

Adopting DevOps practices in a software project

Risto Karjalainen

Bachelor's Thesis

December 2021

Information and Communication Technologies

Bachelor's Degree Programme in Software Engineering

Author(s) Karjalainen, Risto	Type of publication Bachelor's Thesis	Date December 2021
	Number of pages 39	Language of publication English
		Permission for web publication: <input checked="" type="checkbox"/>
Title of publication Adopting DevOps practices in a software project		
Degree programme Bachelor's Degree Programme in Software Engineering		
Supervisor(s) Rantala, Ari		
Assigned by Bittium Wireless Ltd.		
<p>Abstract</p> <p>Bittium Wireless Ltd., the client of this thesis project, has a software project with multiple variants and target platforms. The project did not have a DevOps toolchain, and its development workflow relied heavily on manual processes and lacked visibility into code quality and build status. The goals of this thesis were to identify problems in the old workflow, to implement a new tool-based workflow in order to address the problems, and finally to evaluate the efficacy of the new workflow both quantitatively and qualitatively.</p> <p>To evaluate the success of the new toolchain, build success rates were examined of commits from before and after the adoption of the new toolchain. The study found that main-line builds were inconsistent for commits that were created before the adoption of the new toolchain, and 100% successful for commits that were created after the adoption of the new toolchain.</p> <p>Semi-structured interviews were conducted for project members who experienced both workflows. All interviewees were satisfied with the new toolchain, seeing it as a positive factor for code and software quality. Especially code review and continuous integration were seen as beneficial, while SonarQube's code analysis gained less adoption.</p>		
Keywords/tags devops, continuous integration, code review		
Miscellaneous -		

Content

Acronyms and terminology	3
1 Introduction	4
2 Goals and methods	5
3 DevOps practices	6
3.1 Code review	6
3.2 Continuous integration	7
3.3 Code analysis	8
3.4 Containerization	8
4 Implementation	9
4.1 Project description	9
4.2 Old workflow	10
4.3 Setting up the new toolchain	13
4.3.1 Installation	14
4.3.2 Preparing for CI	15
4.3.3 Code Review: Gerrit	20
4.3.4 CI: Jenkins	25
4.3.5 Quality Gate: SonarQube	28
4.4 Putting everything together: the new workflow	30
5 Results	32
5.1 Quantitative analysis	32
5.2 Interview results	35
6 Discussion	36
References	38
Appendices	39
Appendix 1. Build success rate data	39

List of Figures

Figure 1.	DevOps according to Ståhl et al. (2017)	4
Figure 2.	The AMS tool connects to Bittium devices	9
Figure 3.	The old development workflow	10
Figure 4.	The old testing workflow	11
Figure 5.	New toolchain servers	14
Figure 6.	The <code>./dev</code> script runs <code>build.sh</code> inside a container	18
Figure 7.	AMS tool version number scheme	20
Figure 8.	Jira link to issue AMS-899 in Gerrit commit message	21
Figure 9.	Gerrit reports status changes in Jira	21
Figure 10.	Inline comments in Gerrit	23
Figure 11.	A rejected commit (left) and an accepted commit (right)	24
Figure 12.	Submit button is enabled when all the gates pass	24
Figure 13.	Connecting a new agent to Jenkins	25
Figure 14.	A successful Jenkins commit gate	28
Figure 15.	A failed Jenkins commit gate	28
Figure 16.	Project overview in SonarQube	30
Figure 17.	Code quality issue shown in SonarQube	30
Figure 18.	New development workflow	31
Figure 19.	Build success rates	34

List of Tables

Table 1.	Profile of the interviewees	5
Table 2.	AMS tool variants and platforms	10
Table 3.	Mitigating problems in the old workflow	13
Table 4.	Breakdown of the AMS tool version number	20

Acronyms and terminology

AMS

Aftermarket Services

CI

Continuous Integration

I2C

A serial protocol for inter-device communication

IDE

Integrated Development Environment

IT

Information Technology

git

A version tracking and sharing tool for source code

OS

Operating System

TCP/IP

A suite of communication protocols used in computer networks

Qt

Cross-platform toolkit for creating graphical applications

SSH

A cryptographic network protocol

YAML

A data serialization language

1 Introduction

The word DevOps has generated a lot of buzz in the world of software development in recent times. DevOps is even seeping into other fields outside of IT, such as marketing (Vargo 2015). But the term DevOps is not without problems—there is a lack of consensus on how the word is defined, and on what counts and doesn't count as DevOps. Some sources state that DevOps exists to enable continuous practices, while other sources state the reverse: that continuous practices enable DevOps (Ståhl, Mårtensson & Bosch 2017). Yet other sources deny the definition of DevOps as a set of concrete tools and practices altogether, instead defining it as simply a culture of cooperation between organizations—that is, if you can cooperate, you can do DevOps (Vargo 2015). In this thesis, we shall use the term DevOps in the holistic definition proposed by Ståhl et al. (2017): DevOps is a combination of values, principles, methods, practices, and tools as pertains to software development (see Figure 1).

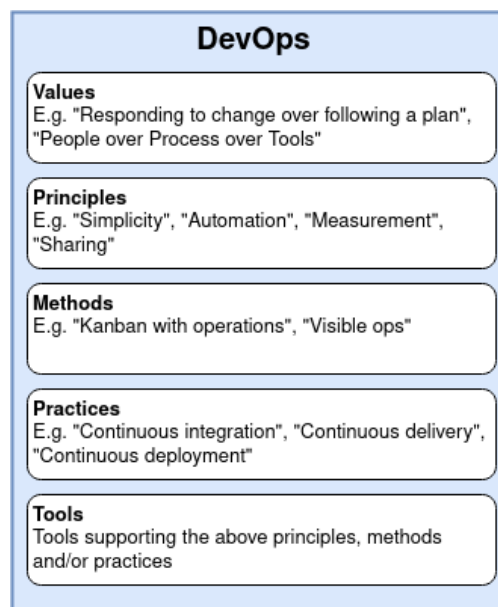


Figure 1: DevOps according to Ståhl et al. (2017)

Given this definition, how can DevOps philosophies and DevOps methodology be applied to improve the workflows in a real life software project? What effects does cultivating “DevOps culture” in a project have? How would you even go about it? These questions are central to this thesis project and we aim to first apply DevOps methodology (with the main focus on DevOps practices and tools) in a real-world software project, then seek to answer the question of what effects—positive or negative—the changes had.

The client of the thesis project and the owner of the targeted software project is Bittium Wireless Ltd. Bittium is a Finnish company of 673 employees (in 2020), which specializes in the development of secure communications and connectivity. Bittium also provides health-care technology products and services in cardiology, neurology, rehabilitation, occupational health and sports medicine. (Bittium 2021)

2 Goals and methods

Bittium has a preexisting software project (described in more detail in chapter 4), which was in need of workflow improvements. The goals of this thesis project were to:

- identify problems in the old workflow,
- address the problems with concrete workflow changes,
- and finally to evaluate the outcome of the workflow changes.

The workflow before the practical implementation phase of this thesis shall be referred to as the “old workflow”, and the workflow after the improvements were implemented shall be referred to as the “new workflow” in this thesis.

Chapter 3 lays out the theoretical basis of the software engineering practices that are used in this thesis work and that are commonly associated with DevOps.

Chapter 4 describes the practical application of the aforementioned theoretical concepts in a real-world software project. The chapter starts by describing the old workflow, then moves on to describing the implementation details of workflow changes and the reasoning behind each change, and finishes by describing the completed new workflow.

In chapter 5, the efficacy of the workflow changes is evaluated quantitatively with an interview with project members, and qualitatively by examining build success rates.

The qualitative analysis consists of a semi-structured interview was held for three project members. The interviews were conducted after the workflow improvements had been implemented and the interviewees had had sufficient time to adjust to the new workflows. The purpose of the interview was to collect impressions of both the old workflow and the new workflow, and to compare the two from multiple different perspectives. The interview structure was kept very loose in order to promote free discussion and maximize novel insights from the participants.

Table 1: Profile of the interviewees

P#	Role	Experience
P1	Architect	20+ years
P2	Developer	15 years
P3	Tester	5 years

The interview was structured chronologically:

- **Introduction.** Purpose of the interview. How the data will be handled and used.
- **Exploring the old workflow.** Exploration of how the interviewees remember the old workflows. What comes to mind first? Specific anecdotes? Finding both good and bad impressions without leading questions. Specific topics to bring up if necessary: building, releasing, testing.

- **Exploring the new workflow.** What comes to mind first when thinking of the new workflow? What is it like to work with the new practices in place? Specific anecdotes. Specific topics to bring up if necessary: code review, build automation, continuous integration, test automation, documentation generation, static analysis, build scripts.
- **Future work.** What would be your vision of the “perfect” workflow? How could we change the new workflow to get closer to that vision? Free discussion.

The interviews were conducted remotely by means of Microsoft Teams and they lasted between 30 minutes and 60 minutes. The interviews were recorded and then transcribed. The data from the interviews is handled anonymously in this thesis, and the recordings were deleted after transcription.

3 DevOps practices

In this chapter, we will examine the theoretical basis for this thesis project. As mentioned in the introduction, DevOps is a collection of values, principles, and methods. We will especially focus on DevOps practices and methods that can be directly applied in practice to improve workflows in a software project. The main practices that will form the backbone of the new workflow are code review, continuous integration, code analysis, and containerization. Each practice will be explained in more detail in their respective sections.

3.1 Code review

Code review is the practice of inspecting program code for problems such as logic errors and problems with design. Reviewing code does not necessarily require any tools, as you can simply open the source code in the medium of your choice and start reading it. Modern code review, however, is commonly associated with tools and techniques that aim to make reviewing code more convenient, and to embed code review as a necessary step before new code is accepted into the mainline. (Bacchelli & Bird 2013)

Modern tools for code review implement convenient features such as blocking patches until they have been approved by a reviewer, discussing the patch in threaded discussions, and leaving inlined review comments directly to specific lines of source code. The typical code review consists of a developer first submitting a patch for review, then other developers inspecting the patch and either leaving comments and questions or simply approving the commit if it has no problems. The patch submitter looks through the comments and questions and makes the necessary fixes to the patch. This submit–comment–fix sequence is repeated until every reviewer has approved the patch, and only then it can be merged to the mainline.

Many potential benefits are attributed to code review (Bacchelli & Bird 2013):

- **Improved software quality.** Many pairs of eyes looking at the code see more than just one pair, which helps to catch bugs earlier. However, a case study at Google described bug finding as “welcome but not the main focus” (Sadowski, Söderberg, Church, Sipko & Bacchelli 2018).
- **Design problems are caught earlier.** Problems with code structure and code design could have far-reaching impacts on code maintainability. Other developers reading the code at an early stage is a direct way to gauge how readable the source code is. Bad readability, or a wrong abstraction or clumsy design could cause problems far down the line.
- **Lateral transfer of skills.** Developers viewing and discussing each others’ code regularly is an opportunity for transfer of skill.
- **Reduced siloing.** Other developers viewing new code before it is merged is a good way to transfer knowledge about that code (Bacchelli & Bird 2013). The scenario is avoided where a solo developer submits patch after patch directly to the mainline, with no other developer knowing how their code works or how to maintain it.

