stories, as they are an easier medium (than heavy, technical documentation for example) to communicate through with the customers, who are critically important in the development process and should be involved at every step [24]. The customers are (usually) the ones with the domain knowledge and should be used throughout the evolving requirements to steer and course-correct the project after each iteration [25]. In the planning phase, the requirements are analyzed and prioritized, as the short iterations (though varied in length depending on the framework [22]) allow only for the top priority tasks to be worked on. However, if priorities change, the planning phase occurs so often (due to the short cycles) that the new circumstances can rapidly be adjusted. The design, implementation and testing phases are the phases where the bulk of the work is done. In these phases, the short cycles force the developers to keep the code simple and focused, but also at a high quality, because bad decisions and sloppy coding lead to an unstable foundation to build upon. Taking shortcuts usually also makes the code more difficult to refactor (structurally changing code without changing the functionality [26]), which will require a large amount of resources when the inevitable time for restructuring and adjusting the code eventually comes [20]. At the end of these phases, a working deliverable is meant to be produced, which can be deployed to the customers to start receiving feedback as fast as possible [27]. The last phase in an iteration is the evaluation of what was done, but also how well it was done. Now is the opportunity to reflect upon the processes and see if something can be adjusted or improved for the upcoming iterations [20]. Then the phases repeat, with the goal of incrementally adding new pieces to the software as well as learning something new from each iteration that makes the development that much better, easier or more effective.

Apart from the phases mentioned above, the key aspect to Agile development remains the people and how they work together [28]. Communication is central in Agile, and not just exchanging information, but actually meeting regularly (many frameworks advocate daily meetings) and talking face to face [20]. This also includes frequently meeting with the customers, who are (or should be) more or less a part of the team [24]. Transparency and everyone being up to date regarding the current situation results in more informed and productive decisions. Management also takes on new forms in Agile. Instead of there only being one decision maker or boss, the responsibility is shared among the team in such a way that the developers make the technical decisions, the customers make the business decisions and the managers make the people decisions [24]. Spreading the responsibility among so many people might seem like a counter-intuitive idea, but the division of responsibility is such that the customers decide what is to be achieved, the developers work out how it can be achieved and the mangers help the developers achieve it by providing the support and resources they need, which makes sense.

2.2 Agile in Aucor

Aucor has been in the web development business for over 10 years and from the beginning has had the goal of being the best WordPress agency in Finland. That goal has been pursued by attempting to be at the forefront of development and product quality. Such levels of quality and upholding them is not easy, though. What a company does (products) as well as how it does it (processes) has to be constantly worked and improved upon. There has always been a company culture and will to improve at Aucor, not just settling for or stagnating on processes, systems or technologies. Inspiration has been taken from Agile methodologies in an attempt to tackle problems that have been recognized (both by the company itself and generally in IT) to slow down progress or create unnecessary work. Adopting these kinds of methodologies will not and did not magically solve all the company's problems, something that is easily being led to believe by hardcore practitioners, but they did alleviate some sore points. At the very least the methodologies introduced the way of thinking that helps to ensure that a CI pipeline is going to be properly adopted by the development process and the team.

The Agile tendencies that Aucor has developed during its history have not necessarily been labeled (internally) as Agile, because many of the practices used in Agile are used as standalone best practices in programming in general. There are, however, four main principles that Aucor operates by that can be related to Agile:

- A starter theme and plugins
- Project-specific git repositories
- Sprints
- Self-organizing and self-improving teams

A more in-depth look into WordPress, themes and plugins will be taken later, but the first point addresses how Aucor has standardized the development process of a website by creating an Aucor Starter base theme [29] and plugins (Aucor Core among others) as a starting point for each project. The theme and the plugins are constantly developed, as new features and methods become relevant and are considered good additions to the components. This style of working reduces the time wasted on inventing the wheel all over again for each project, as functionality often repeats itself on many different websites. It also promotes quality, as developers are familiar with the structure and code standards of the projects and can focus their efforts on extending and tailoring the code, rather than first putting in massive amounts of time just trying to understand it.

Git repositories (or any version control) might seem obvious to many, but they are still not universally used in IT companies. When used correctly, though, common git repositories, despite their simple and even mundane nature, can enhance the daily work of developers in a very agile way. Not only do they contain the most up-to-date version of software for all the members involved to work on, but they also increase the transparency of what is and has been worked on in a project. This is especially true if the repository is used as more than just a code dump, with functionalities like pull-requests and feature branches.

Sprints really encapsulates what the Agile methodology is all about: continuously delivering valuable and working software to customers in short iterations. The customers receive versions of the products early on in development, which means that feedback can also be given early. Based on the feedback, requirements can then be changed as needed at a stage of the development process where it can be taken into consideration without too much hassle. By reducing the time developers have to work on tasks (to a certain limit of course), they are forced to reduce the scope of their work which, in turn, means focusing only on the essentials. The customer projects at Aucor are rarely explicitly sold as sprints (customers without some IT background usually have difficulties understanding the concept of sprints), but internally the projects are still divided into sprints and smaller tasks.

In the end, a motivated and focused team is what ties all the other points together to a working entity. The most important part about Agile is to empower the people doing the work in self-organizing and self-improving teams. Having a manager just telling developers how to work, to improve on processes or to increase profitability will not take them very far. The motivation has to come from the developers, which is best encouraged by letting them have control over their own working environment and ways of working. By giving the developers more responsibility, their willingness to improve on processes that they are in contact with increases, which generally increases the effectiveness of the whole company.

2.3 CI

The actions already taken by Aucor, described in the previous section, are all working towards a better working environment and flow. Although progress has been made, continuous improvement entails that the striving for improvements is never over. Through this thesis, the next step of improvement is the introduction of CI. It is one of the practices that emerged under Agile, more specifically under Extreme Programming [30], which simply advocates developers to frequently integrate the code they are working on into a shared repository [31]. In the past, a common consensus was that software integration becomes difficult and unpredictable when a team of developers work together. Companies therefore postponed integrations as much as possible to not have to deal with them as often [7]. Counter-intuitively, then, CI proposes the exact opposite.

2.3.1 Principles

By integrating frequently and therefore only small batches of code at a time (ideally only a few hours of work), a developer's contribution does not diverge from or fragment the common development effort too much [32]. Subsequently, integrations become easier as potential problems arise earlier and they are easier to find, because of the reduced scope. The latest version of the code base is also more frequently available for developers to work off of. Many teams experience that CI diminishes integrations into non-events and allows them to develop cohesive products faster [7].

More importantly, CI does not strictly rely on any specific tools to be used, only the habit and discipline of actually committing code frequently to a shared code base [7]. The practice can therefore be modified to suit one's situational needs and by using any tools one deems necessary [8]. It then, however, becomes equally important to make the people involved in the process understand the significance of the concept of CI as it is implementing the tools, because they will not achieve much alone [33]. If a CI pipeline is implemented, but the developers do not frequently commit code or write tests for it, the pipeline will not be of any benefit. Combined with thorough understanding of the practice, some basic components have been found effective by the development community to obtain the absolute best results with CI [7]:

- A common code repository with version control
- An automated build process
- Self-testing build
- Integration machine or server

Firstly, a common code repository with version control ensures that developers always work on the latest version of the main source code and that they do not have to search for code in multiple places or fetch each part from a designated developer. In addition, the version control keeps a log of all the changes done to the code base and enables backtracking if something goes wrong, which CI itself does not prevent, but makes it easier to recover from. Software generally requires a large number of files and to keep track of them manually would require a considerable amount of time, but due to version control that is completely unnecessary.

Secondly, the build should be automated. Transforming source code into running software can be very complex and require plenty of compiling, transpiling, linking and more. Doing it all by hand with strange commands is potentially very error prone and, much like manual file tracking would be, a huge waste of everyone's time. Ideally you want to include as much of the build process as possible in the automation, so that basically anyone with a "fresh" machine could just import the code base locally and start up a build script [7].

Thirdly, integration is technically possible without any tests, but an integration could hardly be called successful if the software did not work correctly afterwards. Tests are therefore an integral part of CI and it is highly recommended that testing is a part of the build process. This requires the tests to be in automatic test suites and self-checking, ideally to the point of failing the build in case of errors [7]. The tests should be written alongside the code by the same developer and code that does not have tests associated with it should not be committed to the shared repository, as it cannot be verified to work with already existing code.

Lastly, it is also highly recommended that a testing environment other than that of the developers' is used [7]. It should be a clone, or as close as possible, of the production environment to prevent any errors due to environmental differences, which may be present between developers' machines. The build itself should be the same process as locally, but it can be handled in one of two ways: manually or with a CI server [7]. The manual process requires the developers to start the automated build, but through a CI server the initialization step can be skipped. CI servers can be hooked to monitor changes in a repository and upon detecting changes the servers automatically attempt to build the build.

2.3.2 CI pipeline

The aforementioned components can be used together in a CI pipeline, which in the context of the Agile software development life cycle would be included in the implementation and testing phases. The process starts with a developer fetching the most up-to-date code from a common code repository. The changes or new features are then added to the code, along with tests for the additions. Next, the automated build is built locally, which transforms the code into running software and runs the automated tests.

If everything goes well and the tests do not catch any errors, the developer pulls the (potentially) new changes from other contributors from the repository and builds again. If problems occur, they are solved by the developer who encounters them. These last two steps are repeated until no new changes have been committed to the repository and the integration of them (the build) is successful [7]. At that point, the developer can push the changes to the repository and update the code base. If an external testing environment is used, the developer should make sure that the build is successfully built on the external testing environment as well, before declaring the integration to be complete [7].

The biggest benefit of using a CI pipeline is arguably reduced risk [7]. The pipeline does not make software error-proof, but it makes potential problems significantly easier to solve, or at the very least, find. The faster feedback loop that the frequent commits enable is also an important factor in this, as developers do not only receive faster feedback about what works technically, but also about what works for the customer in terms of the direction of the software. Changing and evolving requirements are a struggle that developers deal with constantly, but one that CI (and Agile in general) tries to harness and actually use as a competitive advantage [20]. Another competitive advantage is the increased product quality that can be obtained through a (correctly used) CI pipeline. The tests themselves contribute a great deal, but the enforcement of writing tests should also shape the code to be testable, meaning the design of the code supports testing [34]. That testability, in turn, enables refactoring, which is usually a sign of high-quality code and long-term thinking, as refactoring is often required when the software is developed further and new features are added. The use of automation to build and test the software also adds quality, as it removes unnecessary human errors from the usually complex and time-consuming processes.

2.4 Testing

Testing is a critical part of CI, but also of software development in general. It is one of the primary ways to assess adequate functional and non-functional behavior of software [35]. The ever-increasing complexity of software and the multitude of environments they are expected to work in makes testing a key asset in assuring software quality [35]. Other ways of accomplishing this include documentation, models from the point of requirements and design, and static methods, such as inspection, review and analysis of code [35]. The goal of all these methods is to reveal potentially faulty behavior and gain confidence in the software once the problems have been dealt with. This is in testing done by dynamically executing the software on a subset of inputs and then analyzing the outputs [36]. Testing has become the standard method of quality assurance (although undoubtedly most effective when combined with the other methods), as the static methods are limited by the fact that they do not actually execute the software [36]. This makes the static methods error-prone, time-consuming and forced to be performed manually [35]. On paper, testing is an activity no developer or development team should ignore, but in reality it can be quite the task to undertake. If approached wrong, testing is often done improperly or completely skipped due to time and budget restrictions [35].

Software testing is a heavily researched area as it faces numerous challenges, smallest of which is not the fact that the growing complexity and heterogeneity of software that makes testing increasingly important is also what makes testing strategies and tools difficult to develop [35]. One of the major reasons for this is the reality condensed in the phrase: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [11]. This comes from the practical limitations of time and budget to exhaustively test the virtually infinite number of inputs and state combinations, which would be required to actually prove the absence of bugs [35]. Thus, only a finite subset of the inputs can be used, which brings forth the difficulty of developing a general, overarching testing strategy: what inputs should be included in the subset? While the reality of never being able to fully test one's software can be disheartening, any testing is better than none. It is also generally agreed upon that testing that focuses on particular properties, such as timing or reliability, is likely to be more effective than just testing the software for general behavior [36]. As such, testing methods and strategies have been developed to help with the task of testing software in different contexts and milieus, but they are far from being applicable in every situation and still require plenty of planning, adapting and customization for each use case [35].

2.4.1 Testing methods

Testing is generally divided into two categories: black-box testing and white-box testing, with the color indicating what the basis for the tests are. These methods (and various combinations of both, called gray-box testing) have been developed in an attempt to make the selection of the finite number of inputs more likely to detect faults

[36]. The methods can be further defined to be tested on different levels and through different objectives, to test specific properties of software [35].

Black-box testing uses system specifications and different input partitioning schemes to run the system against said inputs and the corresponding outputs are then compared to the expected behavior [35]. Schemes that are used include equivalent partitioning, boundary-value analysis and cause-effect graphing, all of which attempt to perform extrapolation on sets of data to effectively reduce the input pool, by identifying similar and redundant data as well as risk areas [36]. The benefit of this kind of testing is that it requires no knowledge of how a system works, only what it is supposed to output. However, the testing is heavily dependent on specification correctness and unambiguousness as well as a large set of inputs to instill sufficient confidence in the software [35].

White-box testing also runs the system against inputs, compares outputs and extrapolates results, but bases the inputs on analysis of the structure and logic of the actual software [36]. This has clear advantages compared to black-box testing, as the one constructing the tests can examine how the system uses the inputs and form groups of inputs that have the same effect, which reduces the number of inputs that need testing. This effectively turns the problem from testing all inputs to testing all the paths a program can take (statements, branches and loops) [36]. The problem with that is obviously that the one making the tests must have access to and sufficient knowledge about the software to write effective tests. The change from input to path coverage does not necessarily eliminate the possibility of an infinite number of test cases either, as iterations in loops, for example, are treated as distinct paths [36]. Combined with a plethora of conditional statements, the number of possible paths can quickly grow uncontrollably.

2.4.2 Testing levels and objectives

As programs and systems have different levels and components, so too can the tests be executed on different levels and on the whole system or just parts of it [35]. The levels to consider are unit, integration, system and acceptance [36]. The smallest independently functioning components must first be tested to work as standalone pieces of software, which make up the unit tests [36]. The smaller pieces are then tested to work together in integration tests, which check that the communication between the components and the collective behavior still complies with the expected behavior [36].

After that the whole system is tested in its intended production environment to detect problems that cannot as such be attributed to be caused by the software independently, but causes failures when attached to specific hardware [35]. Lastly, the software should undergo acceptance testing, which determines if the software is acceptable for the customers: how and to what degree have the specifications been realized in the software. Acceptance testing usually involves the customers to a varying degree through alpha and beta testing [35].

Various objectives, to test different aspects of the software, add a last layer to consider when writing tests, but they also help to focus the testing when facing the harsh reality of prioritizing tests due to a lack of resources. Objectives include functional, non-functional and regression testing [35]:

- Functional testing ensures that the software does what it is supposed to do according to specifications.
- Non-functional testing checks how well the software does what it is supposed to do: how fast, reliable, user-friendly and secure (among others) the software is.
- Regression testing runs selected tests after the introduction of a change in the code, to confirm that the change does not negatively affect how the software has worked previously.

Although the choice of levels and objectives is up to the tester, generally the objectives line up so that the unit and integration tests check for functional requirements and system tests check for non-functional requirements [35]. Acceptance testing can be argued to include both functional and non-functional requirements, as the customers will require both aspects to be up to their standards. Regression testing is executed in whatever scope a change affects, but defining the scope is generally a difficult task [36].

2.4.3 Testing in general

It is easy to assume that testing is done as the last step of software development, as the tests need the code to run on, but this is a misconception. Although only the difficulty of the inputs have been highlighted so far, the reality of writing tests requires a fair amount of effort to be put towards the design of the tests as well. An oracle must also be chosen, which is the agent that analyzes and decides the outcome of the tests [35]. Problems in software can be described with the three terms fault, error and failure [36]:

- A fault is a defect in a program that causes an incorrect state in the program when reached.
- An error is the mistake made by the programmer that causes faults. Errors can be logical or syntactical.
- A failure is the manifestation of the incorrect state caused by a fault and it is the part of the problem we as users see.

Testing only reveals failures, but to find the root of the problems the faults must be identified through analysis [35]. This is why the testing process should start all the way back in the requirement analysis, then systematically proceed and continuously be refined through the design and coding phases, all the way to the beginning of the test case execution [35]. Well-designed tests provide the data needed for any meaningful analysis that the oracle does. The oracle is usually the developer or tester running the tests, but in some cases, if requirements can be made using highly specific formal languages and test cases can be generated automatically from them, the oracle can be another program [35].

A final practical point to remember about testing is that it is very easy to rely too much on tests and push the responsibility of good quality onto them. Although they are a vital part of development, tests cannot save badly specified, designed and implemented software, no matter how well written the tests are [36].

2.5 Web applications

Before dissecting the domain-specific subject of WordPress, which is an application for creating web based content, a clarification of the notion of web applications should be made. Web applications are complex systems based on a client-server resource exchange paradigm, where users (clients) request resources from a remote host (server) through a web browser to be accessed, viewed and interacted with [37]. The fundamental technologies that power web applications are the HTTP (HyperText Transfer Protocol) protocol [38], the HTML (HyperText Markup Language) markup language [39], the CSS (Cascading Style Sheets) markup rendering rules [40] and the JS (JavaScript) scripting language [41].

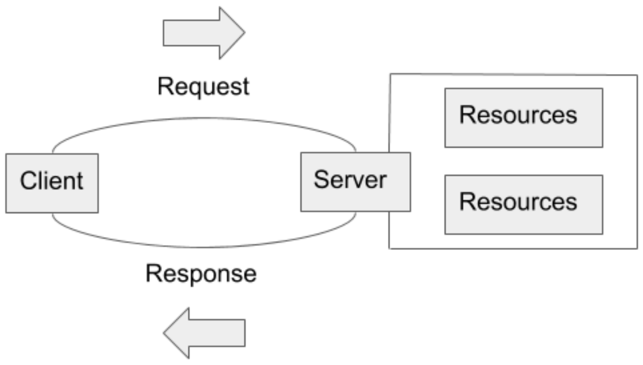
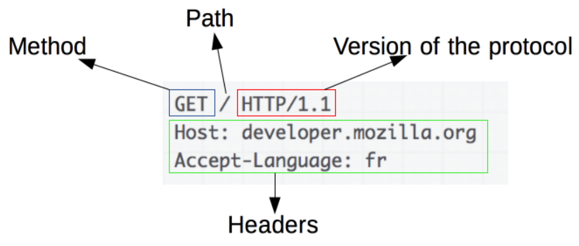


Figure 2.2: Message exchange in HTTP

2.5.1 HTTP

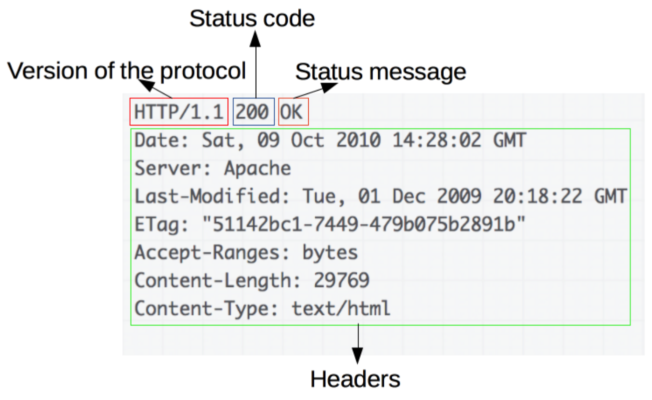
The HTTP protocol is an application-layer protocol created in the 1990's to define a set of rules on how clients and servers may interact with each other, to effectively create what we today know as the Internet [37]. The protocol is sent over a TCP (Transmittion Control Protocol) network and works by clients sending requests to servers, which process the requests and send back responses, ideally with the requested resources [42]. This exchange of messages is visualized in figure 2.2. The figure is simplified in the sense that in reality the two ends rarely speak to each other directly, although the result of the message exchange is the same as if they were. There are numerous entities (modems, routers, other servers) called proxies, hidden away in the network and transport layers in between them, which perform various routing, caching, authenticating and logging operations [42].

The communication between the two parties always begins on the behalf of the client establishing a TCP connection, after which a message is sent in the form of a formatted HTTP request, for which a server is constantly listening [42]. The request is pictured in figure 2.3 and consists of a method, a path to the requested resource (on the server), the version of the HTTP protocol used as well as optional headers and request bodies [42]. In figure 2.3 the method is GET, the path is the root directory "/", the protocol version is 1.1 and two optional headers are included to convey more information about the request. Out of a possible 9 methods [43], GET and POST are the most commonly used. GET is used to simply fetch resources and POST is used to include data in the request body for the server to process [42].



```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

Figure 2.3: HTTP Request example [1]



```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

Figure 2.4: HTTP Response example [2]

The message sent back by the server after a request, an HTTP response pictured in figure 2.4, is very similar to the HTTP request. It, too, contains an HTTP version, a set of headers and an optional body (if a resource is to be returned to the client). Instead of a method and a path, however, it contains a status code and a status message that indicates the success or failure of a request. In figure 2.4 the HTTP version is 1.1, the status code is 200, the status message is 'OK' and the headers contain additional data about the response.

2.5.2 Requested resources

The most typical request made to a server is to gain access to an HTML document [37]. This can be a static web page or, if additional resources such as client side scrips are included, a dynamic and interactive web application. An HTML document is a content page (text, links, images, videos) that is complemented by special HTML tags, which define the structure of how a web page should be rendered in the browser [39]. The tags can divide the content into separate sections, create lists or just increase the semantics of a page by declaring content to be an element of a certain type, such as a header, a footer or a navigation element.

The additional resources that are used to enrich the user experience and to increase the dynamic capabilities of a web page are CSS and JS. The former defines a set of stylistic rules that tells the browser how the HTML document should look (aesthetically more than structurally) and the latter creates interactive events and procedures based on the users' actions on a page [37]. They can both be included in their own tags in the HTML markup or linked to the document from external files, in which case the files are also requested by the client in separate request-response message pairs after the HTML document.

2.6 WordPress and plugin development

The main technology that Aucor uses to create its websites and services is WordPress, which is the most widely used CMS on the market and powers up to 35% of all websites on the Internet [44]. It began as an open-source blogging platform when it was forked from the now deprecated b2/cafelog blogging tool [45] in 2003 [46]. The goal was to create a platform that was aesthetic, user-friendly and web standard compliant, but most of all hassle-free to publish content on [46]. While it is still used today by bloggers all over the world, WordPress has grown into an ecosystem that lends itself to support bigger and more complex infrastructures, such as eCommerce and enterprise websites [47]. For Aucor, WordPress provides an extendable platform that is equipped with an out-of-the-box content management GUI (Graphical User Interface) and functionality, but with the freedom to individually tailor the websites' look and feel according to each customer's needs and wishes.

2.6.1 WordPress ecosystem

For simple websites and blogs, WordPress comes as a complete package in the form of the WordPress core. The idea is that you do not have to have any experience with coding or building a website to create a fully functioning and (relatively) good-looking site. The core contains basic functionality to write, publish, manage and view content, and takes care of security through built-in user and authentication systems. The core is continuously updated and driven forward by a designated team of developers, designers and an actively contributing community [48], but the true strength of the core (and what makes it interesting for developers) lies in its customizability through themes and plugins.

The WordPress ecosystem, and therefore a website that is built upon it, can be seen as a construction of five main components:

- The core
- Themes
- Plugins
- Configurations
- The content

The core is the centrally developed product that provides the essential infrastructure and default behaviour of a website. The most notable functionality is the dashboard tool, a GUI which allows logged in users to easily create and manage the content and configurations of a website [49]. The core is written in PHP (PHP Hypertext Preprocessor) [50] and operates over a MySQL [51] database.

Themes and plugins are what makes WordPress so flexible and customizable as a platform. By being open-source and licensed under the GPLv2 (or later) licence, WordPress allows and even encourages users and developers to take WordPress and make it their own by modifying it as they please [52]. It was realized early on, however, that modifying the code in the core would not be sustainable if a developer wanted to keep receiving the support and updates from the WordPress team, as the modified code would be overwritten when the core's files were updated [46]. The solution was the plugin system. Hooks were introduced in the code base, which allowed developers to run code or modify data at specific points or events during the execution. The result was modified behaviour without touching the code in the core [53]. This allowed the WordPress team to solely focus on developing the core in a direction that made sense for the majority of users, instead of trying to please every developer wanting both a niche functionality and an updateable core. A 80/20 rule started to emerge in the core team: *"Is this useful to 80% of our users? If not, try it in a plugin."* [46]. Plugin development has since then become a massive community and business [54] with tens of thousands of free [55] and premium plugins, which cover a wide range of different functionalities.

While plugins modify the core's functionality, themes provide the front-end design and layout of the websites in WordPress [56]. Themes usually consist of a collection of files that includes [56]:

- Templates for the layout with HTML
- Stylesheets for the design with CSS
- Code files for data and content fetching with PHP
- Image files
- Scripts for adding dynamic front-end behaviour with JS

Much like with plugins, theme development has become a booming business with thousands of both free [57] and premium themes. As a part of the working out-of-the-box mentality, WordPress comes with default themes, which set the bar of creating a simple website extremely low. Apart from its open-source nature, the popularity of WordPress can definitely be credited to the working and flexible symbiosis between the core, the themes and the plugins, which is appreciated by both the users using the platform as well as the developers extending it.

The configurations and the content are the last components that define a WordPress website. The configurations consist of settings for the core as well as the themes and plugins, which give some power to the users to further define how their website looks and how people interact with it [49]. The content does not explicitly affect how a website works or behaves, but it defines to a large extent how the website looks. All of this is mostly stored in the database, which the core (and the themes and plugins through core functions) fetches and presents to the world.

2.6.2 Plugin development

A plugin is a package of code written in PHP to extend or modify WordPress functionality, without overwriting the code in the core [58]. Plugins may vary greatly in size and functionality. The simplest plugin can be a single PHP file printing out a message on the dashboard, while others can be full-featured systems with their own HTML structures and templates, CSS for styles and JS for dynamic functionality [58]. Technically, this makes it possible for plugins to act the role of a theme (and vice versa, as themes can technically modify functionality), but the convention is that themes decide the overall look of a website and plugins only control the design and layout of their own content.

Plugins are the ultimate opportunity to create and build in WordPress. While there are detailed plugin guidelines, [59] the only requirement to create a plugin, from a technical stand point, is a PHP file with a specially formatted header comment [60] containing metadata about the plugin [61]. The guidelines do, however, matter if the plugin is to be released for others to use through the official plugin directory [55], along with a review process before being made public [62]. Since a plugin could potentially be used by millions of websites and by both technical and less technical users, the WordPress team has made security, safety and privacy top priorities. They try to ensure that the plugins they host stay consistent with the goals of the core product: simplicity, transparency, standard and convention compliancy, and open-sourceness [59]. But guidelines and reviews can only help so much and all plugins (mainly plugins that are not free) are not even hosted in the official plugin directory where the reviewing is done. Plugins do not have to be "official" to be used in WordPress, but even if they are, there is no real guarantee of quality or safety for the plugins [63]. Even if the WordPress core itself focuses heavily on security [64], the majority of the responsibility still lies with the developers to ensure the correctness of a plugin and to keep its users content.

Hooks are the feature that plugins and even the core itself rely on to perform modifications on data or to execute pieces of code. There are two types of hooks: actions and filters [61]. The technical difference between the two is that a filter requires a return value, but an action does not. The convention is, however, that actions are used to add data or change behaviour of code, while filters are only used to modify data as it is processed and displayed on the website. They both work with callback functions, which are functions that are attached to a hook of choice and then "called" back and executed by the hook when it itself is executed [53]. When attaching a callback to a hook, additional priority and argument parameters can be given [65]. The priority defines in which order callbacks are executed when multiple are attached to the same hook, while the argument parameter defines how many arguments a callback function accepts (not the value of the arguments). Hooks are not just exclusive to the WordPress core, as developers can and are usually even expected to make custom hooks in their plugins, so that their code can in turn be expanded or modified by other developers. With custom hooks one can even remove other hooks [66].

3 Adopting CI at Aucor - an analysis

With the bigger background concepts covered, the attention can now be shifted to the practical aspects of CI. Like briefly mentioned in the introduction, although the previous chapter's section on CI presented guidelines to adopting such a practice, there is not just one way to implement CI. This means implementing it does not always manifest the perceived benefits equally [6] or without problems [67]. Studies on correlations and exact consequences are still lacking [8], but researchers speculate that it might be a case of different expectations for the people involved [6], project contexts and a variety of attributes restricting implementation [8], or that a company might not organizationally be suited to adopt such a practice [9, 67]. This does not mean that there is no evidence for the expected benefits, in fact there is a good amount of success stories [6, 67], but that there is more to consider for maximum gain than just choosing a favorite CI tool.

Two aspects need to be taken into consideration when implementing a CI pipeline: how the organization will adapt to the introduction of the new practice and what attributes and activities the pipeline itself will take on. A study by Eck et al. [9] comprised a list of implications they deemed necessary for a company to successfully assimilate CI into their operation, which is supported by a study by Shanin et al. [67], who gathered reported problems in adopting continuous practices (integration (CI), delivery (CDE) and deployment (CD)) as well as practices to help combat them. Ståhl et al. [8] found a set of variation points in their study on CI implementations that are ambiguous and up to debate regarding actually setting up a pipeline. Not all of the implications or variation points apply to every company to the same degree, as contextual factors such as company and project size, type, structure and goals vary [8]. This is the case for Aucor, who likely will not encounter all the problems mentioned in the studies, largely due to the relatively small size of their development unit and number of teams operating in parallel.

3.1 CI assimilation

As an Agile practice, CI is an agent of change for any company implementing it, inducing different changes, as it is being assimilated by the company [9]. Again, the sole act of integrating frequently, which is technically what CI is about, does not require more than a change of habit for a developer, but its translation into a real working environment does need more effort and support from an organizational point of view. For a company such as Aucor, which uses Agile methods already, the changes might not be very significant, but companies that are not already familiar with this kind of development models might not have the infrastructure or organizational processes necessary to support CI [67]. Eck et al. divide the assimilation of CI into three stages, according to an adoption theory of technology-induced innovation [9]:

- Acceptance - accepting the innovation and using it daily.
- Routinization - the innovation is considered normal and other processes are adjusted according to it.
- Infusion - the innovation is used extensively and the practitioners are mastering it.

Each stage represents a level of accustomization a company has developed towards an innovation, and each stage comes with implications to company activities. As a company progresses through the stages, the implications become more ambiguous and more trade-off weighing must be done [9]. It is noteworthy that the study [9] found that none of the companies surveyed had reached the infusion phase, and that the scope for this thesis is bringing the company to the acceptance stage. Thus, only groundwork can be laid for the latter stages, as progress to later stages according to the authors of the study [9] only happens through imminent challenges, which is out of control for the thesis. The study [9] also observed that some companies performed according to more difficult implications long before being proficient in others, which suggests that the stage structure presented might not be as sequential and exclusive as it might seem at first. The implications in the different stages are not necessarily in any order of difficulty either.

Acceptance stage

The acceptance stage comes with four implications, all of which lay the groundwork for CI not to be rejected immediately:

- Devising an assimilation path
- Overcoming the initial learning phase
- Dealing with test failures immediately
- Introducing CI for complex systems

Devising an assimilation path, meaning how the introduction and further development of the practice should be handled, is for Aucor suggested to be an *extended nucleus* approach. A core CI system, with automated tests and integration, is implemented for one product, whose ambition and reach are then extended continuously until it is used to develop software used in production. The initiative for it is then a mix between top-down and bottom-up, as the initial system is set in place by a more top-down approach (need recognized and initiated by senior developers), but the further development comes more from a bottom-up approach (developers using the system improve it).

It is very likely that a dip in productivity will occur before the expected increase in it, signifying that the developers as individuals and as a team need time to become familiar with the new ideas, processes and tools to in an effective manner overcome that initial learning phase. A simple way to speed up this process is educating the developers on the ideas behind CI and present the tools chosen for them as a final step of the implementation in this thesis. This might prove to be an especially important step for Aucor, as the developers themselves do not implement the system, but are expected to use it once it is in place.

Dealing with test failures immediately enforces the habit of aiming for quality from the start and not letting the errors accumulate. This can be enforced to different degrees in the CI pipeline by not letting developers merge their code in case of errors and standard defiance.