3.2 Continuous integration

The term continuous integration (CI) is similarly vaguely defined as the word DevOps (Ståhl & Bosch 2014), but in this thesis we take it to mean the practice of integrating new changes into the mainline as often as possible, preferably with automated builds. Ideally, every new commit should be built and tested immediately as it is merged to the mainline. Compared to a model where changes are accumulated over a long time and integrated as a separate integration phase, the practice of CI comes with multiple benefits (Fowler & Foemmel 2006; Duvall, Matyas, S. & Glover, A. 2007; Miller 2008):

- Reduces integration pains later down the line. Integrating infrequently comes with significant risk, as the longer development continues in a separate branch, the harder and more unpredictable integration becomes.
- Increases visibility into the state of the build. Because the newest state of the mainline is built automatically, the state of the build is constantly known and transparent. The mainline won’t be broken without anyone knowing about it.
- Consistent build environments. The “works on my machine” problem is avoided—i.e. when a change works on a developer’s machine but doesn’t work on another machine. CI always builds the project in a known and consistent environment and reports if the build fails.
- Developers get immediate feedback when submitting new changes. If a developer breaks the build, the CI system will notify them. The CI system will also indicate which commit it was that broke the build, so that either fixing the commit or reverting it is easier.
- Increases visibility into the state of the project for managers, product owners, and other stakeholders.
- Reduces repetitive manual processes by automating them.
- Improves software quality. Miller’s case study at Microsoft (2008) estimates that the cost of reaching the same level of quality by manual processes would be at least 40% higher than with CI.
- A working build of the latest mainline is always available for testing and demos.

There are various way to trigger CI builds. The CI system could be polling the source code

repository for change and initiate a new build when new changes are detected. Builds can also be scheduled, for example scheduling longer builds for every night. Some version control software also support triggering the build from the version control when a new commit is merged. For CI, either polling or triggered builds should be used. (Duvall et al. 2007)

The process of CI can be explained with an example. Let's say a developer has just finished implementing a new change. Without CI, they would simply push the new change directly into the shared version control system. If they make any mistakes, or if their own development machine has some configuration that makes the build work for them but not for others, then their new change might break the mainline. With CI, however, when the build is pushed to version control, a CI build is immediately started. When the build finishes, the build status is reported to the developer and the rest of the team. If the mainline was broken, now the development team knows about it and they can avoid checking out the breaking changes until the problem has been addressed.

3.3 Code analysis

The documentation for LLVM's Clang Static Analyzer defines static analysis as "a collection of algorithms and techniques used to analyze source code in order to automatically find bugs" (Clang Static Analyzer 2021). It is a means to automate parts of code inspection, style checks, and error finding.

Some benefits of static analysis are (Clang Static Analyzer 2021):

- Catch potential bugs earlier. Some bugs that might have otherwise escaped detection until the software is in production are found before the faulty code is committed to the mainline.
- Better visibility into code quality through indicators and metrics.

Static analysis also comes with potential problems (Clang Static Analyzer 2021):

- Static analysis takes time, so builds take longer.
- False positives. If there are too many false positives, finding the actual problems hiding among them is no longer cost-effective.

Code analysis can be combined with continuous integration, such that each CI build includes a static analysis step. This way the code quality of the mainline can be traced at all times. Developers, architects and tech leads, project managers, and other stakeholders always have access to up-to-date code analysis information that can be used to make informed decisions about how the project should allocate its resources. (Duvall et al. 2007)

3.4 Containerization

Containerization (also known as OS-level virtualization) is a technology that allows bundling applications and their dependencies into one package, and then running the packaged application in an isolated environment (also called a "sandbox"). These packages are called

“containers”. Containerization is distinct from virtualization, as containers share the host system’s kernel and use its capabilities to isolate themselves. (Hogg 2014)

Packaging dependencies inside the container enables running the application in a consistent environment across different machines. The host machine is kept clean, as there is no need to install dependencies directly on the host machine. Deployment is also simplified, as dependencies come with the application instead of having to be installed manually. Containerized applications can be made modular, so that it is possible to further combine these containers into larger multi-container applications. Some examples of containerization technologies are Docker, LXC, Solaris containers. (Hogg 2014)

4 Implementation

4.1 Project description

The targeted project is a desktop application that is used for aftermarket service and diagnostics of Bittium devices (henceforth referred to as “the AMS tool” or “the application” in this thesis). The application detects when a Bittium device is connected to the computer and allows the user to perform various actions on the connected device, such as factory reset, software update, diagnostics, and so on (see Figure 2). The application supports multiple protocols for connecting to devices, such as TCP/IP, I2C, and so on. Which protocol is chosen depends on the type of the connected device. The application also connects to a web service for a number of purposes, such as authentication, device data, and service logs.

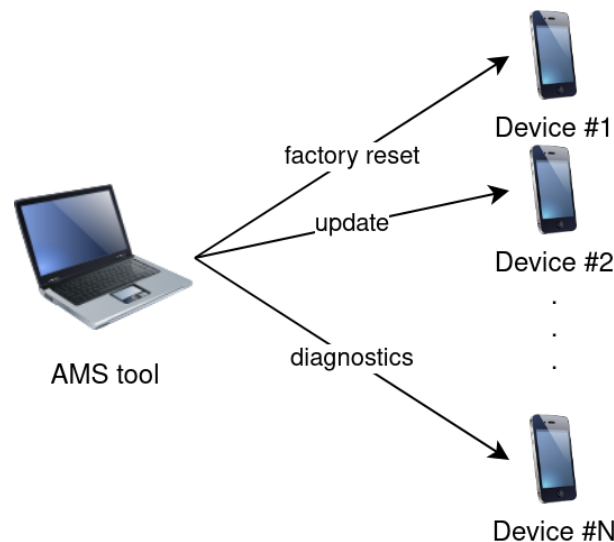


Figure 2: The AMS tool connects to Bittium devices

The AMS tool is a native desktop application written in C++ and Qt. The application targets multiple platforms, the main targets being Ubuntu 16.04, Debian 10, and Debian 11. The application also has multiple variants that are all built from the same codebase. Each variant has different feature sets that are enabled or disabled based on compilation options. Table 2

shows a list of all supported variant and platform combinations.

Table 2: AMS tool variants and platforms

	Debian 10	Debian 11	Ubuntu 16.04
Helpdesk variant			✓
Internal variant			✓
Commissioning variant	✓	✓	

4.2 Old workflow

The old workflow for developing the AMS tool relied heavily on manual processes (see Figure 3). All development, building, testing, integration, and even releasing was performed manually by developers on their personal development machines. Development and running tests were done using Qt Creator (an IDE for C++ / Qt development), and each developer used their personal choice of tools for managing their local git repository.

Code was shared among developers by using a shared remote git repository (called the “mainline”), which had a web front-end called Gitorious that was used for access control and browsing the source code through a web interface. Gitorious also supports merge requests and code reviews, but neither feature was leveraged in the old workflow.

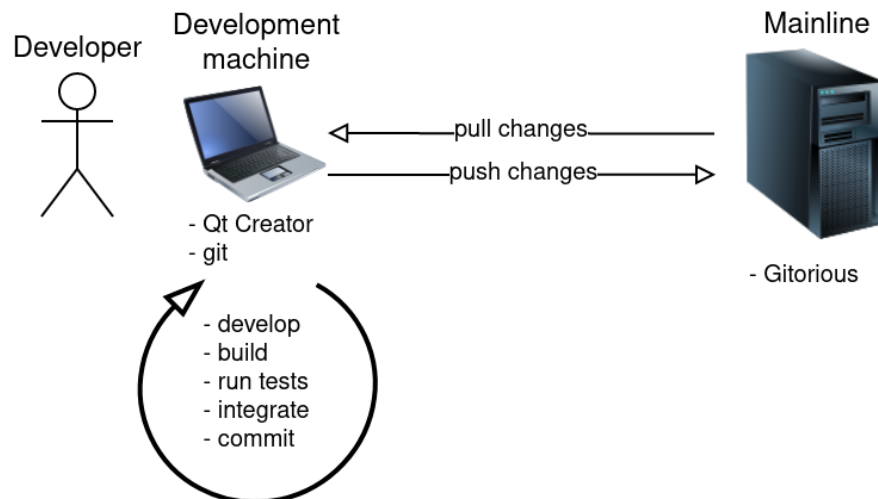


Figure 3: The old development workflow

In the old workflow, testers received latest builds of the application directly from developers, who built it on their personal machines (see Figure 4). The artifacts were transferred typically on USB flash drives and there was no automatic tracking of what exact build and content was delivered to testing. Versioning and content tracking were left up to the developers and testers to keep track of manually, a process that was prone to errors and confusion.

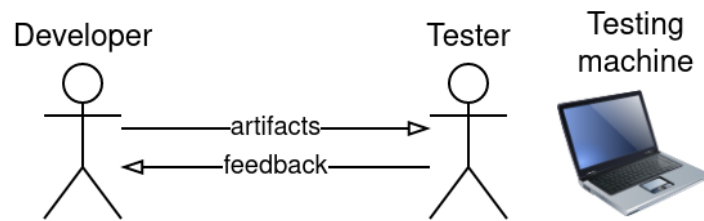


Figure 4: The old testing workflow

In our estimation, the old workflow had three major problems: too much reliance on manual processes, inconsistent mainline, and lack of information. Each of these three problems also brought with them a host of secondary problems, which we will elaborate further in this chapter.

Problem 1: Reliance on manual processes

Manual processes are tedious and leave a lot of room for human errors and inaccuracies. In the old workflow, release builds were made manually by developers and delivered to testers on USB flash memory. If a mistake was made during the build—such as building the wrong revision, building the wrong content, accidentally making the wrong kind of build, forgetting to regenerate localization files, etc.—it wasted a lot of developer time. And the longer the problem went unnoticed, the more time was wasted.

Tracking releases was also done manually. The USB flash memory, which was used for delivering software to testers, quickly became cluttered with different variations of the same files, and it was up to the developers and testers to manually tag them so that it was possible to keep track of which files belonged to which release. This meant that testers had to track manually what versions and what content they were testing—insofar as it was even possible to know with certainty what content was being tested, due to the build itself being the result of an error-prone manual process.

Inconsistent build environment were yet another pain point. Each developer developed on their personal machines, and each developer had customized their system to fit their personal preferences. The version of Qt was decided at the project level, but everything else on the system (OS library versions, compiler versions, etc.) was potentially different. This was a cause for a lot of mysterious problems when using builds from different developers.