The final implication at this stage is introducing CI for complex systems, which addresses the challenge of complexity that CI faces regarding target environments, different technologies used and the test suites. While there are variations, for the most part Aucor uses standardized tools and environments for development, many of which already have the support for testing and different CI services and tools. This should, thus, reduce the complications when adopting the practices.

Routinization stage

The routinization stage marks the point where developers have become comfortable using the practice and are homing their skill using it. Though only speculations can be made from this stage onward, it could definitely prove beneficial to prepare for what is to come. The stage comes with five implications:

- Institutionalizing CI
- Clarifying the division of labor
- CI and distributed development
- Mastering test-driven development
- Providing CI at project start

Institutionalizing CI continues the assimilation path and initial learning phase by taking a stance on how the efforts towards improving and furthering CI should be taken care of once it becomes a part of daily life. From the Agile development methods and a willingness to improve present at Aucor, there is no doubt that a *communities of practice* alternative is the best way to go, meaning that self-emergent leaders take an interest in the subject matter, drive the progress forward and guide others. For this to work, however, the company must grant the time needed for these individuals to delve into improving the craft.

Clarifying the division of labor concerns the work, as it moves through the CI pipeline. Though not by any means trivial, the projects and products in Aucor are relatively small and developed by a small team, so the need to perform any hand-overs or similar, where responsibility for a code change is delegated or even split, is almost non-existent. Therefore, a developer performing the change should see it through that the change passes the CI pipeline as expected. The exception to this is that the final integration to the main branch might be made through a pulling action rather than by default, leaving the final say to the product owner when integrating features. This only encompasses the literal last step of the pipeline, however, and even then should be automatic if, and when, the product owner accepts the integration.

CI and distributed development illuminates the fact that with decentralized teams working on a project, communication and the frequency of it correlates with the frequency and transparency of the CI pipeline. Though frequent synchronization of a common code base can increase the need for communication in case of errors, a well-implemented pipeline, which is accessible, up to date and transparent, conveys much of the needed information implicitly.

Mastering test-driven development, which is a common method in Agile development [35], is not the kind of development that is currently practiced at Aucor. Given some time to become familiar with CI and more comfortable writing tests, however, writing code with the tests already in mind will no doubt become a valuable addition to improving the pipeline.

The last implication for the second stage is providing CI at project start. For CI to be as effective as possible, it is important for it to be present at the start of a project or it will most likely cause more hassle than gain (implementing the pipeline and performing any necessary architectural changes to the code base). For this thesis, the introduction of CI to an already existing product is acceptable for three reasons: the initial hassle will not be dealt with by the developers, the goal is to introduce the company to the acceptance stage and not the routinization stage (for which this implication is meant), and the product is continuously developed, which makes the CI pipeline retain its value despite its relatively late addition.

Infusion stage

The last stage comes with five final implications. The study [9] would declare a company working according to these implications truly masters CI:

- CI assimilation metrics
- Devising a branching strategy
- Decreasing test result latency
- Fostering customer involvement in testing
- Extending CI beyond source code

To be able to prove quantitatively that a pipeline works, CI assimilation metrics must be considered. The most obvious metric to be measured would be the number of (reduced) integration errors during development, but the study [9] also suggests an input metric in the form of institutionalization of reflective activities, such as continuous improvement. This shows a level of sophistication and a sign of progression of CI use.

Devising a branching strategy is one of the implications that do not necessarily follow the proposed sequentiality of the stages, as Aucor, for example, already uses a feature-driven branching strategy when developing their products. Nonetheless, it is a way of not jeopardizing the stability of the common code base, which is the overarching theme of CI.

Test result latency, which ideally is near non-existent, is likely to increase as the size and complexity of both the software and the test suites grow. If the processes are continuously improved, the increase can be mitigated through automation and infrastructure upgrades, as well as test optimization, decomposition and hierarchies.

Fostering customer involvement in testing (more specifically acceptance testing) makes sense, as they are ultimately the ones the software is for, but the study [9] goes as far as suggesting that the customers should formulate the tests on their own for later automated evaluation. While ideally this would be the case, and the study [9] does recognize that the customers need to be convinced that this is an effective approach, it would also require the customers to be somewhat adept or knowledgeable in software development and testing for the tests to be of any use. In Aucor's case, for example, this is seldom the case.

The last implication that a company has progressed to later stages of CI assimilation is extending CI beyond source code. This is definitely more relevant when companies build entire systems, where configurations and firmware are more tightly linked to the software in question, but far less relevant for companies such as Aucor, which also relies on other service providers (like hosting) and has limited control over those domains.

The three stages and the implications listed create a conceptual framework for CI assimilation [9]. It shows the general adjustments (with increasing ambiguity) a company must make to progress with its use of CI. When reviewing the implications, it seems that Aucor has the ability and potential to progress through the stages (certainly the acceptance stage) without too much difficulty.

3.2 Recognized problems in CI implementations

The implications from the previous section highlight operational processes and tasks when CI is implemented correctly, albeit with different levels of sophistication. The study by Shanin et al. [67] complements those points by further focusing on the problems companies have encountered when the implementation has been lacking in some respect. Reviewing them and identifying potential pitfalls pre-emptively will further strengthen the foundation on which the CI pipeline will be built for Aucor. The study [67] contains general problems as well as problems specific to all the three continuous practices (CI, CDE, CD), but only the ones concerning CI will be discussed.

The first problem was poor team awareness and communication due to lack of transparency and challenges in coordination. This reflects the implication of *CI and distributed development*, but applies to teams working in the same vicinities as well. The key is to implement the pipeline with maximum transparency, so that all those involved are constantly aware of, or at the very least provide an easy and accessible way of finding out, the current state of the software and the pipeline. A system signaling the status of the latest build is highly advisable [7], but having access to information such as version control (history) and branches is crucial in conveying the overall situation and awareness. Working towards general knowledge sharing among team members also enables better adoption of CI [67].

The second problem was a lack of investment in skills, suitable tools and technologies. It was reported that the new practices that CI practically is coupled with, such as tests and automation of steps, require new skills and generally new tools to use efficiently. This is directly comparable to the *Overcoming an initial learning phase* and *Introducing CI into complex systems* implications and the failure to foresee them or react to them. A company that is not infrastructurally ready to start working with frequent integration or automation will most likely struggle, as do the companies that fail to provide enough planning, training and documentation about the practices as they are being introduced. It was also mentioned that insufficient investment in clarifying work structure, roles and policies could contribute to more pressure and workload for team members. Much like with the *Clarifying division of labor* implication, this is certainly a much bigger problem in large companies, with multiple teams involved and cross-functional dependencies across the CI pipeline, but the common rules of the company should nonetheless be made clear from the start, even if they do not change anything in practice.

The third problem was a resistance to change, which stemmed from scepticism towards CI, but also a general reluctance to give up old habits. Reminiscent of the *Devising an assimilation path* and *Institutionalizing CI* implications, this is the situation where companies with open mindsets and familiarity with the continuous improvement concepts of Agile have an advantage in seeing through the struggles and welcoming change. Reported alleviations for resistance were the promotion of a positive team mindset, which sparked an excitement towards CI and, like with the lack of investment, made the developers comfortable with CI through training and coaching.

The fourth problem was difficulties in changing established organizational processes and working in distributed teams. The established processes, like with individual resistance to change, are due to the company culture and working models, which in agile companies are less of an issue than in more traditional ones. The distributed teams reportedly had problems with consistent perception among the teams, which again becomes more problematic with size, but can be fought with transparency and more active communication.

The last (CI related) problems were testing, which reportedly was one of the major challenges to overcome when adopting CI, and merging conflicts. Many problems came from low quality tests, but also shortcomings in the surrounding infrastructure held progress back, such as the inability to automate all types of tests and heavy dependencies. The reported solutions to these align nicely with the implications of *Dealing with test failures right away*, *Mastering test-driven development*, *Devising a branching strategy* and *Decreasing test result latency*. By upgrading testing from an afterthought to a primary objective, test quality and coverage can be expected to increase, and proper modularization and decomposition will help with automation and feedback speed, as size and the number of dependencies in the code shrink. The branching strategy and dealing with failures immediately promote good habits, which lead to more stable integrations and overall better products.

Examining the problems, it seems to be a running theme that they are to a high degree caused by treating CI as an isolated tool or event, rather than a practice affecting the whole operation. It is, therefore, not surprising that the sentiments were mirrored in the implications presented by Eck et al. Given that the implications were deemed to be possible at Aucor, the problems listed above should not cause any greater difficulties either.

3.3 CI pipeline variation points

The more practical and technical details laid out in the study by Ståhl et al. [8] are arguably less make-or-break factors for the adoption of CI than the two previous sections, but they were observed to differ in the examined implementations and affected the gained and perceived benefits of CI. The main contribution of the study [8] is a descriptive model of CI that covers evident variation points, enabling better understanding of different implementations and making it easier to compare them. It can also be used without comparisons to gain insight into one's own pipeline. A model will be created for Aucor after the implementation, as it could provide useful information for future improvements. For now, the variation points can be used, much like with the implications and problems described earlier, to highlight points to consider when introducing the CI pipeline and build the foundation even stronger. The study [8] found 16 points or statement clusters, as they called them, that were mentioned in more than one unique source and displayed enough disparity between them to be considered variations.

Build duration

Build duration is an important factor for keeping the pipeline running smoothly and providing feedback to the developers fast enough, so that they still have the changes fresh in memory. With the technologies Aucor uses, there are not as many steps in the build that take time as in software that has to be compiled, for example. The "compiling" done on a WordPress stack (mainly consisting of PHP, which is a dynamic language) could be seen as the linting, transpiling and compression of files, which are not lengthy processes. The one step that does take time and is, at least in a CI pipeline, included in the build process is the testing. This must be kept in mind when writing the tests and constructing the test suites, which can (when designed poorly) take up a significant amount of time when run. The study [8] highlights a report claiming that the increase in the number of tests is the primary factor that influences build duration.

Build frequency

Build frequency, which is closely related to integration frequency, but separated here because of the optional synchronization to each other, is in part what makes the build duration a critical factor as well. The spirit of CI is to integrate frequently, which means (regardless of whether they happen at the same time) that the build should ideally be built as frequently as the number of integrations. The frequencies vary, but builds are usually built multiple times per day which, of course, heavily depends on the build duration. If too long, the build duration physically limits the frequency at which a build can be built. The goal for Aucor is to be able to build very frequently, with builds only taking minutes.

Build triggering

The trigger for starting a build is in the majority of cases the change in source code, but can also be set to a fixed schedule, which would make sense for lengthier builds. The study [8] also mentions a setup where activities are chained in a sequence and where the completion of an earlier step triggers the next. For Aucor, the change in source code is the most logical trigger. For the sequential triggers, there is not really anything else in the build process outside of a successful file linting, transpiling and compressing sequence that could enable (and in a way trigger) the testing. The problem is that those processes are only done locally and what is pushed to the repository are the already ready-made files.

Definition of failure and success

Depending on what one includes in the build process, there are different definitions and tolerances to failure. Zero-tolerance is always the safest way to go, but if static stylistic analysis of the code is included, for example, the "errors" thrown might not be deemed enough for the build to fail. Tolerance to test failures must be considered very carefully, if at all. For Aucor, test failures will have a zero-tolerance policy and high standards for static analysis of the code, as some of the products are publicly available and will need to be presentable, consistent and always in a working state.

Fault duration

While the general consensus from the CI guidelines to the implications is that a fault should be handled immediately, the study reports [8] variations in this respect. The problems often arose from the fact that even though the reaction to a fault was immediate, a solution was not. Other cases had to do with the tolerance to failure, where faults were deemed to be non-critical and intentionally left to be dealt with at a later time. For Aucor, the fault durations should be kept to a minimum. If a problem appears to be persistent, a rollback to a previous version is likely to be the temporary solution until the problem is solved.

Fault handling

When problems arise, the responsibility to solve them and their priority have according to the study [8] been given to a number of people in different scenarios. While a problem was in some cases assigned to a dedicated team to solve with top priority, sometimes the fault tolerance lowered the priority. At the size of the software and the team at which Aucor is working, the clearest and most intuitive way to handle problems is immediately by the developer who causes the build to break. Not only do they know what changed, the urgency makes it possible for operations to resume normally as quickly as possible.

Integration frequency

As mentioned earlier, integration frequency does not necessarily equal build frequency, but the recommended frequency for integration is also multiple times per day. As the build duration is set to be very low for Aucor, the integration frequency will likely be the same as the build frequency. It will not take up a significant amount of time and it retains the sentiment of not straying too far from the code in the common repository (reducing integration problems) and keeping everyone up to date as much as possible.

Integration on broken builds

According to the study [8], the consensus of adding commits to a broken build is that it can be problematic, but not to a degree that it would always be strictly prohibited. It is difficult to imagine a situation for Aucor, where committing changes to a broken build (that is not a repair) would be of any benefit or so critically important that it cannot wait for the build to be repaired. On the contrary, if systems are not in place to shield a broken build to be added onto, mistakes are bound to happen, which potentially only makes the already broken build worse. Prohibiting commits onto broken builds is therefore in the best interests of the company.

Integration serialization and batching

Serialization and batching can become a topic of discussion if multiple changes are done during a single build, when build times are slow, but the integration frequency is high. This is undoubtedly a more common problem when larger teams, or several teams, work on the same software. Neither of these cases are really relevant for Aucor, however, so an optimistic approach highlighted by the study [8], where developers can freely integrate their work when a build is successful on their own machine, is likely to work well.

Integration target

As the products at Aucor are developed according to a feature branch strategy, the integration targets for changes are the development branches they belong to, which are merged into the stable main branch when the features are deemed finished and all the tests have passed.

Modularization

Modularization, in this context, refers to the fact that when integrations are done on a system that is comprised of smaller components, the components might have their own CI pipelines and cycles instead of a common, massive one. The build process is in that case divided up into modules, where a component is only rebuilt if changes affect it or any of its dependencies. In very large entities, this has the benefit of dividing up the testing as well, making it faster to receive feedback as only relevant tests are run. It also makes scaling more viable as quicker, module-specific unit tests can be run on every build, but the more computationally intensive and slower integration and acceptance tests can be staggered throughout development. For Aucor, modularization is technically true, even though the products they develop are not divided into modules, but they themselves can be seen as modules of WordPress. When building, only the plugin is built and tested. The core's correct building and testing is assumed to be dealt with by the developers of WordPress.

Pre-integration procedure

Pre-integration procedures are actions that are done before integrating any changes. According to the study [8], multiple sources report that they do not require developers to perform any significant pre-integration procedures, i.e. building or testing when committing changes, and that it needs to be acceptable to break the build. Maybe it is easier like that when dealing with huge software and enormous build and testing durations, but the more suitable sentiment for Aucor is the one of making the build as difficult as possible to break. As the expected build and testing times will not be insurmountable, developers should build and test their changes locally before integrating anything to the common code repository. The study [8] stresses that such procedures can have negative side effects and that developers may not follow or remember to follow said practices, but being careless about those kinds of procedures just seems to go against the overall goals of CI.

Scope

The scope of CI refers to the number of activities included in it. The study [8] claims the baseline for CI to be compilation of code and running unit tests. On top of that baseline, additional features or more complex testing can be added. For Aucor, the baseline will be complemented by integration tests and static code analysis.

Status communication

Communication and awareness is critical in CI, especially if something goes wrong. The external integration services, such as Travis CI, offer several options of displaying the status of the current build. In addition to providing a status badge to put in the software's repository, the service can be integrated to notify the developers through Slack [68], a communication application used in Aucor, when a build is complete.

Test separation

Test separation was mentioned in the modularization section and can make receiving feedback much quicker, as only a set of the overall tests is run in any given build. More complete test suites can be separated into scheduled events. Given the benefits of scalability and modularization for the future, the tests written for Aucor will be divided up by features, which technically separates the tests, but they will still be run together in a build, because of the added security and the possibility to do so timewise.

Testing of new functionality

When releasing a new product, the developers must have confidence that the product works. To gain that confidence, testing is required. While it is true that CI is used for regression testing, like some sources in the study [8] describe, it is equally important to write tests for the new functionalities as well. Though in this thesis tests are added retroactively, the groundwork for future features will also be laid in regard to the testing frameworks and automation, so the step of writing tests for new features will be much easier.

Through these 16 variation points, a clearer idea of the CI pipeline has been achieved. For a simple idea, very much room has been left for interpretation, but more in the sense of how to reach a goal, not what the goal is. For the most part, a common consensus was found in the study [8], but with contextual factors often impacting the execution. With the benefit of having a smaller team and project sizes, the CI pipeline for Aucor can be constructed to be effective, follow a high standard and be resilient to breakage.

4 IMPLEMENTATION

With a conceptual foundation built from the analysis of relevant studies, the path to implementing a fully functioning CI pipeline in the context of Aucor's development process was clear. This chapter will present the steps taken and the practical implications of setting up the testing environments locally and remotely, writing the tests and combining them all into a cohesive workflow.

4.1 Local development environment

The process began by reviewing the current development environments and determining if and how local testing could be integrated into it. It seemed logical to start from where the developers would also start their development process. For most of its sites, Aucor uses an external hosting and upkeep provider, Seravo's WP-Palvelu [69], who provides a WordPress base [70] for local site development. The base mirrors the layout and stack used on their production servers (WordPress, Nginx [71], MariaDB [72], PHP7) as closely as possible, and includes an opinionated Vagrant box [73]. Vagrant is a command-line tool for virtual machine environment management and build automation, and is used to isolate dependencies and configurations inside shareable "boxes", which further helps keeping the local environment in line with the one in production. The virtualization tool that Vagrant uses in Seravo's base is Oracle's VM Virtualbox [74]. Because a WordPress installation is basically all that is needed to develop the Aucor Core plugin, and because the developers already have Vagrant and Virtualbox installed for working on other projects, setting up a relevant development environment for Aucor Core is trivial. The developers only need to download a new Seravo base and the Aucor Core plugin from GitHub, and start the whole thing up with a single start-command, which prompts some setup-related 'y' or 'n' questions. With everything installed and downloaded from a previous development session, the "daily" setup routine is, thus, reduced to effectively only one step. This does not in itself help to establish a local testing environment, but it is a good start for the pipeline and workflow as a whole, if the initial setup is as simple as possible.

4.1.1 Adding testing support

Apart from the stack and the Vagrant box, the Seravo base also comes with an array of other useful tools, the most important (for WordPress testing purposes) being the WP-CLI [75]. The WP-CLI provides a command-line interface for the WordPress admin, which is used to interact with and manage the site. One of the commands that are given access to through the tool is *wp scaffold plugin-tests* [76]. It turns out that the WordPress core comes with a built-in command to generate the required files to set up a local testing environment and run tests, including example and configuration files as well as an install script to set up WordPress. Despite the fact that WordPress is all about easy setups, to a large extent community driven, and that a concept of test scaffolding such as this is perfectly in line with those sentiments, it was still a positive surprise to have the initial framework set up as easily as that. The configuration files included default ones for PHPUnit [77] and Travis as the unit testing framework and external CI service of choice respectively, which was more than suitable for this thesis. PHPUnit is part of the well-established xUnit family, which made it the first choice for a testing framework anyway, as was Travis, which seamlessly integrates into GitHub (where the Aucor Core repository resides) and is free for open-source projects (which Aucor Core is). The install script that was included installs a copy of WordPress in a temporary location when run, which strictly would not have been necessary as a copy was already received from the Seravo base. The script does also, however, install useful unit testing tools, provide an option to initialize a fresh database instance not to overwrite anything in your development database [78], as well as providing a way to quite easily have two different core versions running at the same time, which can help when testing backwards compatibility of new features. The overlap of WordPress instances, as they do not interfere with each other, was acceptable and, because of the easy compatibility testing, even welcome. For the fresh database, problems arose with insufficient permissions to create one within the newer versions of the Seravo base. It was discovered, however, that the database table prefixes were different for the two WordPress instances and that they would not cause problems with overwriting data, so the initialization of the database during the install script could just be skipped.

4.1.2 Testing framework

Now, then, the development environment had been established to support a testing framework. The next step was to add the testing software: PHPUnit for dynamic testing and PHP_CodeSniffer [79] for static code analysis. PHPUnit is the framework of choice for the development team of WordPress as well, but the core is actually only compatible with the 7.x release tree [80], although the newest version of the framework as of writing this thesis is 9.x [77]. This does leave out some newer functionality, such as more advanced assert statements, but more importantly the support for version 7 ended in February of 2020 [77]. This obviously poses a potential risk for the future, but as the advancement of the support lies with the core team, for now the outdated version of the software must be used. To prevent compatibility issues for the Aucor Core development, and to be able to handle the updating of the version centrally and controlled for the whole team once the support exists, the dependency manager Composer [81] was introduced to the plugin. A dependency manager keeps track of and manages project-specific libraries, automatically installing and updating them according to a lock file. This helps keeping libraries (which in turn might depend on additional libraries) and their versions separated from each other, if and when projects require different configurations. While the PHP_Codesniffer does not specifically have similar issues with its versions and WordPress, it was also included in Composer as good convention and to prevent possible aforementioned issues with other project configurations.

The last step to complete the local testing environment was to configure the configuration files generated by the scaffolding command. These included:

- A PHP_CodeSniffer configuration file. The tool tokenizes files and warns developers when the code they write violates the coding standards defined in the configuration file [79]. This is used in other Aucor projects, as well, so the standards to enforce were already defined.

- A Travis configuration file. Instructions are given in this file for how Travis should set up the testing environment remotely. The specifics for this will be discussed in a later section.

- A PHPUnit configuration file. This file contains attributes and values on how the tests are run, such as pre-execution actions, stop criteria and output formatting. These were for the most part left unchanged, except for the definition of the test suite. For that, a combination of path and directory directions, as well as file prefixes and suffixes, could be given to tell the framework what files to include in the tests.

- A bootstrap file that was by default defined to be used in the PHPUnit configuration file, for any steps that needed to be taken before tests were executed. It gives access to useful functions by including a function file from the WordPress core, loads the plugin being tested and starts up the WordPress instance. The only change to this file was to replace the default plugin name with the real one.

With the files configured, the development environment was now ready for tests to be written. The set up of the entire environment on a "fresh" computer ultimately looked like this:

- Install Vagrant and Virtualbox.
- Download the Seravo base and the Aucor Core plugin.
- Start up the Vagrant box with a start command.
- When the Vagrant box is up, connect to the virtual server with SSH.
- Navigate to the plugin directory.
- Installs the libraries in the Composer file with an install command.
- Run the WordPress install script. At this point one can define what version of the core is to be used in the tests.
- Assuming tests have been written, run tests with a run command.

Again, if the environment had already been set up, the daily routine would be reduced to only starting up the Vagrant box, connecting to the server, changing to the plugin directory, running the install script and running the tests.

4.2 Aucor Core plugin

Before covering the tests, the Aucor Core plugin should be described on at least a surface level, so that a context for the tests is provided. The plugin consists of classes: a main class and classes that divide the functionality up into features and sub-features, as can be seen in appendix A.1. The main class is hooked into a plugins_loaded action, which, when executed by the WordPress core, initializes the class. The main class, then, in turn includes the files for the features through a require_once statement and initializes them into a feature array. From there the features are free to provide the functionality that is declared in their respective classes. The plugins_loaded action is one of the earlier ones to fire [82], as plugins usually modify the way other components, such as the themes, interact with the WordPress core. Thus, their functionality needs to be loaded in before the other components to prevent any conflicting behaviour.

4.2.1 Features and sub-features

The functionality included in the Aucor Core range from restricting or removing non-relevant default WordPress functionality, to changing HTML markup, to altering default settings of other frequently used third-party plugins. The main features are intermediate initializers of the sub-features, not actually containing any functionality of their own (that affects the site), but acting as collections of sub-features. There are 9 features and 33 sub-features in total, each functionality roughly having its own sub-feature (some do more than one thing) and grouped under a main feature according to the nature of the functionality. This is done to enable the ability to easily switch features and sub-features on and off at will. If a functionality is not wanted, for whatever reason, it can be turned off with a custom filter. Alternatively, a whole feature can be turned off with a similar filter, meaning all the sub-features and their functionality that belong to that feature are turned off. A WordPress plugin does not have to be written in classes, but to achieve this kind of flexibility the modularity, isolation and clarity that a class structure provides is essential. Although the functionality between the features and sub-features is quite varied, the basic structure of them does not have to be. To keep things standardized, the classes inherit their structure from the abstract classes in appendixes A.2 and A.3. This keeps the maintenance of the plugin as easy as possible and provides the inheritable attributes and methods, mainly get() and set() methods, for the mass initialization and toggle processes to go as smoothly as possible. Additionally, two implementable methods are defined: the run() and setup() method.

The run() method is essentially the only function that differs between the features, as the setup() method only defines the abstract variables with the classes' unique key, name (here used as more of a description, the description variables are not used at this time at all) and is_boolean values. In the main features, the run() method initializes their respective sub-features into a similar feature array as in the main class, and in the sub-features it defines their functionality with the help of any necessary internal functions. A simple example of a sub-feature's run() method can be seen in appendix A.4. There a callback function, defined below the run() method, is hooked to the admin_bar_menu action to remove a menu item that would allow a user to customize the appearance of the site directly in the dashboard. This is usually to be avoided, as inexperienced users might break the looks of a site by accidentally changing the markup or the styles. The majority of the sub-features operate according to this primary pattern of hooking into an action or a filter and executing a callback function. There are, however, two exceptions in the localization feature, seen in appendixes A.5 and A.6, that break this regularity and general structure of the classes.

4.2.2 Semi-classes

The Localization_String_Translations and Localization_Polyfill classes provide localization functionality for creating a multilingual website, mainly in the form of function wrappers for the plugin Polylang [83], and are doing so by being sort of semi-classes. Polylang is used to easily translate content on a website, but can also be used to register strings in the code for translation and managing those translations in the admin dashboard. The two classes focus on the latter code translation part, with the Localization_String_Translations handling and registering string translations and Localization_Polyfill preserving functionality without Polylang. The former provides useful wrappers for combining the most commonly used methods when translating and printing strings in the code, as well as concentrating the translations into one central, easily manageable location. By the virtue of the methods only being wrappers, the underlying function calls must be made at some point in the execution, and that is where the polyfill functions of the latter class are needed. This is done partly to provide a fail-safe and fallback, preventing fatal errors on a page if the Polylang plugin stops working, but also as a form of standardization. The way of centrally handling and registering strings has been found effective and easy to use, so the polyfills provide a way of replicating the functionality (to a necessary degree) on sites where Polylang is not used, but where the wrapper functions can be used to print out frequently used strings from one location. This makes maintaining and updating the strings less of a hassle.

The aforementioned semi-class structure becomes relevant when using the functionality of the two classes. Their base structure is inherited from the abstract class, as with all the other sub-features, but the main functions are declared outside the class itself, in the same file. The Localization_Polyfill class, for example, does not actually execute anything in its run() method. This is a design decision to keep the flexibility of the overarching feature toggling mentality intact, but to simultaneously provide ease of use for the developers when using the functions. By having the normal sub-feature structure, the functionalities can be disabled, but by declaring the functions outside of the classes they are usable without having to declare new instances of the classes every time.

With the general infrastructure of the Aucor Core plugin explained, the next section covering the test writing process and the tests should be more tangible. The unified structure of the classes as well as the regularity of their way of interacting with the WordPress core made the testing more focused in the sense that the bigger question became how to test the classes regarding the framework (syntax, methods) instead of the functionality.

4.3 Writing the tests

The test writing was the process that consumed the bulk of the time for the practical part of the thesis. Although testing could be divided into smaller chunks, it became very apparent why companies that do not handle testing correctly during development struggle (time and budgetwise) to achieve sufficient testing at the end of a project, let alone retro-actively. As has been established, however, testing is a critical part of software development and with the right methodologies, such as CI, it can become an inherit activity throughout the development process. The scope of the tests was to cover all the features and sub-features, to make sure that when new functionality is introduced nothing previously added would break or be altered. In a plugin such as the Aucor Core, where much of the functionality changes the behavior of the WordPress core, regression testing should be in big focus, as undoing something in one feature that was set in a previous one is frighteningly easy.

4.3.1 Test design

The first question that had to be answered was how to test the features and sub-features, meaning the testing method. Having access to the source code and a fairly good understanding of how the plugin was supposed to work, the decision of white-box testing and path coverage was not difficult to make. Being able to base the input characteristics on internal structure and logic analysis, rather than just specifications, was immensely helpful. The caveat to this, however, was that a big part of what happened internally came from the WordPress core, such as filters and built-in functions used by the plugin. This introduced an element of gray-box testing, as documentation was the primary source of information, but a significant amount of time was still spent deciphering the source code when the specifications were lacking. In the end, it did not make the testing much more difficult, even though it did create some gray areas and overlaps regarding testing responsibility. The consensus when testing WordPress plugins is to only test what the plugin one is testing does, not the functions provided by WordPress. They are assumed and expected to be tested by the designated WordPress development team. However, the only role some of the features had was to execute a built-in function to change some default behavior and in those cases testing was done on those functions as well. At least this way every path was being covered and it could with greater confidence be said that the paths were taken in the correct manner. While it may to some seem like non-optimal use of developers' time, focusing on the path coverage like this, rather than thinking too much about whether a function is the responsibility of WordPress or not (which meant a number of redundant tests were created), made the testing quite straightforward. The classes and the methods in the Aucor Core plugin created very isolated areas and paths to test and, in combination with the function specifications provided by WordPress, for the more gray-box testing the number of inputs to test and outputs to analyze were reduced to a manageable level.

The next question was the definition of testing levels and objectives. When defining the scope of the thesis, the levels were set as unit and integration testing, and closely coupled with them the objective naturally became functional testing (ensuring the software does what it is supposed to do). The objective held up, but what became apparent as more tests were written was that the levels of the tests were not as easily defined as first expected. Practically it made little difference, but in the end almost every test technically became an integration test. As mentioned before, most of the sub-features interact with the core through hooks, which already makes them integration tests, but even if that was neglected the sub-features and their methods more often than not used built-in WordPress functions to accomplish their tasks. This is largely due to the infrastructure of the WordPress core, such as the separation of functionality in admin versus public views and its heavy use of global variables, which makes accessing and changing data a cumbersome task (sometimes even with the built-in functions). Again, this did not make things vastly more difficult, as all the built-in functions used did not need to be explicitly tested, but it did introduce an element of complexity to the tests, especially when trying to mock certain inputs or create the conditions under which certain paths were taken.

Finally, a more operational decision had to be made: how to construct the test suite. As the difference between unit and integration tests was noted to be almost non-existent, or rather almost no pure unit tests existed, the division into those two groups was not really suitable. After writing the first few tests and taking the features one at a time, the most natural way of grouping them then became just that, by features and sub-features. The tests were divided into directories according to the main features and further into separate files according to the sub-features. In the files, the functionalities were divided into test functions, not as finely as one assertion per function, but one functionality per function, to keep an easy overview intact. In the end, the organization of the tests in a plugin with this size makes very little difference performance-wise, as no separation of tests was needed during the execution. To keep in line with good conventions of modularity and simplicity and to ensure scalability for future additions, however, it was seen as a worthwhile step to take.

4.3.2 Test anatomy

An example of the anatomy of a test can be seen in appendix A.7. Tests run with PHPUnit are created as classes that extend a testing class provided by the framework. In this case, a more customized test class (based on the framework's class) was used, which was a part of the utility tools included in the install script when setting up the local testing environment. The test classes allow the declaration of variables and methods, both "normal" and test methods. The test methods are public functions with a test* naming convention that are run during the testing. They contain the assertions that are the bases for unit and integration tests. The other "normal" methods can be used as data providers or helper functions for the tests, such as the setup() and teardown() methods. Those two, specifically, are methods that are run before and after every test method respectively without explicit function calls and provide a way to set up and remove conditions for all tests in a class in a concise and centralized manner. For testing the features and sub-features, the setup() function was where the feature under test was initialized, as an instance of the class was needed to access the functionality inside.

When the feature was initialized, it in turn initialized the sub-features belonging to it, which were used in the sub-feature tests to indirectly test the initialization process through the parent feature. In the teardown() method, the features were unset just to keep things clean and to not carry anything over to the next tests.

For the main features, the testing was limited to the inherited methods from the abstract class, as they did not have any other significant role than to initialize the sub-features. The methods were a setup() and various get() and set() methods, as seen in appendix A.2, for internal values. When the __construct() method was run as the class was initialized, the setup() as well as the set() methods were all used during the process. This made covering the paths of the main features as simple as checking the return values from the get() methods, because if they returned anything the setup() and set() methods had done their job.