Due to heavy emphasis on manual processes, integration also became very difficult and time-consuming. Big features could spend weeks or even months in separate branches before they were integrated to the mainline. Testing of new features would be delayed until they were merged. This often meant a long and arduous integration phase, where developers and testers would spend weeks or more hunting bugs and trying to get all the features to work together. From a tester's point of view, this could mean a long period of inactivity during which there was barely anything to test, and then a sudden surge of overactivity when a feature was merged.

Problem 2: Inconsistent mainline

In the old workflow, commits were pushed from the developers machine directly to the mainline with no quality gating. There were no quality or style checks that would block a bad commit from being merged or, at the very least, inform the team of the bad commit. Design problems and architectural problems, as well as obvious logic errors and code that wouldn't even compile, regularly made their way into the mainline without hindrance.

Other than design problems or bugs, quality problems can also arise from simple human errors. Even the best developer makes mistakes, such as failing to include all the local changes in the commit, or including too much. Such mistakes could mean that even though the project seems to work on your personal machine, pushing the incomplete commit to the mainline would break it. The mistake would be noticed only when the next developer pulls the changes and tries to compile them.

Another common human error was accidentally including temporary files or debug code in a commit. In one case, a developer unintentionally committed a 200 megabyte ZIP-file, which caused a clean-up operation that forced everyone on the team to update their local repositories (rewriting the history on the mainline means that everyone has to update their local repositories to match the amended mainline).

Whether it was a human errors or tool errors, or any other category of error, the lack of quality gating made blocking and detecting them much harder, which lead to losing time to debugging sudden build problems, and a an overall reduction in code quality and software quality.

Problem 3: Lack of information

The old workflow lacked transparency into the state of the mainline (also relates to Problem 1 and Problem 2). At any given point in time, the mainline might not pass all the tests (if it even compiled) and no one would know about it. This was far from cost-effective, as it meant that developers had to spend time troubleshooting the build itself, often unsure what broke it. The lack of automatic static analysis meant that there were no code quality metrics easily available either, so the team was unable to make informed decisions about what should be prioritized.

The lack of a formalized code-review process was also a problem: because no one got the opportunity to see new code before it was pushed to the mainline, there was noticable siloing of knowledge within the team. Over-the-shoulder code reviews were sometimes performed, but they were an exception rather than a rule. Each team member had their own components that they developed, and they rarely ventured outside their own territory to see what was going on elsewhere in the codebase. This not only caused developers to have less knowledge about the codebase than they could, but it also missed a crucial opportunity to share general programming knowledge among the team.

4.3 Setting up the new toolchain

In order to address the problems in the workflow, we constructed a toolchain. Each of the tools that we selected mitigates at least one of the primary problems that we identified in chapter 4.2. The focus was especially on automating manual workflows and bringing more visibility into the status of the mainline.

The approach we took consisted of three parts, each of which directly addressed at least one problem in the old workflow:

- **Automation** mitigates Problem 1: Reliance on manual processes
- **Quality gating** mitigates Problem 2: Inconsistent mainline
- **Improved visibility** mitigates Problem 3: Lack of information

Following this three-point strategy, we used the practices introduced in chapter 3 to find and implement concrete improvements. The practices were chosen based on their theoretical benefits, and each practice answered different needs (see Table 3)—code review and static analysis to form the quality gate, and CI as the glue that ties everything together. In the spirit of DevOps, every practice came with the promise of improved visibility through automated feedback mechanisms.

Table 3: Mitigating problems in the old workflow

	Automation	Quality gating	Improved visibility
Code review		✓	✓
Continuous integration	✓	✓	✓
Static analysis		✓	✓

The tools that we selected for implementing the practices in this project are:

- **Gerrit** for code review. Gerrit is a web-based tool for conducting code reviews. It allows teams to view and discuss incoming new code, and to accept or reject new changes.
- **Jenkins** for continuous integration. Jenkins is a web-based tool for automation of various tasks, such as building and testing software.
- **SonarQube** for static analysis and quality checks. SonarQube is a web-based tool for inspecting code quality and creating so called “quality gates”, i.e. automated checks for source code that must be passed before the code can be merged.

The biggest factor in the tool selection was the level of familiarity at Bittium and especially among project members. A small survey among the project members revealed that many of them had already used or at least heard of Gerrit and Jenkins before. We also considered GitLab, which supports both code review and continuous integration, but it was a less-known tool at Bittium at the time so we chose not to pioneer it in this project, instead going with the more familiar option to minimize risk and to maximize value.

To host the new tools, we provisioned four server machines `server01`, `server02`, `server03`, and `server04` (see Figure 5), one server for each primary tool, and one dedicated machine for the Jenkins agent for running builds. A basic installation of the tools was done using Salt (described in section 4.3.1), and the configuration for each tool is described in their respective sections after that.

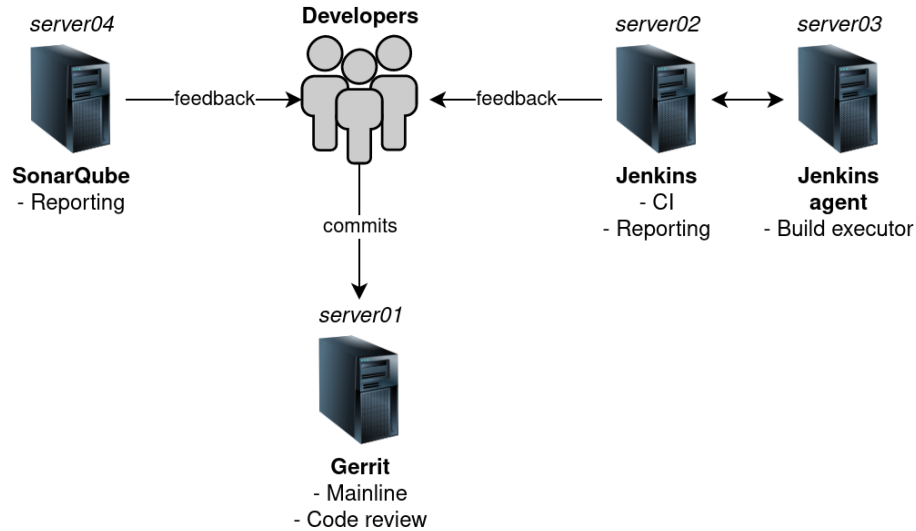


Figure 5: New toolchain servers

4.3.1 Installation

To install the primary tools for the new toolchain, we used a tool called Salt. Salt (also known as SaltStack) is a tool for remote task execution and configuration management (Salt (software) 2021).

Configuration management is a huge topic in its own right, and out of scope for this thesis, so we will skip most of the details of installing Salt and creating Salt states, but we were able to use Salt to automate the tedious parts of making a bare-bones installation of the tools needed in this project. Our utilization of Salt was limited to only installing the tools, and configuration and maintenance were left to be done manually.

Salt uses configuration files called “Salt states” for defining how each tool is installed. We were able to source the Salt states for Gerrit, Jenkins and SonarQube from another project within the company, so it was a natural choice to use them rather than spend time writing our own state files.

The main Salt state file, which is named `top.sls` (called the “Top file” in Salt terminology), is a YAML file that contains the information on what states to install on which server. For this project, the Top file looks as follows:


```

base:
  '*':
    - tools
  'server01':
    - gerrit
    - apache.gerrit
    - backup
  'server02':
    - jenkins
    - apache.jenkins
    - backup
  'server03':
    - docker
    - docker.compose
    - jenkins-agent
  'server04':
    - sonar
    - apache.sonar

```

The YAML-snippet installs the basic tools that are needed for server maintenance (such as editors) on all the four servers (states for the wildcard '*' apply to all Salt minions). Then it installs Gerrit on `server01`, Jenkins on `server02`, Jenkins agent on `server03`, and SonarQube on `server04`. Additionally, an Apache reverse proxy is installed and configured on all the servers that have outward facing services. The purpose of the reverse proxy is to provide TLS termination for additional security. Servers that contain important data such as source code and build artifacts (`server01` and `server02` respectively) also get a backup script, which makes regular backups of important data such as source code and build artifacts.

The Jenkins build agent only requires a Docker installation and no other dependencies (for reasons that are further elaborated in section 4.3.2, where we discuss the implementation of the build scripts for CI).

With the completed Top file, next we only needed to apply the Salt states to all the machines by running the following command from the Salt master:

```
$ salt '*' state.apply
```

After the command finished successfully, we now had working installations for Gerrit, Jenkins, and SonarQube, and we could begin configuring them.

4.3.2 Preparing for CI

Before configuring the new toolchain—especially CI—first we had to do some preparatory work.

Firstly, the build system needed a way to run builds without an IDE. Developers used Qt Creator to develop and build the application, but running a graphical tool from the CI system

was not an option. A more appropriate solution was to develop a script that could be run from the command line and which would be easier to integrate with a CI system.

Because the AMS tool targets multiple platforms, the build script needed to support building for different target platforms, preferably from the same machine. We could have provisioned separate Jenkins agent machines (either virtual or bare metal) for each of the targeted platforms, but that would've precluded the possibility of building for all platforms from the same machine (for example a developer's machine) and would've been less cost-effective due to increased server costs.

Furthermore, the AMS tool had no version information. As such, if you were handed the AMS tool without knowing what version it was, it would have been very difficult or impossible to find out which exact build it was. Versions had to be tracked manually, which was both inconvenient and prone to error. We needed to devise a way to embed version information directly into the application so that even if all you had was the binary file, it would still be possible to trace it back to the exact commit and build number that it came from.

Build script for multiple target platforms

To be able to run builds from CI and to be able to target multiple platforms from one machine, we developed a build script that utilizes containerization technology.

Options that we considered for building for different platforms (but the same architecture) were chroot, virtual machines, and finally containers. All three were viable options, but we chose containerization as it was deemed the least complicated way of the three to get running. All of the aforementioned methods can be automated, but creating a chrooted build environment would've be much more involved compared to simply spinning up a container. Virtual machines would've be too heavy to spin up and down at will. Therefore for the speed and convenience we chose containerization, and specifically Docker.

The build script that we developed supports building the project, running unit tests, running static analysis, and various other functionalities. All supported functions are performed inside Docker containers. Based on the theoretical examination of containerization in section 3.4, Dockerizing the build process comes with clear benefits:

- Dependencies for each target platform are now encapsulated inside Docker images instead of being installed directly on the host machine. The host machine only needs working installations of Docker and Bash to run builds. Bash is default shell on most Linux distributions and therefore likely to be available by default, so only Docker needs to be installed by the operator.
- All the installed dependencies can be tightly controlled, eliminating problems caused by inconsistent build environments on different machines. New dependencies can not creep in silently, because they are clearly and unambiguously defined and version controlled in Docker configuration files. Build environments remain known and consistent, as changes to the host machine do not affect builds.

We built a “master script” that starts a container and runs a build script inside it. The master script—which we named the `dev` script—was built upon the following simple idea:

```
docker run \  
  --rm \  
  -v "$PWD:/workarea" \  
  -w "/workarea" \  
  "$BUILDENV_IMAGE" \  
  ./build.sh $*
```

This command does several things for us. The most important part is that it starts a new Docker container from the Docker image specified by the variable `$BUILDENV_IMAGE`, which we call the “build environment”. The build environment could be for example `buildenv-ubuntu-xenial` if we wanted to build for Ubuntu 16.04 (xenial), or `buildenv-debian-buster` if we wanted to build for Debian 10 (buster), and so on. This simple idea is very powerful, as it means that any build environment can be used by simply changing the value of the `$BUILDENV_IMAGE` variable, as long as a docker image with the given name exists.

The last line causes a call to another script called `build.sh` inside the container, which is the main build script that itself doesn’t know about Docker or containers. It simply runs in whatever environment the `dev` script places it in.

The option `-v $PWD:/workarea` mounts the current directory inside the container in the directory `/workarea`. If the command is run from the source code repository root, that means that the source code becomes available at `/workarea` in the newly started container. `-w /workarea` sets the working directory inside the container to `/workarea`.

Lastly, `--rm` causes Docker to delete the container immediately upon exit. The containers themselves do not contain any important data—everything import is in the `/workarea` directory that resides on the host system’s filesystem—so instead of letting old containers clutter the system, we can delete them after one use.

The `dev` script ended up growing much bigger than just the one `docker run` command, but everything else that it does is simply ease-of-use and convenience features, with the gist of the script always remaining the same: spin up a build environment and run `build.sh` inside it.

The finished implementation is a build script system that consists of three main components (see Figure 6):

- `dev`, or the “master script”, is the front-end bash script that developers call directly. It starts up a containerized environment and runs `build.sh` inside it. The supported environments are Ubuntu 16.04, Debian 10, and Debian 11. Any of the supported build environments can be selected with a command line argument, or if no build environment is provided, the default environment (Ubuntu 16.04) is used.
- `build.sh` is a bash script that contains the logic for building the application and other build-related tasks, which are divided into subcommands such as `build`, `test`, `clean`, `cppcheck`, and so on. Developers do not call `build.sh` directly—instead, it’s the `dev` script that calls it in one of the supported build environment containers.
- `buildenvs/` contains the supported build environments in subdirectories. Each build environment has their own Docker configuration, which contains information about what build tools and dependencies are installed for the specific build environment, and how they are installed. Adding a new build environment is as simple as creating a new Docker configuration file inside the `buildenvs/` directory.

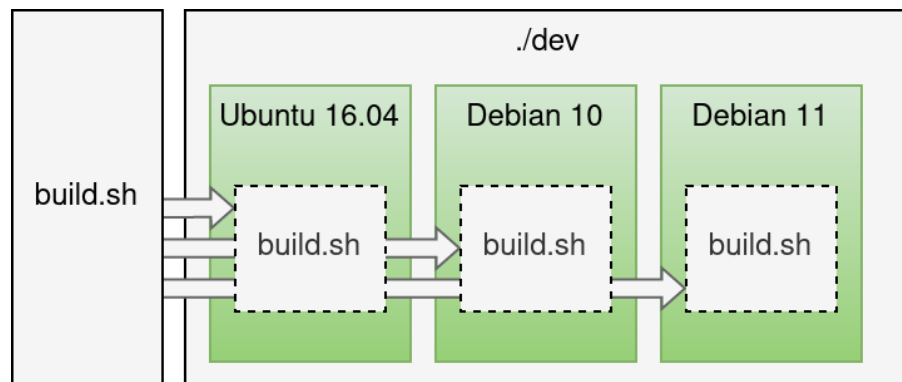


Figure 6: The `./dev` script runs `build.sh` inside a container

The finished build script can be used by both developers and the CI pipeline alike. The command line interface for the `dev` script looks as follows:

```
USAGE: dev [OPTIONS] ARGS
```

```
Runs the build script (tools/scripts/build.sh) in a docker
container that has all the necessary tools and libraries for
building the project
```

```
OPTIONS:
```

```
-e    Choose build platform (default 'ubuntu-xenial')
-l    List available build platforms and exit
```

```
ARGS are passed to the build script as-is. See build script
help for more information on build script arguments.
```

An example usage would be:

```
./dev -e buildenv-debian-buster build helpdesk release
```

Calling the command results in the following sequence:

1. `dev` starts up a container from a Docker image called `buildenv-debian-buster` and mounts the source code directory inside the new container
2. `build.sh build helpdesk release` is called inside the newly created container, which starts a release build for the `helpdesk` variant.
3. After the build finishes, the container is deleted. If the build succeeded, a release build for Debian 10 (buster) is placed on the disk.

The `build.sh` script itself is a normal bash script that supports various basic operations like `build` for building the project, `clean` for removing temporary build files, `test` for running unit tests, `coverage` for generating a test coverage result, and so on.

The build environments are defined in Dockerfiles, each of which set up the build environment that is capable of building the targeted variants for that specific platform. An example dockerfile for Debian 10 (buster) could be as simple as:

```
FROM debian:buster
RUN apt-get update && \
    apt-get install -y \
        build-essential \
        git-core \
        qt5-default \
        qt5-qmake \
        qt5-qmake-bin \
        libqt5core5a \
        ccache \
        gcovr
```

The example Dockerfile creates a new Docker image that is based on a publicly available `debian:buster` image, which, as the name implies, is a Debian 10 (buster) environment. Then it installs the build dependencies with Debian's default package manager `apt`. With the Dockerfile create, the build environment image is built with the command:

```
docker build -t buildenv-debian-buster .
```

After the command finishes, the image should be visible in the system under the name `buildenv-debian-buster`, and the newly created build environment is ready to be used from the `dev` script with the `-e buildenv-debian-buster` option.

Versioning

To improve the traceability of application binaries to their respective releases, commits, and build numbers, we devised a versioning scheme for the AMS tool. The version number consists of multiple components. Figure 7 shows an example of a valid version number and lists

its components, with detailed descriptions for each component in Table 4.

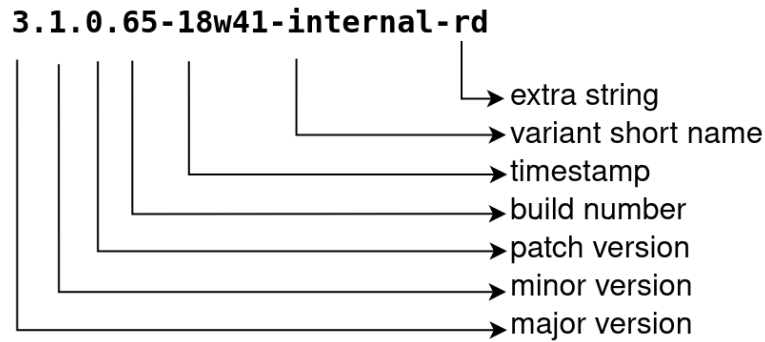


Figure 7: AMS tool version number scheme

Table 4: Breakdown of the AMS tool version number

Component	Description
Major version	Variant number. 2 for Helpdesk variant, 3 for Internal variant, and 5 for Commissioning variant.
Minor version	Increases after implementing new features
Patch version	Increases after implementing error fixes
Build number	Build number in CI, 0 for non CI builds
Timestamp	Build year and week number. Formatted as “YYwWW”, where YY is the years as two numbers, followed by the letter ‘w’, and finally WW is the week number. E.g. “17w31” would mean the year 2017, week 31.
Variant short name	A short name for the variant: “helpdesk”, “internal” or “commissioning”
Extra string	An extra string that tells what kind of a build the binary came from. “testing” for commit gate builds, “rd” for mainline builds, “release” for release builds.

After implementing the versioning scheme, we now have the ability to check the binary version number and other relevant information from the binary file itself:

```
$ amstool --version
Internal 3.1.1.77-18w43-internal-testing
Build time: 2018-10-23T06:51:07
Commit: 0549797
```

4.3.3 Code Review: Gerrit

After installing Gerrit in chapter 4.3.1, we now had Gerrit running but it still needed to be configured. We skip the details of domain integration here, but after it was completed, we had administrator access to the Gerrit instance and were able to start configuring it.

Gerrit-Jira integration

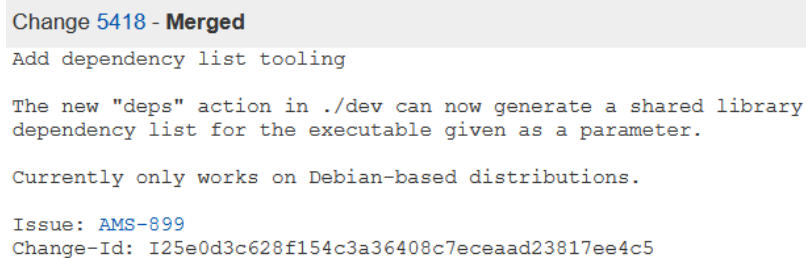
We used the `its-jira` plugin for integrating Gerrit with Jira, which the project uses for issue tracking. First we enabled the `its-jira` plugin by editing the Gerrit configuration file `etc/gerrit.config` and inserting the following configuration:

```
[plugin "its-jira"]
  enabled = true
```

Next we enabled Jira comment links from the `its-jira` plugin. The comment links feature creates clickable links from commit message to Jira issues when they are mentioned (see Figure 8). The feature was enabled by adding the following configuration to `gerrit.config`:

```
[commentLink "its-jira"]
  match = ([A-Z0-9]+--[0-9]+)
  html = <a href=\"https://example.com/jira/browse/$1\">$1</a>
  association = SUGGESTED
```

The `match` option is a regular expression that causes anything of the form `ABC-123` in commit messages to be interpreted as Jira issues. The `association` option defines whether all commit messages must contain a link to a Jira issue or not. We chose the value `SUGGESTED`, which does not enforce Jira issue mentions, but notifies if the commit message is missing a Jira issue mention. This was mostly a project policy decision—while some projects prefer that all changes reference a Jira issue for traceability reasons, the AMS tool project opted for a softer landing by encouraging Jira issue references but not enforcing them.



```
Change 5418 - Merged
Add dependency list tooling

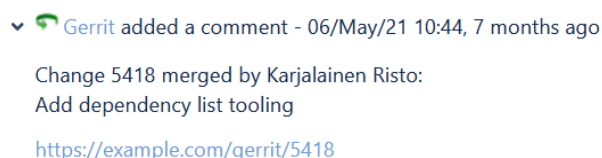
The new "deps" action in ./dev can now generate a shared library
dependency list for the executable given as a parameter.

Currently only works on Debian-based distributions.

Issue: AMS-899
Change-Id: I25e0d3c628f154c3a36408c7eceaad23817ee4c5
```

Figure 8: Jira link to issue AMS-899 in Gerrit commit message

We also implemented `its-jira` actions that cause Gerrit to automatically comment on Jira issues about the status of changes that are linked to the issue through comment links (see Figure 9). This is beneficial, as now Jira issues automatically contain a trail of related commits and their statuses.



```
▼ 🌱 Gerrit added a comment - 06/May/21 10:44, 7 months ago

Change 5418 merged by Karjalainen Risto:
Add dependency list tooling

https://example.com/gerrit/5418
```

Figure 9: Gerrit reports status changes in Jira

Additionally, we specified a Jira action that allows setting Jira issue statuses to “Implemented” directly from the commit message using special keywords. The Jira configuration for the new actions is placed in a file named `etc/its/actions.config` within Gerrit’s installation directory:

```
[rule "abandoned"]
  its-name = its-jira
  association = somewhere
  event-type = change-abandoned
  action = add-standard-comment
[rule "merged"]
  its-name = its-jira
  association = somewhere
  event-type = change-merged
  action = add-standard-comment
[rule "implemented"]
  its-name = its-jira
  association = footer-Implements-Issue
  event-type = change-merged
  action = Start Progress
  action = Set Implemented
```

Finally, Gerrit was restart to enable the new options. All of the `its-jira` configuration put together creates the following triggers in Gerrit (in order):

- Leave a Jira comment when an associated change is abandoned.
- Leave a Jira comment when an associated change is merged.
- Change the Jira issue status to Implemented when a commit is merged whose commit message footer contains the text `Implements-Issue: <issue ID>`.