The tests for the sub-features started the same way as the features by checking their internal values from the get() methods, but branched out when testing their individual functionalities. The vast majority of the sub-features functioned according to the primary structure of having a callback function attached to a hook, so the coverage was achieved by confirming that the attachment to the hook was successful and checking that the return values from the callback functions were correct. The latter often included mocking arguments to use as inputs or to compare with the outputs. For some of the tests, the mocks were as simple as numeric values, but others required setting up more complex objects, such as navigation bars or users with different levels of permissions. Factories, which also were included in the install script, helped to an extent with the more complex mocking, but the excessive use of global variables and the meager use of object oriented programming in WordPress became very apparent when dealing with the bigger entities. The solution to how to cover a path was not always simple and as global variables persisted through the entirety of test execution, constant attention had to be maintained to not cause problems for tests running later, which sometimes used the same variables as earlier tests. The task was not made easier by the fact that no comprehensive lists of WordPress' use of global variables existed, which meant reading the source code or relying on information deduced by other users.

The semi-classes were arguably the ones that differed the most from the mass of sub-features. Luckily, the partial class structure did not complicate the testing of them. It was mainly a question of properly testing all the different variations of functions with various combinations of inputs. Those sub-features are, after all, the most volatile ones, as a problem with their functions can potentially cause fatal errors on a site, opposed to the user only losing that particular functionality when the other sub-features fail. This is because they are the only sub-features that developers uses in their code, rather than as features through the WordPress interface, so the margin of error is less.

4.4 Remote testing environment

After writing the tests, the last step to tie everything together was to configure the remote CI server. At this point, features and sub-features could (and should) be tested locally, but despite all the standardization and control of the processes there was still a chance of the dreaded "it works on my machine" phenomenon or the developers just not running the tests. And even if Aucor's own developers have their standards that reduce this, the possibility of other people contributing code is not unimaginable, as many of Aucor's products are open-source projects. In those cases, there is no guarantee of what standards the contributors have, let alone if they follow any testing conventions. These were the problems that a remote server was there to solve, providing an unbiased base for automatically running the tests when code is committed to a repository.

4.4.1 Connecting to Travis

To begin with, signing up to Travis was required, which was done by connecting a GitHub account to the service. When logged in, an overview of the activity and the status of repositories and builds could be seen from a dashboard. This was also where GitHub repositories could be added for tracking. To ensure that the integration of the service did not cause any problems or disruptions in the main repository, a fork of the Aucor Core repository was created, until everything was in place and ready to be used. After the forked repository had been connected by choosing it from the list of repositories, it was ready to be automatically built and tested by Travis.

4.4.2 Configuring the Travis file

Before being of any use, the server had to be given configurations and instructions on how to react to a trigger. The configurations were made in the Travis file mentioned earlier, which was generated with the test scaffold. The file is used by Travis to create a build: a step by step guide on how to set up the environment needed to run the tests and how to run the tests. The file is in YAML (YAML Ain't Markup Language) [84] format and is formally specified using a JSON (JavaScript Object Notation) [85] schema [86]. The configurations were given as key-value pairs, both the more straightforward single-value keys, such as the language, operating system and environment variables, and the more complex ones, such as the job matrix and the commands to be run. The Travis file for the Aucor Core plugin can be seen in appendix A.8. In total, there were 30 key values that could be configured [86], most of which in this case were left on default (not declared), as the set up was not overly complicated. The most noteworthy ones were the branches, jobs, before_script and script keys.

Branches

In the branches key, a curated list of branches can be given to either be explicitly included or excluded from the build tracking. This is to control the set of branches that fire the automation sequence and can be used to exclude development branches for unfinished work or to only include the main branch, for example. For the Aucor Core plugin, it was decided that all branches should trigger a build, as it ensures that the feature branches are clean before merging with the master branch and that the master branch is checked one more time once the merge is done. By default, Travis automatically reacts to pull-requests, meaning that commits coming from outside sources do not explicitly have to be defined, as they are built automatically when such have been detected.

Jobs

The jobs key accepts a matrix of build variables, which are separated into individual jobs run in parallel during the execution of the build. In Aucor Core's case, the latest PHP versions were combined with the different (in the plugin) supported WordPress core versions. The tests were run on them all to ensure that the plugin works in all of its intended target environments. This provided an easy way to catch backwards compatibility problems with older core versions, all of which might not always be explicitly tested locally (although possible). The jobs can be created by defining them explicitly or by giving sets of arguments, which are expanded to a matrix of combinations of those arguments. The former is preferred, as the latter can result in a number of irrelevant combinations being run, which unnecessarily strain the servers and occupy resources.

Before_script

The before_script key is used to define commands to be run before any tests are executed. For Aucor Core, it basically contained the same steps that was done for the local testing environment, with the exceptions of excluding the Seravo base and including an optimization step that was included in the file by default. The base is included locally to help developers with the actual development of the plugin, not just the testing. When it comes to testing, the plugin is independent in the sense that it contains all the necessary files needed to run on any WordPress instance. When testing the plugin remotely, then, the base adds no value as the WordPress instance, tools and database from the scaffold install script can be used. The optimization step was to remove Xdebug [87] if present, which is a PHP debugger tool that is completely unnecessary when running on the remote server and which removal causes a significant performance increase.

Script

The script key is where the test commands are finally added. Environment variables provided in the jobs matrix can here be used to run different tests. In Aucor Core's case, the only separation of tests were the PHPUnit tests and the PHP_CodeSniffer test. The latter was only run in one job and with the latest PHP version, as it only performs a static syntax check of the code and does not rely on the specific version of the WordPress core.

With the Travis file configured, subsequent commits to the repository now activated the remote server to initialize, execute builds and run the tests on the configured target environments. From the Travis dashboard, the builds and the individual jobs could be inspected and even followed in real time, with detailed logs provided to be analysed afterwards as well. After each build, a visual status was attached to the triggering commit over at GitHub, making the status of the source code transparent without having to access the Travis dashboard. With a local testing environment set up, tests for the features and sub-features written, and a remote CI server configured, a fully functioning CI pipeline had been implemented.

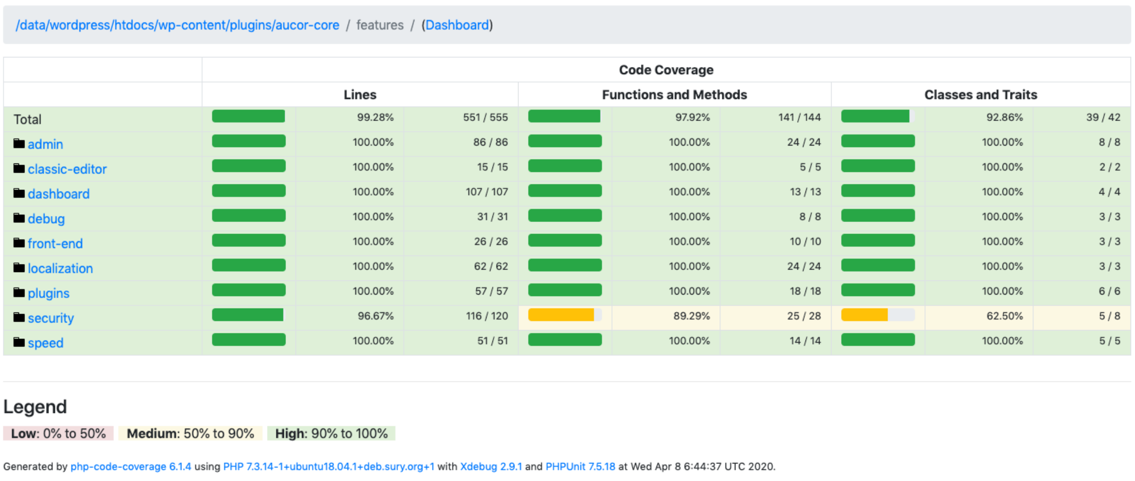
5 Results and evaluation

The main contributions of CI for a development process are the preventive measures and the resulting reduced risk. For the scope of the thesis, the development frequency of the Aucor Core plugin was not high enough to collect hard data for the CI assimilation metrics suggested as one of the implications of the infusion stage in the study by Eck et al. [9]. A definitive number could therefore not be given on how much the CI pipeline reduced errors or prevented problems. However, the results described below prove that the pipeline is a worthy addition and a step in the right direction for a smoother and safer development process. Additionally, a more qualitative metric was taken into account when reflecting on the results: a perceived increase in the quality of life of the developers, meaning some aspect of their daily work had become easier, faster or less frustrating, for example. If developers find a system easy and effective to use, they will most likely continue using it.

5.1 The CI pipeline

5.1.1 Local development

Extending the local development environment to encompass testing functionality was relatively painless, as a result of the already built-in support for the tools in WordPress and WP-CLI. There are other frameworks that could have been used instead of the WP-CLI scaffold, but as the files and the tools from it could be re-used later when configuring the Travis environment, it seemed like a natural choice. The idea of adding the testing functionality on top of the already familiar platform was to minimize the number of problems the developers could encounter when re-adjusting to the new practices of testing and CI in general.


/data/wordpress/htdocs/wp-content/plugins/aucor-core / features / (Dashboard)

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total	🟩	99.28%	551 / 555	🟩	97.92%	141 / 144	🟩	92.86%	39 / 42
admin	🟩	100.00%	86 / 86	🟩	100.00%	24 / 24	🟩	100.00%	8 / 8
classic-editor	🟩	100.00%	15 / 15	🟩	100.00%	5 / 5	🟩	100.00%	2 / 2
dashboard	🟩	100.00%	107 / 107	🟩	100.00%	13 / 13	🟩	100.00%	4 / 4
debug	🟩	100.00%	31 / 31	🟩	100.00%	8 / 8	🟩	100.00%	3 / 3
front-end	🟩	100.00%	26 / 26	🟩	100.00%	10 / 10	🟩	100.00%	3 / 3
localization	🟩	100.00%	62 / 62	🟩	100.00%	24 / 24	🟩	100.00%	3 / 3
plugins	🟩	100.00%	57 / 57	🟩	100.00%	18 / 18	🟩	100.00%	6 / 6
security	🟩	96.67%	116 / 120	🟨	89.29%	25 / 28	🟨	62.50%	5 / 8
speed	🟩	100.00%	51 / 51	🟩	100.00%	14 / 14	🟩	100.00%	5 / 5

Legend

Low: 0% to 50% **Medium:** 50% to 90% **High:** 90% to 100%

Generated by php-code-coverage 6.1.4 using PHP 7.3.14-1+ubuntu18.04.1+deb.sury.org+1 with Xdebug 2.9.1 and PHPUnit 7.5.18 at Wed Apr 8 6:44:37 UTC 2020.

Figure 5.1: Feature and sub-feature code coverage

5.1.2 Tests

When the test suite was deemed finished (for now), it mostly contained integration tests and had a code coverage of 99.2% of the features and sub-features, pictured in figure 5.1. Both "shortcomings" were a direct result of the WordPress core, but they were still more than acceptable for the CI pipeline. The lack of pure unit tests reflects the way plugins (of Aucor Core's type) have to navigate the landscape of global variables and lack of object oriented internal structures of the WordPress core, not that functionality was left unchecked by the tests.

The internal structure of the core was also the reason a full 100% coverage was not reached, as the three uncovered paths in question were unreachable by the tests. Two of the cases concerned constants that disabled default functionality for security reasons. In the code, both constants were checked to be defined and subsequently defined by their respective sub-features if the checks failed. The problem was that a state could not be reached during testing where the constants were not defined, since the definition of them happened before the tests were even run when the code was included. The paths where they were explicitly defined by the sub-features could, thus, never be taken in the tests. The only thing that could be done was to test that the constants were in fact defined, which ultimately was the most important thing to check for anyway. This did leave one line of code in each sub-feature uncovered, but if problems were to occur in the definition of the constants, the assertions that are in place would catch them, since they do not take into account when the definitions happen, only that they have happened.

The third path that was not covered had to do with a redirection of users on a specific admin page, if they did not have the required permissions to view the content. The way redirection works in WordPress and PHP requires any code using it to also call either the die() or the exit() function. These stop PHP from processing any further data after the redirection, which prevents any data from the original location leaking into the HTTP response that is returned as a result of the redirection. Clearly, then, the die() or exit() function calls are very warranted. Unfortunately for testing, however, the functions also stop PHPUnit from executing any further tests. The redirection path, thus, remained uncovered, but that particular functionality was one of the ones using only built-in WordPress functions, so the confidence of it working still remained high. Something that was briefly considered was a special testing constant, which might have enabled testing of the problematic paths, but it was fairly quickly discarded as it would mean affecting the actual code by injecting checks for the testing constant. While it would not necessarily have been a huge undertaking, that kind of hacky solution could hardly be considered refactoring the code in the name of testability. Also, as the functionality where the paths were not fully covered were still covered enough to retain confidence in them, it was not deemed to add enough value to be worth looking further into this solution.

While the full long-term benefits and effects of the tests require more development cycles than what could be provided during the thesis to be noted, a few immediate results were achieved due to their introduction. On three occasions, a method was discovered to not work as intended (although not entirely broken) when constructing the tests for it. In all cases, the fault did not directly lie with anyone who wrote the code, but was caused by changes in the built-in WordPress functions that the methods used in combination with custom code. This further highlights the importance of regression testing, as updates might slightly change the functions provided by WordPress and therefore break a functionality without developers touching the code. The same risk of course exists for the redirection method mentioned above, which path was not fully covered, but in that case the method contained no custom code and therefore has a lesser chance to fail. A fair amount of refactoring was also done, in many cases to ease the testability of functionality that used anonymous functions, but also because the tests now ensured that everything would indeed work as before. The hook callbacks, for example, were initially defined with the literal class names in every sub-feature, which often made for unnecessarily long definitions. Instead, they could be defined with the $this keyword, which reduced the error-proneness when considering maintenance of the sub-features and made overviewing the classes much easier.

As for the testing speed, the features and sub-features could now all be tested in a matter of 10 seconds. Testing every functionality manually could before take several if not tens of minutes, so the speed-up with automated tests was definitely worth it.

5.1.3 Remote testing

Configuring the remote testing environment with Travis included plenty of trial and error, but after 21 builds the CI pipeline could finally be called complete, with passing builds on the integration server. The basic idea of setting it up was clear from the start, but the details required their fair share of fine-tuning, especially the steps in the before_script phase. Although the scaffold install script made things much easier, trying to fit everything together was not as straightforward as it seemed at first.

The biggest problems were caused by PHPUnit and its dependencies. Although the Aucor Core is not supposed to be backwards compatible to the extreme, such as with PHP5 or any ancient version of the WordPress core, it still needs to be ensured that it works on at least the versions in recent memory. Active updating of both PHP and the WordPress core is performed on the websites in production, but legacy issues can hold the process back on some sites. Unfortunately, the last WordPress core versions that are considered recent memory and that some of the websites in production still use, 5.0 and 4.9, required PHPUnit 6 to test. This meant that both version 6 and 7 of PHPUnit had to be included in the setup, since the support for PHP in PHPUnit 6 ends at 7.2, which in turn is not enough for the newer core versions. For this to work, the frameworks had to be installed globally on the server, as problems arose with the need to allow "conflicting" dependencies of the two versions. The Aucor Core plugin's Composer lock file, designed for exactly such scenarios, forbade it be done through the non-global install.

The struggle was worth the effort, though, because much like the tests that revealed issues when being constructed, the remote server already ended up working exactly as the fail-safe it was intended. Running the tests on the older WordPress core versions caught a few minor conflicts in the expected behaviour, which could be fixed immediately before they escalated into anything more serious later. The whole ordeal with the different PHPUnit versions also acted as practice for the inevitable need to include version 8 of PHPUnit, which is the first version of the framework to support the already released PHP7.4. In the end, Travis CI turned out to be a good fit for the Aucor Core plugin. Arguments could be made that other services can provide better customizability, but with the plugin and the testing environment being relatively simple, not much else needs to be customized than what already is. The service was easy to use, it could be integrated into other services used in the company, such as Slack, and what already became apparent when running the failing builds: the logs and reports on the Travis CI dashboard were immensely helpful when problems occurred. The build times on the remote server were on average around 9 minutes, which is significantly longer than locally, but understandably so, as 10 build with different PHP and WordPress versions were first built and then tested. The builds generally ran in parallel with sets of 5 builds. The fact that the build duration is longer on the remote server is not as impactful as it would be locally either, as the duration can still be counted in minutes and the builds are built and tested automatically, so the developers can do something else for the duration of the external builds.