Adding a Jenkins user to Gerrit

In preparation for Continuous Integration, we added a user for Jenkins in Gerrit. Jenkins needs an account on Gerrit for accessing repositories and reporting build results. We had a previously generated SSH key pair for Jenkins, so we needed to register the public key for the newly created user. We created the user and registered the public key in one command:

```
cat jenkins_ssh_id.pub | ssh -p 29418 server01.example.com \
  gerrit create-account \
    --group "Non-Interactive Users" \
    --ssh-key - \
    jenkins
```

The command created a user named “jenkins” on Gerrit, places the user in the “Non-Interactive Users” group, and registered the public key `jenkins_ssh_id.pub` for the new user.

Moving the repository to Gerrit

Moving the mainline repository from the old Gitorious server was simple. First we needed to create a new project on Gerrit through the web user interface. Then we cloned the repository from the old mainline, added the new Gerrit server as a new remote for the repository, and pushed the repository to the newly specified remote. This had to be done with a user account that had the ability to push changes directly to Gerrit without going through code review, or the whole history would've had to be reviewed. We achieved the repository transfer by running the following set of commands with an administrator user:

```
git clone ssh://user@oldserver.example.com/amstool
cd amstool
git remote add gerrit \
    ssh://admin@server01.example.com:29418/amstool
git push --all gerrit
git push --tags gerrit
```

Now that the code was successfully on Gerrit, developers were able to move from the old mainline to the new mainline by either updating their old repositories to use the mainline or by cloning the repository again from Gerrit.

Code review and build verification in Gerrit

In Gerrit, other developers have the opportunity to review new code before it is merged to mainline. They can leave inlined comments (see Figure 10) and score it before it is merged to mainline. The code can not be merged before it has passed review.

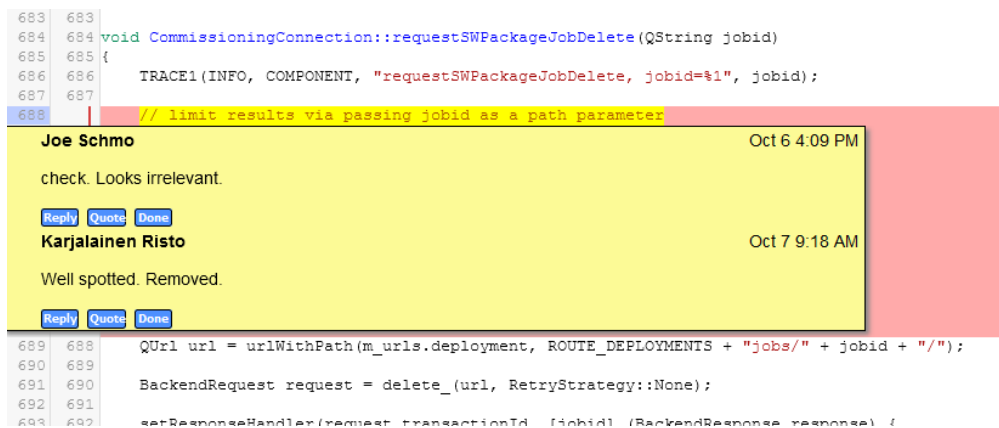


Figure 10: Inline comments in Gerrit

There are four possible scores that the reviewers can give to the change:

- **-2** "This shall not be merged." A score of -2 blocks merging the change completely, even if other reviewers approve it. This block remains in effect until the score is removed. The reviewer who gave the -2 score (or a project lead in special cases) removes the blocking score after problems with the change have been addressed.

- **-1** “I would prefer this is not merged as is.” The reviewer has problems with the change, but does not feel the problems are strong enough to block merging the commit if other reviewers approve it. The change can be merged even if it has -1 scores, but usually even these smaller problems are addressed before merging.
- **+1** “Looks good to me, but someone else must approve.” The change looks good to the reviewer, but they want someone else to review it too before it can be submitted.
- **+2** “Looks good to me, approved.” The change looks good and can be merged to the mainline.

Additionally, commits have a second score “Verified”, which is automatically filled in by the automatic build (called a “commit gate”) that is run by Jenkins (see section 4.3.4). The commit gate ensures that the new build works by building the project, running unit tests, and running static analysis. The Verified label can be given one of two values:

- **-1** “The build fails.” The project has build errors, unit tests fail or the new changes don’t pass the quality gate. Merging is blocked.
- **+1** “The build passes.” Merging is allowed if the commit passes review.

After other developers have approved the commit and the commit gate build has verified the commit, the change can be merged (see Figure 12). If the commit didn’t pass review or the commit gate failed, it can’t be submitted to mainline until the problems have been resolved. Figure 11 shows examples of possible commits scores.

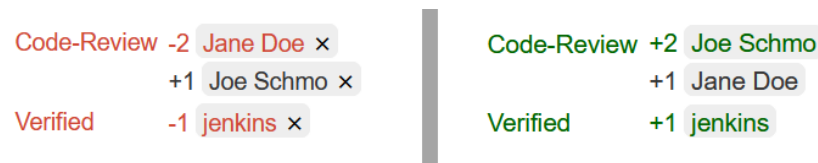


Figure 11: A rejected commit (left) and an accepted commit (right)

After the commit has been submitted to mainline, another automated build job is triggered. This build job is built from the newest state of the mainline and produces the necessary artifacts that are ready to be used in testing or as release candidates.



Figure 12: Submit button is enabled when all the gates pass

4.3.4 CI: Jenkins

Salt created a basic Jenkins master installation running on `server02` and a Jenkins agent on `server03` (see section 4.3.1). The Jenkins master was automatically connected to the company domain by the Salt configuration and we were set up as the administrator user. The Jenkins agent server was provisioned with the Jenkins agent software and a working Docker installation. However, both installations were still otherwise unconfigured, so we needed to configure them.

First we needed to connect the new Jenkins agent to the Jenkins master. We did it by navigating to “Manage Nodes and Clouds” in the Jenkins control panel and selecting New Node. We filled in the details like the name of the server, and set the Launch Method to “Launch agents via SSH”, filling in the hostname and credentials (see Figure 13). After clicking Save, Jenkins connected to the agent successfully and the agent showed up in the list of connected nodes. Now the Jenkins master was ready to delegate jobs to the Jenkins agent.

We also installed the Gerrit trigger plugin from Jenkins’ Plugin Manager. The Gerrit trigger plugin allows the creation of jobs that are automatically started when certain events occur on Gerrit, such as when a new patch is uploaded or when a change is merged.

The screenshot shows the Jenkins 'New Node' configuration page. The form is filled with the following information:

- Name:** server03 (AMS tool)
- Description:** AMS tool linux build slave
- Number of executors:** 4
- Remote root directory:** /home/jenkins
- Labels:** amstool amstool-linux
- Usage:** Only build jobs with label expressions matching this node
- Launch method:** Launch agents via SSH
- Host:** server03.example.com
- Credentials:** jenkins2 (with an 'Add' button next to it)
- Host Key Verification Strategy:** (empty)

A blue 'Save' button is located at the bottom left of the form.

Figure 13: Connecting a new agent to Jenkins

Jenkins jobs

After connecting the Jenkins agent to the Jenkins master, and with the Gerrit trigger plugin installed, the next step was to create the necessary jobs for running AMS tool builds. Jenkins jobs are automated tasks that are run by Jenkins. Triggering a job can be done based on configurable criteria, such as timers (for example weekly builds), Gerrit events (for example commit gates), or even manually.

In our case, we created two jobs:

- **A commit gate job**, which is run every time a new patch is uploaded to Gerrit. The purpose of the commit gate is to test that the new commit builds successfully and passes the quality requirements before it can be merged. A failing commit gate will cause Jenkins to post a negative score to the related change on Gerrit, which will block merging the commit to the mainline.
- **A mainline build**, which is triggered when a commit is merged to the mainline. The purpose of the mainline build is to always build the newest state of the mainline so that binaries and artifacts are available for testing. It also reveals if the mainline has somehow broken despite the commit gate.

Both jobs are almost identical, only their triggers differ. The job definitions were placed in the AMS tool source code repository in a file named `Jenkinsfile`, which is the canonical place for Jenkins pipeline definitions. The job definition file is written in a scripting language called Apache Groovy.

We wrote a `Jenkinsfile` to build all the variants for all their supported platforms (see Table 2 for a list of supported variants and platforms). For this purpose, we used a Jenkins pipeline feature called “Matrix builds”, which allows running the same build steps multiple times, each time with different parameter values. We defined `debian-buster`, `debian-bullseye`, and `ubuntu-xenial` as the supported platforms in the `PLATFORM` variable, and `Commissioning`, `Helpdesk`, and `Internal` as the variants that we want to build. Then we added exclusion rules for variant-platform combinations that we did not want to build.

In the `Jenkinsfile`, the part that deals with the matrix build looks like this (other parts are omitted for clarity):

```

matrix {
  axes {
    axis {
      name 'PLATFORM'
      values 'debian-buster', 'debian-bullseye', 'ubuntu-xenial'
    }
    axis { name 'VARIANT'
          values 'Commissioning', 'Helpdesk', 'Internal' }
    }
  excludes {
    exclude {
      axis { name 'PLATFORM' values 'ubuntu-xenial' }
      axis { name 'VARIANT' values 'Commissioning' }
    }
    /* more excludes... */
  }
  stages {
    stage ('Build') {
      util.build_variant(variant: "$VARIANT", platform: "$PLATFORM")
    }
  }
}

```

The pipeline code causes Jenkins to call a function named `util.build_variant()` with all of the platform and variant combinations that are not excluded in the exclusion list as parameters. `util.build_variant()` is a helper function that uses the dev script that is described in section 4.3.2 to build the project, run tests, run code analysis, and archive the build artifacts for successful builds.

Using the `Jenkinsfile`, we created the commit gate job on Jenkins. First we created a new Pipeline job and set “Gerrit event” as the build trigger. We set “Patchset created” as the trigger (i.e. the condition that causes the build to be run) and filled in the server and branch information. Next we configured the pipeline definition to be fetched from the git repository by choosing “Pipeline script from SCM”, setting the SCM field to value “Git”. With these configurations, Jenkins fetches the `Jenkinsfile` from the AMS tool source code repository and uses it to run the commit gate job. The configuration was tested by uploading test commits to Gerrit. A Jenkins plugin named Blue Ocean shows a visualization of the matrix pipeline (see Figure 14).

If the commit gate build fails, the Blue Ocean view can be used to pinpoint exactly which variant, platform and build step failed (see Figure 15). Additionally, when a build fails, Jenkins blocks the commit on Gerrit, stopping the commit from being merged into the mainline, and leaves a comment that links to the build that failed.

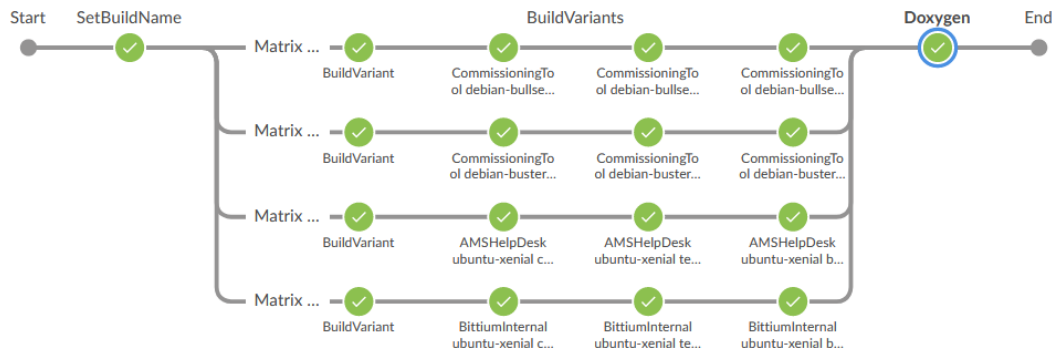


Figure 14: A successful Jenkins commit gate

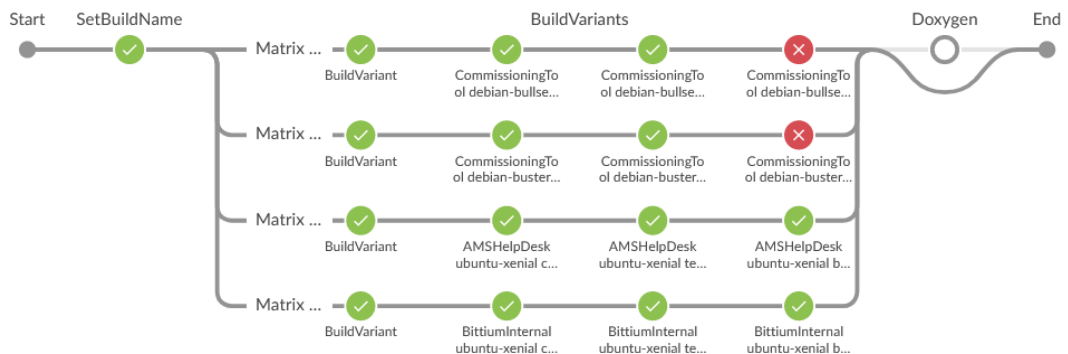


Figure 15: A failed Jenkins commit gate

A second, almost identical job was created for the mainline build job. The main difference was that we set the Gerrit trigger to “Change merged” instead of “Patchset created”, which means that the job is triggered when a commit has already passed the commit gate and code review, and is merged to the mainline. This job can be used to fetch the latest mainline artifacts for testing or releases, and to see the status of the mainline.

4.3.5 Quality Gate: SonarQube

The last component in the toolchain is SonarQube, which acts as the quality gate for each new commit. The basic idea is that when a new build is triggered on Jenkins, one or more steps in the build script uses SonarQube to check if code quality is up to standard. If the quality criteria are not met, the build fails and merging the failed commit to the mainline is blocked. We used the SonarQube C++ Community plugin (sonar-cxx) for analyzing the source code. The plugin works by analyzing reports from other code analysis tools, which we also needed to set up.

We used the following code analysis tools for SonarQube:

- **cppcheck**, which is a static code analysis tool for C++,
- **Clang Static Analyzer**, also a C++ static analyzer, and
- **gcovr**, which is a tool for generating test coverage reports.

Adding quality checks to the dev script

We implemented support for all of the quality check tools in the `./dev` script, which allows us to generate the reports from the command line. For example, to run `cppcheck` and to generate the `cppcheck` report files for SonarQube, we implemented the command:

```
./dev cppcheck <variant> <buildmode>
```

For Clang Static analysis we implemented:

```
./dev scanbuild <variant> <buildmode>
```

And lastly, generating a test coverage report involves first making a debug build and then running all the tests, which generates a bundle of `gcovr`-files that SonarQube can use for test coverage data.

Finally, we implemented an action for `dev` that allows us to send the analysis and coverage reports to SonarQube for analysis.

Putting all of it together, a full SonarQube commit gate sequence consists of running `cppcheck`, running Clang static analyzer, building a debug build, running all tests to generate test fail/pass data and test coverage data, and finally sending all the data to SonarQube for analysis. The full sequence of commands for the Internal variant of the AMS tool looks as follows:

```
./dev cppcheck internal debug
./dev scanbuild internal debug
./dev build internal debug
./dev test internal debug
./dev sonar internal debug
```

After running the set of commands, the build results become available on SonarQube.

SonarQube reports

After a project is set up in SonarQube and it has collected the first set of reports, the project overview becomes available (see Figure 16). The overview shows statistics and information about code quality, such as the number of potential bugs and vulnerabilities that were found by static analysis. The overview also shows other information like the number of unit tests and the test coverage percent, and even gives an estimate for how much work it would take to fix all the issues (“technical debt”).

SonarQube shows potential problems inlined inside the source code (see Figure 17), which makes it convenient to find problems and make decisions on what to do about them. Automation finds and reports the potential issues, but the decision of how to handle them is based on a human review. Individual issues can be flagged as “confirmed”, “false positive”, or “won’t fix”. If an issue is marked confirmed, that indicates that the issue is thought

to be a real problem that needs to be fixed. False positive means that the automatic tooling misidentified some piece of code as problematic, but a human check says otherwise. A “won’t fix” status means that there might be a real problem, but for one reason or another it will not be fixed.

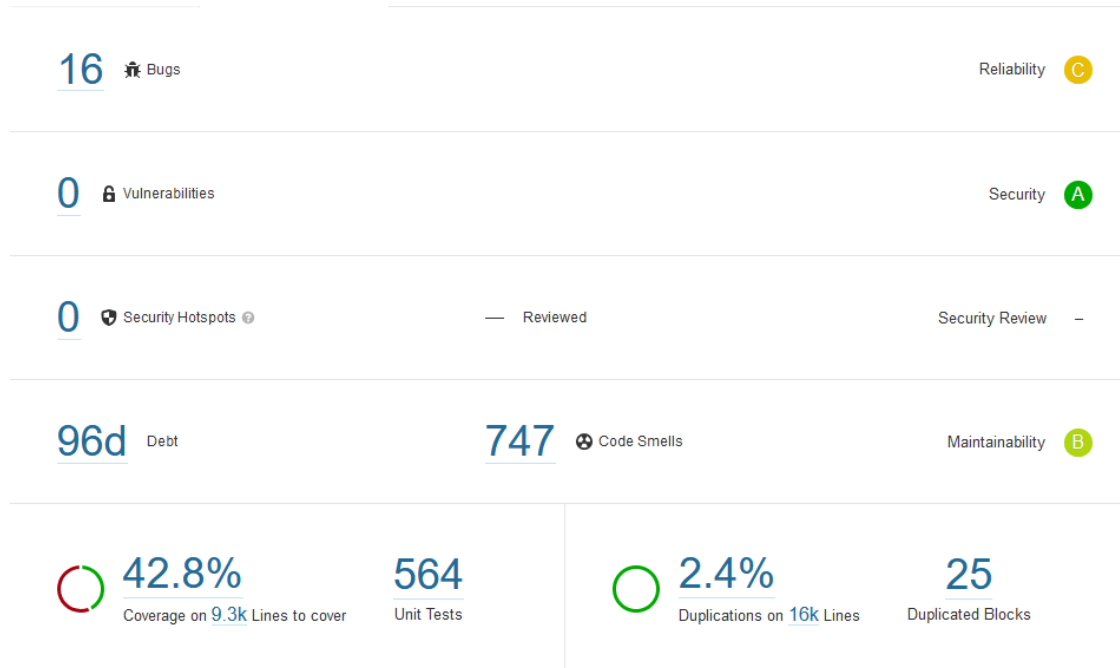


Figure 16: Project overview in SonarQube

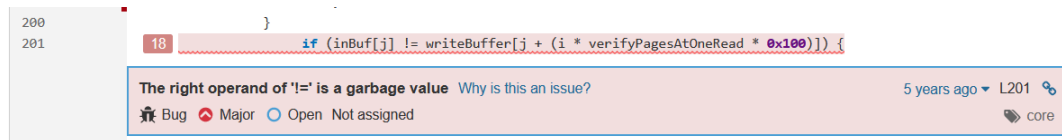


Figure 17: Code quality issue shown in SonarQube

4.4 Putting everything together: the new workflow

Combining all the tools that we installed in this chapter gives us the finalized, improved workflow (see Figure 18). Similarly to the old workflow (Figure 3), the developer still does the main part of their work on their own personal development machine, but instead of pushing changes directly to the mainline, the developer pushes their changes to Gerrit, where commits go through code review and build verification before they can be merged to the mainline. The new toolchain also gives quick feedback to the developer of every stage, so that the developer is able to make informed decisions.

The flow of a typical use case would be:

1. The developer makes a new change on their machine
2. The developer pushes the change to Gerrit
 - Human reviewers inspect the code, leave comments, and either give their approval or rejection.
 - Jenkins and SonarQube check that the commit still builds for all the variants and target platforms, and that any new code passes quality checks.
3. If either Jenkins or human reviewers rejected the commit, the developer has to fix the issues and upload an amended version of the commit for a new round of reviews and verification (back to step 2).
4. When the commit passes both human review and Jenkin's commit gate, it can be merged to the mainline.
5. Merging the commit to the mainline triggers a new mainline build on Jenkins.