5.2 Adopting CI

To what extent Aucor will ultimately assimilate CI remains to be seen, but the company has by all accounts been introduced to at least the targeted acceptance stage presented in the study by Eck et al. [9]:

- Devising an assimilation path. The groundwork for the extended nucleus approach has been laid. A core system has been put in place for a product in production, which can be used as a base and an example to extend the activity to other products.
- Overcoming the initial learning phase. The developers have been provided with education through a presentation and through written instructions on the pipeline itself and the concepts involved. This should reduce the steepness of the learning curve once the developers start using CI more extensively.
- Dealing with test failures immediately. To promote good code quality and dealing with problems immediately, restrictions have been put in place that forbid code that has not passed all tests (including the code writing standards) to be

merged into the master branch. This also removes the possibility of any accidental pushes or merges (that have not passed the tests) into the master branch.

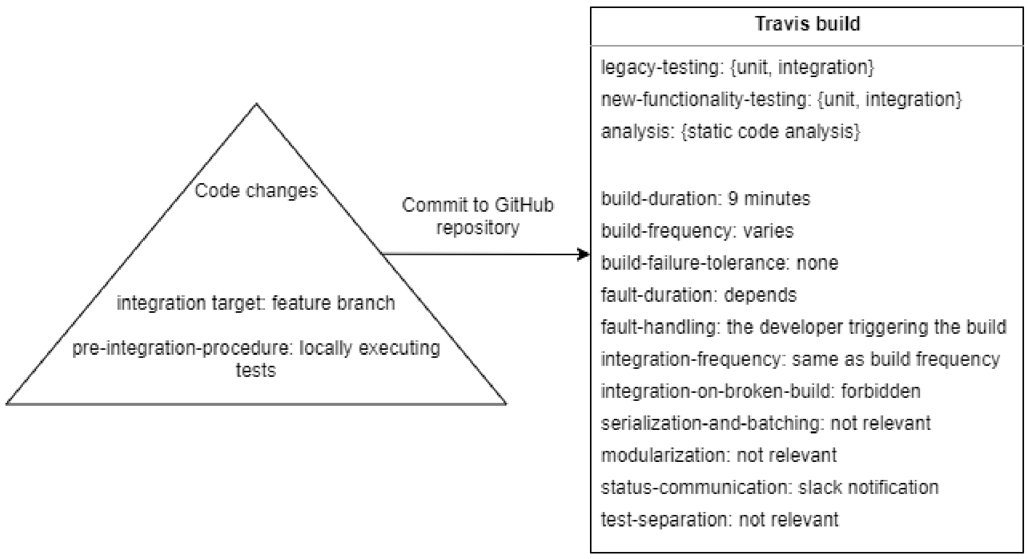
- Introducing CI for complex systems. The Aucor Core plugin is not by any means the most complex systems in the world, but in the context of WordPress and the other Aucor products it represents a standard complexity level. Thus, introducing it for the Aucor Core means that it can further be introduced to other products as well, as the target environments and systems are relatively homogeneous and standardized throughout the company.

Apart from these points, as mentioned earlier, the pipeline has already proven successful as a proof of concept through the problems that were fixed due to the introduction of the tests and the regression testing on older WordPress versions with Travis CI. This should at the very least lead to the developers to not reject the system immediately, seeing that it actually is an advancement towards a safer way of developing their products. The developers involved have also expressed general excitement towards the possibilities that such a pipeline could provide in the development processes of other products. The analysis done on the latter stages of the CI assimilation did not reveal any immediate obstacles either, so the continuation of the progress and the assimilation is now up to actively engaged developers and the support from the company.

5.3 A descriptive model of CI

Lastly, to help the company in its efforts to progress their use of CI to the next stages in the future, a descriptive model of the current system in place has been created in figure 5.2, as described in the study by Ståhl et al. [8]. The model can work as a base for comparing different CI pipelines, but also (and more importantly here) as a way to gain insight into one's own pipeline and how to improve it. By referring to this model, the developers have a foundation on which decisions and progress can be based. Although not overly complicated, the model can help with the overview of the system and contains the final stances on the variation points (for this iteration) discussed in the study by Ståhl et al. [8] and in the context of Aucor in chapter 3.

The node on the left describes the input attributes, the arrow describes the build trigger on the remote integration server, and the node on the right describes the scope attributes on the top as well as the build characteristics and result handling attributes



Code changes
integration target: feature branch
pre-integration-procedure: locally executing tests
Commit to GitHub repository
Travis build
legacy-testing: {unit, integration}
new-functionality-testing: {unit, integration}
analysis: {static code analysis}
build-duration: 9 minutes
build-frequency: varies
build-failure-tolerance: none
fault-duration: depends
fault-handling: the developer triggering the build
integration-frequency: same as build frequency
integration-on-broken-build: forbidden
serialization-and-batching: not relevant
modularization: not relevant
status-communication: slack notification
test-separation: not relevant

Figure 5.2: Descriptive model of the CI pipeline

on the bottom. The legacy testing listed as one of the scope attributes refers here to the regression testing done on the code that is already there, before any new changes. Three of the attributes in right node were marked as "not relevant", as discussed in chapter 3, but included for future reference when they might become relevant.

The attributes that were left ambiguous were the build frequency and the fault duration, as they are extremely dependent on the situation. As the Aucor Core does not undergo changes all the time, but rather in phases or even just occasionally, the build frequency varies greatly. The same goes for fault duration, which cannot be estimated before a fault happens. The reaction to the fault and the measures taken to repair it should nonetheless be immediate.

6 CONCLUSIONS AND FUTURE WORK

The effectiveness of CI is undoubtedly more accentuated in big cross-functional development teams and software, where the code is much more fragmented and the dependencies between the parts are more complex. However, through the tools that have been developed to increase the benefit of the "integrate often" mindset, smaller development processes can achieve substantial gain as well. Where there was uncertainty and time-consuming manual labor before, there is now automated processes that ensure that each step in the building and testing of the software is done every time and in the same manner. Through the systems put in place, the developers at Aucor will hopefully be more inclined to put effort towards actively building the software, knowing that there is tangeable proof that everything still works as expected after changes in the code and that the testing will not take very much time to perform.

Although by no means perfect, the pipeline works as a proof of concept for a more streamlined development and even opening doors for more future improvements. For a WordPress plugin such as Aucor Core, which is not only available on GitHub, but in the official plugin directory of WordPress as well, the process of releasing the plugin to the official directory contains multiple steps. This is done manually for now, but could potentially be handled automatically as an extension of the CI pipeline, effectively making it either CD or even CDE (depending on how far the automation can be taken). This would further reduce the number of error-prone manual tasks and save time for the developers.

In the end, the CI tools and pipelines can only take a team or a company so far. They provide fail-safes, prevent problems and make work more effective, but the work still has to be done. Version control does not replace communication, automation does not make developers integrate more frequently and tests do not save bad code. It is the mindset and teamwork of CI that truly produces the results, with the tools only helping the developers go from great to even greater.

BIBLIOGRAPHY

[1] Http request example (image). Accessed on 21.03.2020. URL: https://media.prod.mdn.mozit.cloud/attachments/2016/08/09/13687/5d4c4719f4099d5342a5093bdf4a8843/HTTP_Request.png.

[2] Http response example (image). Accessed on 21.03.2020. URL: https://media.prod.mdn.mozit.cloud/attachments/2016/08/09/13691/58390536967466a1a59ba98d06f43433/HTTP_Response.png.

[3] Margaret Rouse. What is best practice? - definition from whatis.com. Accessed on 29.08.2019. URL: https://searchsoftwarequality.techtarget.com/definition/best-practice.

[4] Martin Fowler. Continuous integration (original version). Accessed on 29.08.2019. URL: https://martinfowler.com/articles/originalContinuousIntegration.html.

[5] Aucor oy. Accessed on 05.09.2019. URL: https://www.aucor.fi/.

[6] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. 03 2013. URL: https://www.researchgate.net/publication/266632251_Experienced_Benefits_of_Continuous_Integration_in_Industry_Software_Product_Development_A_Case_Study, doi:10.2316/P.2013.796-012.

[7] Martin Fowler. Continuous integration. Accessed on 08.11.2019. URL: https://martinfowler.com/articles/continuousIntegration.html.

[8] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48 – 59, 2014. URL: http://www.sciencedirect.com/science/article/pii/S0164121213002276, doi:https://doi.org/10.1016/j.jss.2013.08.032.

[9] Alexander Eck, Falk Uebernickel, and Walter Brenner. Fit for continuous integration: How organizations assimilate an agile practice. 08 2014. URL: https://www.researchgate.net/publication/280489870_Fit_for_Continuous_Integration_How_Organizations_Assimilate_an_Agile_Practice.

[10] Wordpress.com: Create a free website or blog. Accessed on 05.09.2019. URL: https://wordpress.com/.

[11] Edsger W. Dijkstra. Notes on structured programming. Accessed on 13.11.2019. URL: http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/EWD-notes-structured.pdf.

[12] Aucor/aucor-core. Accessed on 05.09.2019. URL: https://github.com/aucor/aucor-core.

[13] Github. Accessed on 29.11.2019. URL: https://github.com/.

[14] Travis ci - test and deploy your code with confidence. Accessed on 13.12.2019. URL: https://travis-ci.org/.

[15] What is lean? Accessed on 13.09.2019. URL: https://www.lean.org/WhatsLean/.

[16] Kent Beck, James Grenning, and Robert C. Martin et al. Manifesto for agile software development. Accessed on 29.08.2019. URL: http://agilemanifesto.org/.

[17] What is agile software development? | agile alliance. Accessed on 29.08.2019. URL: https://www.agilealliance.org/agile101/.

[18] Don Wells. Agile software development: A gentle introduction. Accessed on 07.11.2019. URL: http://www.agile-process.org/.

[19] Jim Highsmith. History: The agile manifesto. Accessed on 07.11.2019. URL: http://agilemanifesto.org/history.html.

[20] Kent Beck, James Grenning, and Robert C. Martin et al. Principles behind the agile manifesto. Accessed on 20.09.2019. URL: https://agilemanifesto.org/principles.html.

[21] Margaret Rouse. What is waterfall model? - definition from whatis.com. Accessed on 26.09.2019. URL: https://searchsoftwarequality.techtarget.com/definition/waterfall-model.

[22] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *Proc. Espoo 2002*, pages 3–107, 01 2002. URL: https://www.researchgate.net/publication/292542090_Agile_Software_Development_Methods_Review_and_Analysis.

[23] Murat Erder and Pierre Pureur. Chapter 7 - continuous architecture in practice: A case study. *Continuous Architecture*, 87:161 – 185, 2016. URL: http://www.sciencedirect.com/science/article/pii/B9780128032848000075, `doi:https://doi.org/10.1016/B978-0-12-803284-8.00007-5`.

[24] Don Wells. Most important first. Accessed on 20.03.2020. URL: http://www.agile-process.org/first.html.

[25] Don Wells. Iterative planning. Accessed on 20.03.2020. URL: http://www.agile-process.org/iterative.html.

[26] Martin Fowler. Refactoring. Accessed on 11.11.2019. URL: https://refactoring.com/.

[27] Don Wells. Working software. Accessed on 20.03.2020. URL: http://www.agile-process.org/working.html.

[28] Alistair Cockburn and Jim Highsmith. Agile software development, the people factor. *Computer*, 34(11):131–133, 2001. URL: https://ieeexplore.ieee.org/abstract/document/963450.

[29] Aucor/aucor-starter. Accessed on 17.10.2019. URL: https://github.com/aucor/aucor-starter.

[30] Don Wells. Extreme programming: A gentle introduction. Accessed on 18.10.2019. URL: http://www.extremeprogramming.org/.

[31] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176 – 1859, 2017. URL: http://www.sciencedirect.com/science/article/pii/S0164121215001430, doi:https://doi.org/10.1016/j.jss.2015.06.063.

[32] Don Wells. Continuous integration. Accessed on 08.11.2019. URL: http://www.extremeprogramming.org/rules/integrateoften.html.

[33] What is continuous integration? | agile alliance. Accessed on 29.08.2019. URL: https://www.agilealliance.org/glossary/continuous-integration/.

[34] Vahid Garousi, Michael Felderer, and Feyza Nur Kılıçaslan. A survey on software testability. *Information and Software Technology*, 108:35 – 64, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0950584918302490, doi:https://doi.org/10.1016/j.infsof.2018.12.003.

[35] Francesca Lonetti and Eda Marchetti. Chapter three - emerging software testing technologies. *Advances in Computers*, 108:91 – 143, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0065245817300529, doi:https://doi.org/10.1016/bs.adcom.2017.11.003.

[36] Marc Roper. Software testing. *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 41 – 47, 2003. URL: http://www.sciencedirect.com/science/article/pii/B0122274105008590, doi:https://doi.org/10.1016/B0-12-227410-5/00859-0.

[37] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. Chapter 1 - technologies for web applications. *Designing Data-Intensive Web Applications*, pages 3 – 58, 2003. URL: http://www.sciencedirect.com/science/article/pii/B9781558608436500025, `doi:https://doi.org/10.1016/B978-155860843-6/50002-5`.

[38] Http | mdn. Accessed on 21.03.2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP.

[39] Html: Hypertext markup language | mdn. Accessed on 20.11.2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTML.

[40] Css: Cascading style sheets | mdn. Accessed on 20.11.2019. URL: https://developer.mozilla.org/en-US/docs/Web/CSS.

[41] Javascript | mdn. Accessed on 20.11.2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[42] An overview of http - http | mdn. Accessed on 21.03.2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[43] Http request methods - http | mdn. Accessed on 21.03.2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[44] W3techs - extensive and reliable web technology surveys. Accessed on 06.09.2019. URL: https://w3techs.com/.

[45] Michel Valdrighi. b2 - a classy weblog tool. Accessed on 20.11.2019. URL: https://cafelog.com.

[46] Siobhan McKeown. Chapter three - on forking wordpress, forks in general, early wordpress and the community. Accessed on 20.11.2019. URL: https://wordpress.org/about/history/chapter3.pdf.

[47] Tag: Fortune 500 | wordpress.org. Accessed on 06.09.2019. URL: https://wordpress.org/showcase/tag/fortune-500/.

[48] Project organization - make wordpress core. Accessed on 20.11.2019. URL: https://make.wordpress.org/core/handbook/about/organization/.

[49] Administration screens | wordpress.org. Accessed on 20.11.2019. URL: https://wordpress.org/support/article/administration-screens/.

[50] Php: Hypertext preprocessor. Accessed on 20.11.2019. URL: https://www.php.net/.

[51] Mysql. Accessed on 20.11.2019. URL: https://www.mysql.com/.

[52] About us: Our mission | wordpress.org. Accessed on 20.11.2019. URL: https://wordpress.org/about/.

[53] Hooks | plugin developer handbook | wordpress developer resources. Accessed on 20.11.2019. URL: https://developer.wordpress.org/plugins/hooks/.

[54] Jordi Cabot. Wordpress: A content management system to democratize publishing. *IEEE Software*, 35(3):89–92, 2018. URL: https://ieeexplore.ieee.org/document/8354434.

[55] Wordpress plugins | wordpress.org. Accessed on 20.11.2019. URL: https://wordpress.org/plugins/.

[56] What is a theme? | theme developer handbook | wordpress developer resources. Accessed on 20.11.2019. URL: https://developer.wordpress.org/themes/getting-started/what-is-a-theme/.

[57] Wordpress themes | wordpress.org. Accessed on 20.11.2019. URL: https://wordpress.org/themes/.

[58] What is a plugin? | plugin developer handbook | wordpress developer resources. Accessed on 21.11.2019. URL: https://developer.wordpress.org/plugins/intro/what-is-a-plugin/.

[59] Detailed plugin guidelines | plugin developer handbook | wordpress developer resources. Accessed on 21.11.2019. URL: https://developer.wordpress.org/plugins/wordpress-org/detailed-plugin-guidelines/.

[60] Header requirements | plugin developer handbook | wordpress developer resources. Accessed on 21.11.2019. URL: https://developer.wordpress.org/plugins/plugin-basics/header-requirements/.

[61] Plugin basics | plugin developer handbook | wordpress developer resources. Accessed on 21.11.2019. URL: https://developer.wordpress.org/plugins/plugin-basics/.

[62] The team - make wordpress plugins. Accessed on 21.11.2019. URL: https://make.wordpress.org/plugins/handbook/the-team/.