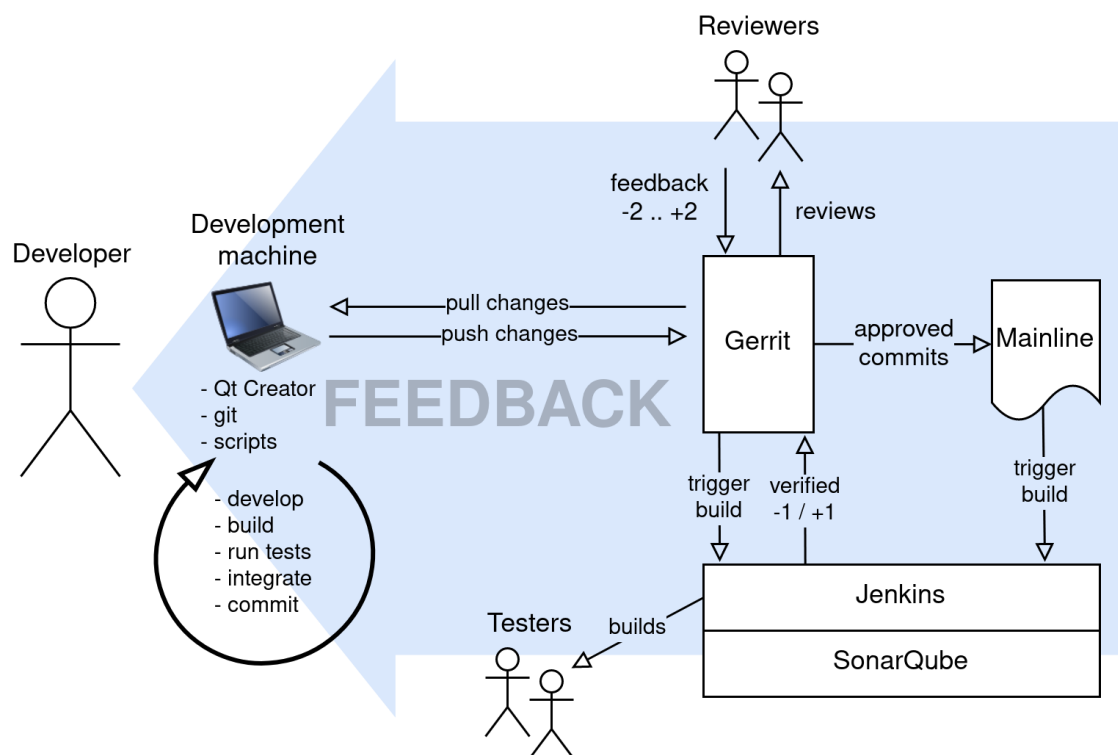


Figure 18: New development workflow

Where testers previously received builds directly from developers (see Figure 4), now testers can download builds from Jenkins at their own convenience, without developer involvement. The binaries produced by Jenkins always have a consistent build environment, and they support version numbering so that all binaries are traceable back to the exact Jenkins build that produced them.

The new workflow addresses all the issues that were identified in the old workflow. Manual processes are fixed by automating them with Jenkins, the inconsistency of mainline builds is mitigated by checking each commit before it can be merged, and the lack of visibility is mitigated by generating quick feedback from multiple points within the workflow.

5 Results

5.1 Quantitative analysis

Evaluating the old workflow through metrics was challenging. Data for it was not readily available because manual processes do not yield themselves well to recording statistical information meticulously. Jenkins and SonarQube generate metrics for the new workflow automatically, but those metrics were not available for the old workflow. Therefore we devised a way to generate metrics for the old workflow after the fact. The approach we took was to use the commit history of the source repository to automatically check out all past changes and attempt to build them one by one.

For the build attempts, we used a slightly altered version of the dev script that we implemented in section 4.3.2. The main alteration consisted of adding a timeout for the build attempt, as some older builds were so badly broken that the build got stuck in an infinite loop and would not finish naturally. We set the timeout to 10 minutes, which was ample time for a successful build to finish.

After the dev script exited for each build, the success or failure of the build was determined by searching the project tree for the main executable file that all builds are expected to produce. If the executable file was found, the build was considered successful, and a failure otherwise.

The script that we used to build each commit and to determine success or failure looked roughly as follows:

```
git clone "ssh://server01.example.com/amstool" && cd amstool
git log --pretty="format:%H" --reverse > ../commitlist
while IFS= read -u 9 commithash; do
    git clean -fxd
    git checkout "$commithash"
    rm -f dev
    cp ../dev dev
    ./dev build "$variant" release
    buildstatus="FAIL"
    for artifact in "$(find . -type f -and -name 'amstool')"; do
        if file "$artifact" | grep "LSB executable"; then
            buildstatus="OK"
        fi
    done
    echo "${commithash}:${buildstatus}" >> ../builddata
done 9< ../commitlist
```

The script was run separately for all three variants (helpdesk, internal, commissioning) by

setting the variable `$variant` to the appropriate value. Finally the results for all variants were collected into one file. The results were grouped by month, which was taken from the commit date, and the monthly success rate for each variant was calculated by dividing the number of successful builds in each month by the total number of commits in that month. See Appendix 1 for the raw data for monthly success rates. Then we plotted the success rates with respect to time for each variant (see Figure 19).

The first toolchain related commit was created on September 20 of 2018. The graphs were plotted at one month resolution, so the start of the new workflow was marked on October 1st, which also was the first day of the first full month with the new workflow. Each of the graphs in Figure 19 show a dotted vertical line to mark the date.

Before the new workflow adoption, the graphs show heavy fluctuation for the build success rates. There are periods when the success rates for two or three variants were synchronized, but most of the time they varied independently. The effect is clearest near the beginning of 2018, where the graph shows that as development of the new Commissioning variant began, build success rates for other variants crashed. We speculate that this is because of the reliance on manual processes—if a developer worked on one variant, there was a danger of breaking other variants without knowing it because there was no mechanism to block broken commits or inform developers of the breakage automatically.

Some of the instability during the year 2017 can be explained by the fact that the project was still in its beginning phases when most of the development work was focused on implementing new features. Maturation didn't begin until late 2017 and early 2018. While this may be a factor, it alone is not enough to explain the drastic differences before and after CI was introduced.

After the new workflow was taken into use, the success rates for all variants jumped to 100% and stayed there permanently. The quantitative analysis demonstrates that builds were very inconsistent with the old workflow (there were very few months where all the builds passed), and very consistent with the new workflow.

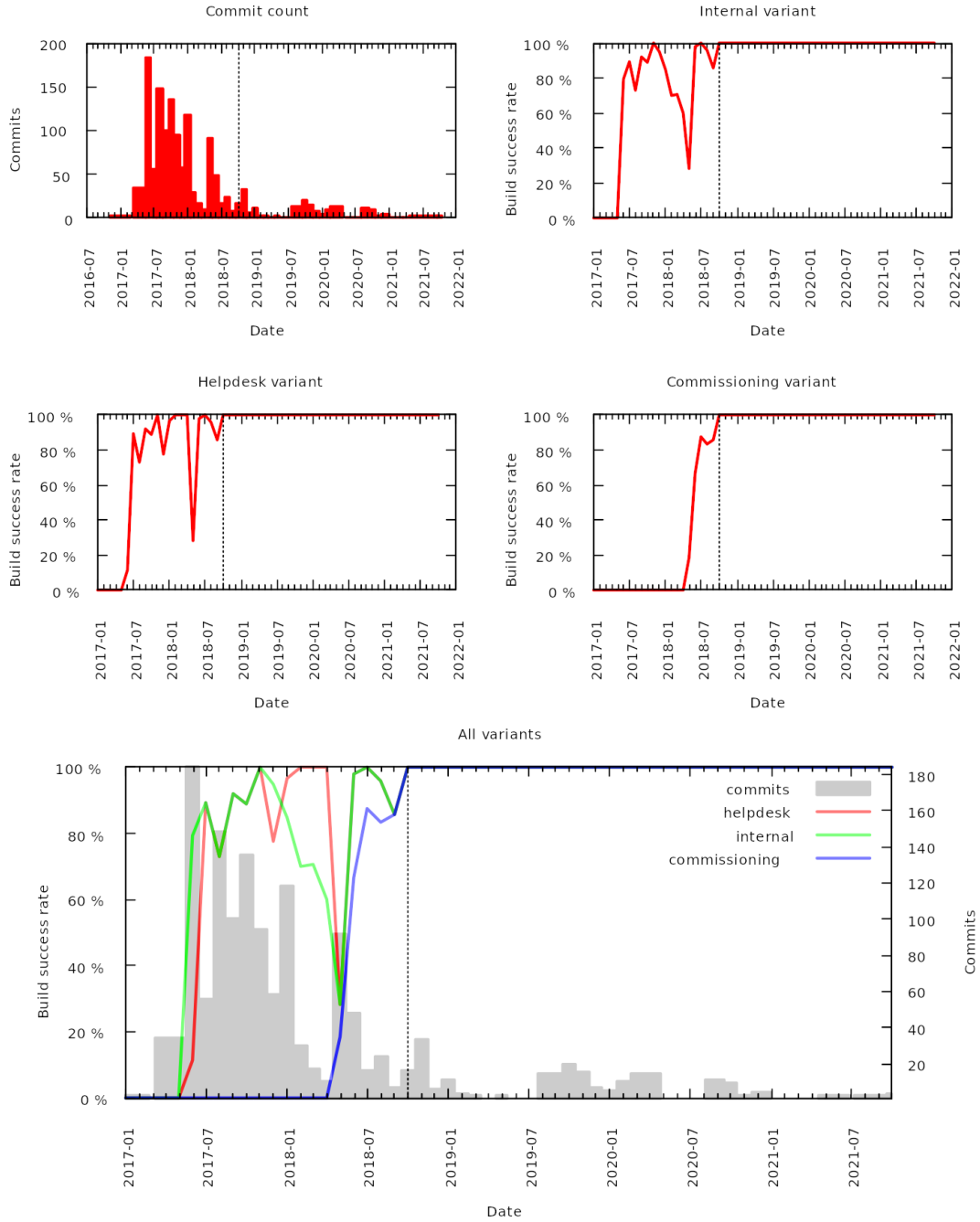


Figure 19: Build success rates

5.2 Interview results

To evaluate the success of the new workflow from different perspectives, we interviewed three project members who had experienced both the old workflow and the new workflow. All the interviewees had different roles in the project (as shown in Table 1): Person 1 was an architect, Person 2 was a developer, and Person 3 was a tester. The interview was a semi-structured interview that was structured in chronological order. We started by asking the interviewees about the old workflow and their experiences with it. Then we moved on to the new workflow, asking about experiences and comparing it to the old workflow. The last part of the interview discussed future improvements. The full interview structure and the profiles of the interviewees are described in chapter 2.