[63] Teemu Koskinen, Petri Ihantola, and Ville Karavirta. Quality of wordpress plugins: An overview of security and user ratings. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, pages 834–837, 2012. URL: https://ieeexplore.ieee.org/document/6406333.

[64] Sara Rasso, Barry Abrahamson, and Michael Adams et al. Security | wordpress.org. Accessed on 21.11.2019. URL: https://wordpress.org/about/security/.

[65] Actions | plugin developer handbook | wordpress developer resources. Accessed on 21.11.2019. URL: https://developer.wordpress.org/plugins/hooks/actions/.

[66] Advanced topics | plugin developer handbook | wordpress developer resources. Accessed on 21.11.2019. URL: https://developer.wordpress.org/plugins/hooks/advanced-topics/.

[67] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. URL: https://ieeexplore.ieee.org/document/7884954.

[68] Where work happens | slack. Accessed on 04.04.2020. URL: https://slack.com.

[69] Wp-palvelu | premium hosting ja ylläpito wordpress-sivustoille. Accessed on 22.03.2020. URL: https://wp-palvelu.fi/.

[70] Seravo/wordpress. Accessed on 22.03.2020. URL: https://github.com/Seravo/wordpress.

[71] Nginx | high performance load balancer, web server & reverse proxy. Accessed on 29.11.2019. URL: https://www.nginx.com/.

[72] Mariadb foundation - mariadb.org. Accessed on 29.11.2019. URL: https://mariadb.org/.

[73] Vagrant by hashicorp. Accessed on 29.11.2019. URL: https://www.vagrantup.com/.

[74] Oracle vm virtualbox. Accessed on 29.11.2019. URL: https://www.virtualbox.org/.

[75] Command line interface for wordpress | wp-cli. Accessed on 29.11.2019. URL: https://wp-cli.org/.

[76] wp scaffold plugin-tests | wordpress developer resources. Accessed on 29.11.2019. URL: https://developer.wordpress.org/cli/commands/scaffold/plugin-tests/.

[77] Phpunit - the php testing framework. Accessed on 26.03.2020. URL: https://phpunit.de/.

[78] Plugin unit tests - wp-cli – wordpress.org. Accessed on 13.12.2019. URL: https://make.wordpress.org/cli/handbook/plugin-unit-tests/.

[79] squislabs/php_codesniffer. Accessed on 13.12.2019. URL: https://github.com/squizlabs/PHP_CodeSniffer.

[80] Php: Phpunit - make wordpress core. Accessed on 13.12.2019. URL: https://make.wordpress.org/core/handbook/testing/automated-testing/phpunit/.

[81] Composer. Accessed on 29.11.2019. URL: https://getcomposer.org/.

[82] Plugin api/action reference « wordpress codex. Accessed on 26.03.2020. URL: https://codex.wordpress.org/Plugin_API/Action_Reference.

[83] Polylang - making wordpress multilingual. Accessed on 27.03.2020. URL: https://polylang.pro/.

[84] The official yaml web site. Accessed on 04.04.2020. URL: https://yaml.org/.

[85] Json. Accessed on 04.04.2020. URL: https://www.json.org/json-en.html.

[86] Travis ci build config reference. Accessed on 28.03.2020. URL: https://config.travis-ci.com/.

[87] Xdebug - debugger and profiler for php. Accessed on 04.04.2020. URL: https://xdebug.org/.

SVENSK SAMMANFATTNING

KONTINUERLIG INTEGRERING INOM WORDPRESS INSTICKSPROGRAMUTVECKLING

Introduktion

Agil systemutveckling är ett paraplybegrepp som omfattar en mängd metoder och praxisar som under utveckling av mjukvara har insetts förbättra och effektivisera utvecklingsprocesserna. Det som de agila metoderna har gemensamt är att de siktar på att minska på byråkrati, onödig dokumentation och långa utvecklingscykler, samtidigt som de framhäver betydelsen av grupparbete, kommunikation och ett snabbt, iterativt arbetssätt. I och med denna avhandling kommer ett av dessa praxis, kontinuerlig integrering, att införas som en del av utvecklingsprocessen för ett av företaget Aucor Oy:s produkter.

Aucor Oy är ett webbutvecklingsföretag vars fokus ligger i utvecklandet av webbplatser och -tjänster med innehållshanteringssystemet Wordpress, samt utvecklandet av insticksprogram som utvidgar funktionaliteten av systemet i fråga. Produkten vars utvecklingsprocess som behandlas är insticksprogrammet Aucor Core, som företaget använder på flera av sina webbplatser och som innehåller mycket av funktionaliteten som gör tjänsterna snabbare, säkrare och lättare att använda. I och med att den innehåller kritiska förbättringar är det viktigt att försäkra sig om att den fortsätter att fungerar klanderfritt, speciellt då produkten utvecklas vidare och det uppstår en risk att det görs oavsiktliga fel i gammal funktionalitet. Försäkrandet om fortsatt fungerande kommer att underlättas med hjälp av kontinuerlig integrering och målet är att förbättra utvecklingsprocessen på två sätt. För det första skapas mjukvarutester för insticksprogrammets olika egenskaper och för det andra konfigureras en integreringsomgivning, som uppehålls av tredjepartsföretaget Travis CI. Testerna ser till att programmet bibehåller sin tidigare funktionalitet under fortsatt utveckling, medan integreringsomgivningen bidrar med automation som minskar på manuellt arbete och försnabbar utvecklingsprocessen.

Bakgrund

I grund och botten handlar kontinuerlig integrering om minskning av risker, vilket inte kräver dyra eller komplexa system för att åstadkomma, utan en rutin och disciplin att ofta, helst flera gånger om dagen, integrera arbete med en central källkod. När utvecklare jobbar och integrerar med bara små kodavsnitt åt gången undviker de att koden avviker för mycket från den gemensamma utvecklingsinsatsen, vilket därför leder till smidigare integreringar. Problem förebyggs genom att omfånget av kod som kan orsaka fel hålls snävt samt genom att utvecklare allt oftare har tillgång till den senaste versionen av kod att börja jobba på. Trots det faktum att kontinuerlig integrering i princip inte kräver annat än rätt inställning och arbetsmoral hos utvecklarna, så har det under årens lopp emellertid konstaterats att det finns ett antal verktyg som kan hjälpa ett team att uppnå praxisens fulla potential:

- En central datakatalog med versionskontroll för källkoden.
- En automatiserad process för bygget.
- Ett självtestande bygge.
- En integreringsserver.

Många utvecklare anser en central datakatalog vara väldigt vardaglig, men det är fortfarande ett verktyg som inte universellt har tagits i bruk inom branschen. En datakatalog försäkrar att utvecklarna alltid har tillgång till den senaste källkoden utan att behöva leta efter den från flera olika ställen eller samla in den i bitar från andra utvecklare. Versionskontrollen uppehåller en historik av koden i datakatalogen, vilket gör det lättare att spåra fel och vid behov återvända till ett tidigare skede av utvecklingen. Tillsammans förmedlar de också implicit information om projektets status och framsteg, vilket är lättillgängligt för utvecklarna under projektets gång. Kommunikation inom mjukvaruutveckling är inte alltid lätt, så egenskapen att utvecklarna kan få information indirekt via verktygen de använder är en stor fördel.

Ett mjukvaruprojekt består vanligtvis av olika slags filer som måste bearbetas innan de kan användas i sitt slutliga format i ett bygge. Processen kan vara ytterst komplex och innehålla en mängd transformationer, kompileringar och ihoplänkningar av filer, där risken för mänskliga misstag är hög. Därför borde ansvaret överlåtas till en automatiserad process, som ser till att bygget byggs upp på samma sätt varje gång, och som dessutom brukar vara betydligt snabbare än en människa. Detta gäller också för automatiseringen av testning.

Teoretiskt sätt är tester inte nödvändiga för en integrering, men en integrering kan knappast kallas lyckad om mjukvaran inte fungerar eller om funktionaliten har ändrat under integreringen. Därför införs oftast tester, som ser till att ett etablerat utgångsläge inte förvrängs under sammanfogandet av flera komponenter. Tester kan skapas för många olika nivåer och aspekter av en mjukvara, men i kontinuerlig integrering begränsas testerna för det mesta till enhetstester och integreringstester. Enhetstester testar att de minsta beståndsdelarna, som enstaka funktioner, fungerar som de ska och integreringstester försäkrar att beståndsdelarna tekniskt fungerar ihop.

Genom att använda de ovannämnda punkterna är steget till ett effektivt arbetsflöde och kontinuerlig integrering inte stort. Det som ytterligare rekommenderas är en integreringsserver antingen lokalt eller på distans. Servern fungerar som en opartisk plattform där byggen och tester kan köras vid sidan om de som utvecklarna kör på sina egna datorer. Detta skapar en kontroll för att ändringar i koden faktiskt fungerar utanför utvecklarnas egna omgivningar också. Dessutom ger det en möjlighet att kontrollera kvaliteten på koden som utomstående utvecklare, vars standarder, professionalitet eller skicklighet inte går att garantera, bidrar med. Detta är ett återkommande fenomen inom utvecklingen av mjukvara som baserar sig på öppen källkod, som till exempel Wordpress och produkterna som Aucor Oy utvecklar.

Implementering

Det första steget för att införa kontinuerlig integrering i utvecklingsprocessen av Aucor Core var att förbereda den lokala utvecklingsomgivningen för testning. Som testningsverktyg användes Phpunit och Phpcodesniffer, det första för att köra dynamiska tester och det andra för att utföra statisk kodanalys. Båda verktygen används av Wordpress egna utvecklingsteam, så ett preliminärt stöd för verktygen fanns redan inbyggt i systemet. Det krävdes dock projektspecifika konfigurationer för att få allt att passa in i den virtualiserade utvecklingsomgivningen, baserat på Oracles Vm Virtualbox, som används i Aucor.

När omgivningen var klar för testning var det nästa steget att skriva testerna. Tester skrevs för instickprogrammets alla egenskaper, och eftersom källkoden fanns tillgänglig användes en strukturbaserad testteknik hellre än bara en speficikationsbaserad testteknik. Det omvandlade problemet från att kräva testning på alla möjliga indata till att kräva testning på alla möjliga stigar programmet kan ta. I egenskap med att instickprogrammet var klassbaserat bildade de olika egenskaperna relativt isolerade stigar, som inte var oöverkomligt svåra att testa.

Det sista steget var att konfigurera den externa integreringsservern, som knöt ihop allt till en komplett helhet. Servicen som användes var Travis CI, som övervakade en kopplad datakatalog och automatiskt skapade nya byggen då den upptäckte ändringar i datakatalogen. De projektspecifika konfigurationerna skrevs in i en speciell datafil, som servern använde för att återskapa den lokala utvecklingsomgivningen och som den sedan körde givna kommandon på. I Aucors fall var serverns huvudsakliga uppgift att köra testerna på den nya versionen av bygget.

Resultat

Ett vattentätt, kvantitativt utlåtande om hur mycket bättre utvecklingsprocessen blev i och med införandet av kontinuerlig integrering skulle ha krävt en större mängd data än vad som kunde samlas under avhandlingens gång. Frekvensen på utvecklingscyklerna för Aucor Core var helt enkelt inte tillräckligt hög. De data som lyckades insamlas antydde dock starkt på att kontinuerlig integrering och verktygen som valdes för utvecklingsprocessen bidrog till en snabbare, säkrare och smidigare mjukvaruutveckling.

Testningsfunktionaliteten som infördes i den lokala utvecklingsomgivningen möjliggjorde att testerna som skrevs täckte upp till 99,2% av insticksprogrammets egenskaper, vilket i praktiken betydde att totalt förblev bara tre stigar i egenskaperna otestade. Det betyder att när mjukvaran utvecklas vidare kan det i och med testerna så gott som garanteras att alla oavsiktliga förändringar i gammal funktionalitet fångas. Orsaken till att de tre stigarna förblev otestade var att de var otestbara, på grund av den interna strukturen hos Wordpress. Två av stigarna handlade om konstanter och hur tillstånden där konstanterna inte var definierade inte kunde nås under testerna. Den tredje stigen berörde omdirigering av användare och hur en sådan operation måste sluta med en funktion som avslutar kodkörningen (av säkerhetsskäl). Tyvärr avslutade funktionen också körningen av testkoden, som gjorde att stigen inte kunde testas.

Även om de långvarigare effekterna av testerna återstår att betraktas gav testerna i tre av egenskaperna omedelbara resultat. I alla tre egenskaper användes Wordpress egna inbyggda funktioner, som Wordpress smått hade ändrat på sedan de tagits i bruk. Ändringarna var inte tillräckligt betydliga att egenskaperna slutade fungera helt, men ändå såpass att utdatan inte var exakt den som var meningen. Det var inte förrän testerna skrevs och den förväntade utdatan uttryckligen definierades som felen märktes och kunde åtgärdas. Det bevisade ytterligare kraften hos tester och hur andra faktorer än fel som utvecklarna gör kan påverka hur mjukvaran fungerar.

Liksom testerna återstår integreringsserverns fulla potential att bli uppnådd när utvecklarna börjar använda den mer aktivt, men också den gav omedelbara resultat. Ett mycket kraftigt sätt att använda servern på är att köra bakåtkompatibilitettestning. Då försäkras det att insticksprogrammet fungerar även på webbsidor som använder en äldre version av Wordpress. Ett antal mindre potentiella risker fångades under testningen av de äldre Wordpress versionerna, som kunde åtgärdas innan de växte till allvarligare problem.

Sammanfattningsvis kan det konstateras att kontinuerlig integrering onekligen har en betydligare inverkan i större utvecklingsteam, där koden är mer utspridd och beroendena mellan delarna är mer komplexa, men att också mindre utvecklingsteam kan dra nytta av verktygen som praxisen erbjuder. För Aucor Core ersattes den osäkra och tidskrävande manuella testningen av automatiska och så när som heltäckande tester, som kan köras på några sekunder och försäkrar utvecklarna om att allt fungerar som det borde. Dessutom kopplades insticksprogrammets datakatalog till Travis CIs integreringsserver, som automatiskt kör testerna på ett antal Wordpress versioner och försäkrar att allting fungera även på äldre webbsidor. Servern fungerar också som en opartisk platform där koden från utomstående utvecklare kan granskas automatiskt lika kritiskt som den egna. Meningen var att öka på utvecklarnas förtroende och tillit till utvecklingsprocessen, vilket har bemöts med iver och planer om att utvidga systemet till andra processer. Verktygen hjälper mycket, men det som bör kommas ihåg är att de bara kan höja nivån på utvecklarnas arbete, inte göra det åt dem. Versionkontroll ersätter inte kommunikation, automation får inte utvecklare att integrera oftare och testning räddar inte usel kod. Det är grupparbetet, attityden hos utvecklarna och viljan att förbättra som praxisen förespråkar som producerar de bästa resultaten och som i sista hand gör kontinuerlig integrering till en så effektiv metodologi.

A Appendix

A.1 aucor-core.php

```php
<?php
/**
 * Plugin Name:    Aucor Core
 * Description:    The Aucor brand's core functionality
 * Version:        1.0.13
 * Author:         Aucor Oy
 * Author URI:     https://www.aucor.fi
 * Text Domain:    aucor-core
 */

// constant: plugin's root directory (used in some sub_features)
define('AUCOR_CORE_DIR', plugins_url('', __FILE__));

class Aucor_Core {


  // var: list of features in the plugin - array
  public $features;

  public function __construct() {

    /* Features
    ----------------------------------------------- */

    require_once 'abstract-feature.php';
    require_once 'features/admin/class-admin.php';
    require_once 'features/classic-editor/class-classic-editor.php';
    require_once 'features/dashboard/class-dashboard.php';
    require_once 'features/debug/class-debug.php';
    require_once 'features/front-end/class-front-end.php';
    require_once 'features/localization/class-localization.php';
    require_once 'features/plugins/class-plugins.php';
    require_once 'features/security/class-security.php';
    require_once 'features/speed/class-speed.php';

    /* Sub features
```

```
------------------------------------------------ */

require_once 'abstract-sub-feature.php';

// admin
require_once 'features/admin/sub_features/class-admin-gallery.php';
require_once 'features/admin/sub_features/class-admin-image-link.php';
require_once 'features/admin/sub_features/class-admin-login.php';
require_once 'features/admin/sub_features/class-admin-menu-cleanup.php';
require_once 'features/admin/sub_features/class-admin-notifications.php';
require_once 'features/admin/sub_features/class-admin-profile-cleanup.php';
require_once 'features/admin/sub_features/class-admin-remove-customizer.php';

// classic-editor
require_once 'features/classic-editor/sub_features/class-editor-tinymce.php';

// dashboard
require_once 'features/dashboard/sub_features/class-dashboard-cleanup.php';
require_once 'features/dashboard/sub_features/class-dashboard-recent-widget.php';
require_once 'features/dashboard/sub_features/class-dashboard-remove-panels.php';

// debug
require_once 'features/debug/sub_features/class-debug-style-guide.php';
require_once 'features/debug/sub_features/class-debug-wireframe.php';

// front-end
require_once 'features/front-end/sub_features/class-front-end-excerpt.php';
require_once 'features/front-end/sub_features/class-front-end-html-fixes.php';

// localization
require_once 'features/localization/sub_features/class-localization-polyfill.php'
    ;
require_once 'features/localization/sub_features/class-localization-string-
    translations.php';

// plugins
require_once 'features/plugins/sub_features/class-plugins-acf.php';
require_once 'features/plugins/sub_features/class-plugins-gravityforms.php';
require_once 'features/plugins/sub_features/class-plugins-redirection.php';
require_once 'features/plugins/sub_features/class-plugins-seo.php';
require_once 'features/plugins/sub_features/class-plugins-yoast.php';

// security
require_once 'features/security/sub_features/class-security-disable-admin-email-
    check.php';
require_once 'features/security/sub_features/class-security-disable-file-edit.php
    ';
require_once 'features/security/sub_features/class-security-disable-unfiltered-
    html.php';
require_once 'features/security/sub_features/class-security-head-cleanup.php';
require_once 'features/security/sub_features/class-security-hide-users.php';
```

```
    require_once 'features/security/sub_features/class-security-remove-comment-
        moderation.php';
    require_once 'features/security/sub_features/class-security-remove-commenting.php
        ';

    // speed
    require_once 'features/speed/sub_features/class-speed-limit-revisions.php';
    require_once 'features/speed/sub_features/class-speed-move-jquery.php';
    require_once 'features/speed/sub_features/class-speed-remove-emojis.php';
    require_once 'features/speed/sub_features/class-speed-remove-metabox.php';

    /* Helper functions
    ----------------------------------------------- */

    require_once 'helpers.php';

    /* Initialize features
    ----------------------------------------------- */

    $features = array(
      'aucor_core_admin'          => new Aucor_Core_Admin,
      'aucor_core_classic_editor' => new Aucor_Core_Classic_Editor,
      'aucor_core_dashboard'      => new Aucor_Core_Dashboard,
      'aucor_core_front_end'      => new Aucor_Core_Front_End,
      'aucor_core_localization'   => new Aucor_Core_Localization,
      'aucor_core_plugins'        => new Aucor_Core_Plugins,
      'aucor_core_security'       => new Aucor_Core_Security,
      'aucor_core_speed'          => new Aucor_Core_Speed,
      'aucor_core_debug'          => new Aucor_Core_Debug,
    );

  }

  /**
   * Get the features
   */
  public function get_features() {
    return $this->features;
  }

}

// init
add_action('plugins_loaded', function() {
  $aucor_core = new Aucor_Core;
});

// load translations
add_action('plugins_loaded', function () {
  load_plugin_textdomain( 'aucor-core', false, basename( dirname( __FILE__ ) ) . '/
      languages/' );
});
```

A.2 abstract-feature.php

```php
<?php
/**
 * Abstract Class Feature
 *
 * Structure and required functionality for each feature.
 * Every feature should inherit this class.
 */
abstract class Aucor_Core_Feature {

  /* ===================================================================
  01. Things to implement
  =================================================================== */

  // var: key - string
  private $key;

  // var: name - string
  private $name;

  // var: description - string, not used in version 1
  private $description;

  // var: is the feature active or inactive (disabled) - boolean
  private $is_active;

  // var: list of sub_features in the feature - array
  private $sub_features;

  // function: setup feature
  abstract public function setup();

  // function: init sub_features
  abstract public function sub_features_init();

  /* ===================================================================
  02. Things to inherit
  =================================================================== */

  /**
   * Construct
   */
  public function __construct() {
    $this->setup();
    if ($this->is_active()) {
      $this->sub_features_init();
    }
  }
  /**
   * Check if the feature is active
   *
   * @return bool
   */
  public function is_active() {
    return apply_filters($this->key, $this->is_active);
  }

  /**
   * Set function
   */
  public function set($key, $value) {
    if (property_exists($this, $key)) {
      $this->$key = $value;
    }
  }

  /**
   * Get functions
   */
  public function get_key() {
    return $this->key;
  }

  public function get_name() {
    return $this->name;
  }

  public function get_description() {
    return $this->description;
  }

  public function get_sub_features() {
    return $this->sub_features;
  }

}
```

A.3 abstract-subfeature.php

```
<?php
/**
 * Abstract Class Sub_Feature
 *
 * Structure and required functionality for each sub_feature.
 * Every sub_feature should inherit this class.
 */
abstract class Aucor_Core_Sub_Feature {

  /* ====================================================================
  01. Things to implement
  ==================================================================== */

  // var: key - string
  private $key;

  // var: name - string
  private $name;

  // var: description - string, not used in version 1
  private $description;

  // var: is the sub_feature active or inactive (disabled) - boolean
  private $is_active;

  // function: setup feature
  abstract public function setup();

  // function: run feature
  abstract public function run();

  /* ====================================================================
  02. Things to inherit
  ==================================================================== */

  /**
   * Construct
   */
  public function __construct() {
    $this->setup();
    if ($this->is_active()) {
      $this->run();
    }
  }

  /**
   * Check if feature is active
   *
   * @return bool
   */
  public function is_active() {
    return apply_filters($this->key, $this->is_active);
  }

  /**
   * Set function
   */
  public function set($key, $value) {
    if (property_exists($this, $key)) {
      $this->$key = $value;
    }
  }

  /**
   * Get functions
   */
  public function get_key() {
    return $this->key;
  }

  public function get_name() {
    return $this->name;
  }

  public function get_description() {
    return $this->description;
  }

}
```

A.4 class-admin-remove-customizer.php

```
<?php
/**
 * Class Admin_Remove_Customizer
 */
class Aucor_Core_Admin_Remove_Customizer extends Aucor_Core_Sub_Feature {

  public function setup() {

    // var: key
    $this->set('key', 'aucor_core_admin_remove_customizer');

    // var: name
    $this->set('name', 'Remove customizer from admin bar');

    // var: is_active
    $this->set('is_active', true);

  }

  /**
   * Run feature
   */
  public function run() {
    add_action('admin_bar_menu', array($this, 'aucor_core_remove_customizer_admin_bar
        '), 999);
  }

  /**
   * Remove customizer from admin bar
   *
   * @param WP_Admin_Bar $wp_admin_bar the admin bar
   */
  public static function aucor_core_remove_customizer_admin_bar($wp_admin_bar) {
    $wp_admin_bar->remove_menu('customize');
  }

}
```

A.5 class-localization-string-translations.php

```
<?php
/**
 * Class Localization_String_Translations
 */
class Aucor_Core_Localization_String_Translations extends Aucor_Core_Sub_Feature {

  public function setup() {

    // var: key
    $this->set('key', 'aucor_core_localization_string_translations');

    // var: name
    $this->set('name', 'Handling and registering site specific string translations');

    // var: is_active
    $this->set('is_active', true);

  }

  /**
   * Run feature
   */
  public function run() {
    add_action('init', array($this, 'aucor_core_string_registration'));
  }

  /**
   * String translations
   */
  public static function aucor_core_string_registration() {
    if (function_exists('pll_register_string')) {
      $group_name = get_bloginfo();
      $strings = apply_filters('aucor_core_pll_register_strings', array());
      foreach ($strings as $key => $value) {
        pll_register_string($key, $value, $group_name);
      }
    }
  }

}

/**
 * Get localized string by key
 *
 * @example ask__('Social share: Title')
 *
 * @param string $key unique identifier of string
 * @param string $lang 2-character language code (defaults to current language)
 *
 * @return string translated value or key if not registered string
 */
if (!function_exists('ask__')) {

  function ask__($key, $lang = null) {

    $strings = apply_filters('aucor_core_pll_register_strings', array());
    if (isset($strings[$key])) {
      if ($lang === null) {
        return pll__($strings[$key]);
      } else {
        return pll_translate_string($strings[$key], $lang);
      }
    }

    // debug missing strings
    aucor_core_debug_msg('Localization error - Missing string by key {' . $key . '}',
         array('ask__', 'ask_e'));

    return $key;

  }

}

/**
 * Echo localized string by key
 *
 * @param string $key unique identifier of string
 * @param string $lang 2 character language code (defaults to current language)
 */
if (!function_exists('ask_e')) {
  function ask_e($key, $lang = null) {
    echo ask__($key, $lang);
  }

}

/**
 * Get localized string by value
 *
 * @example asv__('Share on social media')
 *
 * @param string $value default value for string
 * @param string $lang 2 character language code (defaults to current language)
 *
 * @return string translated value or value if not registered string
 */
if (!function_exists('asv__')) {

  function asv__($value, $lang = null) {

    $strings = apply_filters('aucor_core_pll_register_strings', array());
    if (array_search($value, $strings)) {
      if ($lang === null) {
        return pll__($value);
      } else {
        return pll_translate_string($value, $lang);
      }
    }

    // debug missing strings
    aucor_core_debug_msg('Localization error - Missing string by value {' . $value .
        '}', array('asv__', 'asv_e'));

    return $value;
  }

}

/**
 * Echo localized string by value
 *
 * @param string $value default value for string
 * @param string $lang 2 character language code (defaults to current language)
 */
if (!function_exists('asv_e')) {
  function asv_e($value, $lang = null) {
    echo asv__($value, $lang);
  }
}
```

A.6 class-localization-polyfill.php

```
<?php
/**
 * Class Localization_Polyfill
 */
class Aucor_Core_Localization_Polyfill extends Aucor_Core_Sub_Feature {

  public function setup() {

    // var: key
    $this->set('key', 'aucor_core_localization_polyfill');

    // var: name
    $this->set('name', 'Preserve functionality without Polylang plugin');

    // var: is_active
    $this->set('is_active', true);

  }

  /**
   * Run feature
   */
  public function run() {

  }

}

/**
 * This structure of not having the polyfills in the class is so that the functions
 * might be used outside the class (i.e. the theme) without having to declare an
     instance
 * of the class first. The instance created below is to still maintain the option of
     diabling
 * the polyfills, if that need ever rises for some reason.
 */

/**
 * Get site locale
 *
 * @return string locale 2 character language code
 */
function aucor_core_get_site_locale() {
  $locale = get_locale();
  if (strlen($locale) >= 2) {
    return substr($locale, 0, 2);
  }
  // invalid locale
  return '';

}

$instance = new Aucor_Core_Localization_Polyfill;

/**
 * Fallback Polylang (preserve functionality without the plugin)
 */
if ($instance->is_active()) :
  if (!function_exists('pll__')) {
    function pll__($s) {
      return $s;
    }
  }
  if (!function_exists('pll_e')) {
    function pll_e($s) {
      echo $s;
    }
  }
  if (!function_exists('pll_esc_html__')) {
    function pll_esc_html__($s) {
      return esc_html($s);
    }
  }
  if (!function_exists('pll_esc_html_e')) {
    function pll_esc_html_e($s) {
      echo esc_html($s);
    }
  }
  if (!function_exists('pll_esc_attr__')) {
    function pll_esc_attr__($s) {
      return esc_attr($s);
    }
  }
  if (!function_exists('pll_esc_attr_e')) {
    function pll_esc_attr_e($s) {
      echo esc_attr($s);
    }
  }
  if (!function_exists('pll_current_language')) {
    function pll_current_language() {
      return aucor_core_get_site_locale();
    }
  }
  if (!function_exists('pll_get_post_language')) {
    function pll_get_post_language($id) {
      return aucor_core_get_site_locale();
    }
  }
  if (!function_exists('pll_get_post')) {
    function pll_get_post($post_id, $slug = '') {
      return $post_id;
    }
  }
  if (!function_exists('pll_get_term')) {
    function pll_get_term($term_id, $slug = '') {
      return $term_id;
    }
  }
  if (!function_exists('pll_translate_string')) {
    function pll_translate_string($str, $lang = '') {
      return $str;
    }
  }
  if (!function_exists('pll_home_url')) {
    function pll_home_url($slug = '') {
      return get_home_url();
    }
  }
endif;

unset($instance);
```

A.7 test-admin-remove-customizer.php

```
<?php
/**
 * Class AdminRemoveCustomizerTest
 *
 * @package Aucor_Core
 */

class AdminRemoveCustomizerTest extends WP_UnitTestCase {

  private $admin;

  public function setUp() {
    parent::setUp();
    $this->admin = new Aucor_Core_Admin;
  }

  public function tearDown() {
    unset($this->admin);
    parent::tearDown();
  }

  // test admin sub feature
```

```
public function test_admin_remove_customizer() {

  // needed to mock the admin bar
  require_once ABSPATH . WPINC . '/class-wp-admin-bar.php';

  $class = $this->admin->get_sub_features()['aucor_core_admin_remove_customizer'];
  // key
  $this->assertNotEmpty(
    $class->get_key()
  );
  // name
  $this->assertNotEmpty(
    $class->get_name()
  );
  // status
  $this->assertTrue(
    $class->is_active()
  );

  /**
   * Run
   */

  // check action hook
  $this->assertSame(
    999, has_action('admin_bar_menu', array($class, '
        aucor_core_remove_customizer_admin_bar'))
  );

  // AUCOR_CORE_REMOVE_CUSTOMIZER_ADMIN_BAR()

  // mock admin bar
  $args = new WP_Admin_Bar;
  $args->add_node(array(
      'id' => 'customize'
    )
  );
  // add extra item so the admin bar isn't empty when checking after removal
  $args->add_node(array(
      'id' => 'test'
    )
  );

  // run callback function
  $class->aucor_core_remove_customizer_admin_bar($args);

  // check that the node has been removed
  $this->assertArrayNotHasKey(
    'customize', $args->get_nodes()
  );
}
```

```
}
```

A.8 travis.yml

```
os: linux
dist: trusty

language: php

notifications:
  email:
    on_success: never
    on_failure: change

branches:
  only:
    - /.*/

cache:
  directories:
    - $HOME/.composer/cache

jobs:
  include:
    - php: 7.3
      env: WP_TRAVISCI=phpcs
    - php: 7.3
      env: WP_VERSION=latest
    - php: 7.3
      env: WP_VERSION=5.2
    - php: 7.3
      env: WP_VERSION=5.1
    - php: 7.3
      env: WP_VERSION=5.0
    - php: 7.2
      env: WP_VERSION=latest
    - php: 7.2
      env: WP_VERSION=5.2
    - php: 7.2
      env: WP_VERSION=5.1
    - php: 7.2
      env: WP_VERSION=5.0
    - php: 7.2
      env: WP_VERSION=4.9

before_script:
  - export PATH="$HOME/.composer/vendor/bin:$PATH"
  - |
    # Remove Xdebug for a huge performance increase:
    if [ -f ~/.phpenv/versions/$(phpenv version-name)/etc/conf.d/xdebug.ini ]; then
      phpenv config-rm xdebug.ini
    else
      echo "xdebug.ini does not exist"
    fi
  - |
    # Install WP test environment
    if [[ ! -z "$WP_VERSION" ]] ; then
      bash bin/install-wp-tests.sh wordpress_test root '' localhost $WP_VERSION
    fi
  - |
    # Install PHPUnit (version depending on WP version, older cores are incompatible
        with PHPUnit 7)
    if [[ "$WP_VERSION" == "5.0" || "$WP_VERSION" == "4.9" ]] ; then
      composer global require "phpunit/phpunit:^6"
    else
      composer global require "phpunit/phpunit:^7"
    fi
  - |
    if [[ "$WP_TRAVISCI" == "phpcs" ]] ; then
      composer global require "squizlabs/php_codesniffer:^3"
    fi
script:
  - |
    if [[ ! -z "$WP_VERSION" ]] ; then
      phpunit
    fi
  - |
    if [[ "$WP_TRAVISCI" == "phpcs" ]] ; then
      phpcs --extensions=php --ignore=*/vendor/*
    fi
```