The different perspectives rising from the interviewees' different roles in the project proved enlightening. Person 3 (tester) mostly brought up testing related issues. Person 2 (developer) focused on development workflows. Person 1 (architect) had a more holistic view and discussed both development and testing. Despite the different roles and perspectives, all interviewees were familiar with the tools and workflows and made use of them regularly, which gives an insight into why DevOps is often strongly connected to cooperation: it often resides at the intersection of different competence areas and organizations.

When asked how interviewees remembered the old workflow, Person 1 (Architect) and Person 3 (tester) both brought up the inconsistent build status in the mainline. Person 1 (architect) said that the mainline was almost never fully working, and that bringing the mainline to an acceptable level of quality required temporarily stopping the development of new features and focusing on maturation. Person 3 (tester) commented that when an unreviewed commit was uploaded, it might cause the mainline to not even compile, which would result in wasted time. Person 2 (developer) also agreed that there was no way to know if the mainline was working or not, but they also liked the ability to work fast and freely without restrictions at the start of the project when new features were being implemented.

Making releases was remembered as chaotic in the old workflow. Person 1 (architect) said that releases were built on specific developer's machine and there was always trepidation about whether the build would even work if made on someone else's computer, due to the inconsistent build environments. Person 2 (developer) said the process was very manual, and files were transferred to testers on USB flash memory without much organization. Person 3 (tester) said that the the USB flash memory might have a dozen different versions with inconsistent numbering and file name suffixes like "final" or similar. Person 3 (tester) also stated that the build numbers for the binaries that were delivered to testing were not available anywhere, so versioning was a matter of manual bookkeeping. Jenkins was seen as a solution to the releasing problems. Especially person 3 (tester) was satisfied that they can now get build artifacts directly from Jenkins, and that they are properly versioned.

When discussing code review and Gerrit, all the interviewees saw it as a positive factor. The

common sentiment was that while code review seemingly slows down development, it is worth the cost because it helps in avoiding problems that would arise without it. Person 1 (architect) commented that without code review, we would be back to the old bad situation (referring to the old workflow). Person 2 (developer) saw code review as necessary, especially close to release. Person 3 (tester) told a specific anecdote about accidentally including a block of test code into a commit, and the problem being found in code review before the commit was merged to mainline.

Jenkins was also seen very positively. Besides helping with releases, Person 2 (developer) also said that they are happy that Jenkins verifies the build before it can be merged, as doing all the work of checking all the variants manually by developers would be tedious.

SonarQube was familiar to all of the interviewees, but it had seen less use. Person 1 (architect) said that they like SonarQube, but hadn't used it in several months. Person 2 (developer) knew that the tool exists, but had not used it.

When asked directly how the interviewees would compare the new workflow to the old workflow, the responses were positive. Person 1 (architect) brought up visibility into mainline status and the traceability of old builds that Jenkins offers. Person 2 (developer) stated that the new workflow is the best way to do software development that they know of. Person 3 (tester) saw especially automation as a strong positive and something to aim for, and stated that in that and other respects the current workflow is better than the old workflow.

6 Discussion

This thesis project has been long in the making. The first meeting with the client, where the goals of the project were roughly outlined for the first time, was held almost four years ago in early 2018. The implementation phase took place mostly over the fall and winter of the same year.

The implementation phase was reasonably straightforward, but the broad scope of the assignment proved a bigger difficulty than anticipated when writing this report. There was a constant struggle over what aspects of DevOps to include in the report and what to leave out. In the end, the core topics were whittled down to code review, continuous integration, and code analysis. Leaving out many interesting topics was not an easy decision, but it was a necessary one for this thesis to ever see the light of day.

In the years after its initial conception in 2018, the toolchain has gone through numerous smaller improvements, fixes, and updates, but the core concepts have remained the same and the system still works well. The times when pulling new changes from the mainline might break the build or when you were able to push changes without code review are now a distant memory, and the ability to make releases easily and to add and remove target platforms through the dockerized build system have proven invaluable over the years.

The toolchain is still not perfect, however. The interviews revealed the low adoption rate of SonarQube, which is perhaps the biggest shortcoming of this thesis project. The interviewees said that they like SonarQube in principle, but were unable to specify why they don't use it. We speculate that the reason (or at least a factor) for the low usage of SonarQube is related to a failure in setting up the quality gate filters optimally. As it is currently, SonarQube shows too many false positives and searching through the noise to find the signal is too time-consuming. SonarQube is a powerful tool, but making optimal use of it seems to require continual attention to keeping the quality rule filters up to date so that the warnings are always relevant and interesting.

Two of the three interviewees also brought up test automation as the item that needs the most improvement. The project does have unit tests and integration tests, but at the time of writing this thesis, system testing is still done manually. It would be possible to automate large parts of system testing with tools such as Robot Framework—perhaps a topic for another thesis.

For the quantitative analysis section in the results chapter, gathering metrics that could be used for gauging the success of the thesis project was problematic. Automation in the new workflow naturally generates a lot of useful metrics, but those metrics are not available for the old workflow and therefore could not be used for evaluating the success of the new workflow itself. The approach that we took in this thesis—i.e. rewinding old commits to check their build success rates—grew out of a necessity to find at least some ways to measure the old workflows. An interesting future study would be finding other metrics to measure besides build success rates—for example code quality indicators, test coverage data, test pass and failure data, etc—and seeing how the metrics evolve during and after the adoption of CI.

From the author's perspective, the most striking result and the biggest success of the thesis was the build success rate graph (Figure 19). It is easy to say “we will block broken commits from the mainline”, but it's another thing to see with your own eyes how the build success rate jumps to 100% after the introduction of CI, and how unbudgingly it remains there. Such a strong visualization drives home how valuable CI is, especially for a project like the AMS tool, where multiple variants are built from the same codebase.

The author's personal conclusion from working on this thesis, and from experiencing both manual and automated workflows in various projects, is that no software project—even a small one—should be without CI. There is an initial cost to setting up all the necessary tooling and workflows, but with all the pains and problems that it can save you from, the investment should pay itself back manyfold over the life cycle of the project.

References

- Bacchelli, A. & Bird, C. 2013. Expectations, outcomes, and challenges of modern code review. In: *2013 35th international conference on software engineering (ICSE)*. IEEE, pp.712–721.
- Bittium. 2021. Company overview. Internet. Accessed on 7.11.2021. Retrieved from <https://www.bittium.com/about-bittium/facts-figures/company-overview>.
- Clang Static Analyzer. 2021. Web article. Internet. Accessed on 14.11.2021. Retrieved from <https://clang-analyzer.llvm.org/>.
- Duvall, P., Matyas, S. & Glover, A. 2007. *Continuous integration: Improving software quality and reducing risk*. Pearson Education.
- Fowler, M. & Foemmel, M. 2006. Continuous integration. Accessed on 13.11.2021. Retrieved from <https://martinfowler.com/articles/continuousIntegration.html>.
- Hogg, S. 2014. Software containers: Used more frequently than most realize. Internet. Accessed on 30.11.2021. Retrieved from <https://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html>.
- Miller, A. 2008. A hundred days of continuous integration. In: *Agile 2008 conference*. IEEE, pp.289–293.
- Sadowski, C., Söderberg, E., Church, L., Sipko, M. & Bacchelli, A. 2018. Modern code review: A case study at Google. In: *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. pp.181–190.
- Salt (software). 2021. Article from Wikipedia. Internet. Accessed on 22.11.2021. Retrieved from [http://en.wikipedia.org/w/index.php?title=Salt%20\(software\)&oldid=1050581920](http://en.wikipedia.org/w/index.php?title=Salt%20(software)&oldid=1050581920).
- Ståhl, D. & Bosch, J. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, **87**: 48–59.
- Ståhl, D., Mårtensson, T. & Bosch, J. 2017. Continuous practices and DevOps: Beyond the buzz, what does it all mean? In: *2017 43rd euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, pp.440–448.
- Vargo, S. 2015. The 10 myths of DevOps. Internet. Accessed on 7.11.2021. Retrieved from <https://www.youtube.com/watch?v=oT7-L6Oe4Tg>.

Appendices

Appendix 1. Build success rate data

Date	Commits	Helpdesk	Internal	Commissioning
2017-01	2	0.00 %	0.00 %	0.00 %
2017-05	34	0.00 %	0.00 %	0.00 %
2017-06	184	11.41 %	79.35 %	0.00 %
2017-07	56	89.29 %	89.29 %	0.00 %
2017-08	148	72.97 %	72.97 %	0.00 %
2017-09	100	92.00 %	92.00 %	0.00 %
2017-10	135	88.89 %	88.89 %	0.00 %
2017-11	94	100.00 %	100.00 %	0.00 %
2017-12	58	77.59 %	94.83 %	0.00 %
2018-01	118	96.61 %	84.75 %	0.00 %
2018-02	30	100.00 %	70.00 %	0.00 %
2018-03	17	100.00 %	70.59 %	0.00 %
2018-04	10	100.00 %	60.00 %	0.00 %
2018-05	92	28.26 %	28.26 %	18.48 %
2018-06	48	97.92 %	97.92 %	66.67 %
2018-07	16	100.00 %	100.00 %	87.50 %
2018-08	24	95.83 %	95.83 %	83.33 %
2018-09	7	85.71 %	85.71 %	85.71 %
2018-10	16	100.00 %	100.00 %	100.00 %
2018-11	33	100.00 %	100.00 %	100.00 %
2018-12	6	100.00 %	100.00 %	100.00 %
2019-01	11	100.00 %	100.00 %	100.00 %
2019-02	3	100.00 %	100.00 %	100.00 %
2019-03	2	100.00 %	100.00 %	100.00 %
2019-04	1	100.00 %	100.00 %	100.00 %
2019-05	2	100.00 %	100.00 %	100.00 %
2019-06	1	100.00 %	100.00 %	100.00 %
2019-09	14	100.00 %	100.00 %	100.00 %
2019-10	20	100.00 %	100.00 %	100.00 %
2019-11	15	100.00 %	100.00 %	100.00 %
2019-12	7	100.00 %	100.00 %	100.00 %
2020-01	5	100.00 %	100.00 %	100.00 %
2020-02	10	100.00 %	100.00 %	100.00 %
2020-03	14	100.00 %	100.00 %	100.00 %
2020-07	1	100.00 %	100.00 %	100.00 %
2020-09	11	100.00 %	100.00 %	100.00 %
2020-10	9	100.00 %	100.00 %	100.00 %
2020-11	2	100.00 %	100.00 %	100.00 %
2020-12	4	100.00 %	100.00 %	100.00 %
2021-02	1	100.00 %	100.00 %	100.00 %
2021-03	1	100.00 %	100.00 %	100.00 %
2021-04	1	100.00 %	100.00 %	100.00 %
2021-05	2	100.00 %	100.00 %	100.00 %
2021-09	2	100.00 %	100.00 %	100.00 %
2021-10	3	100.00 %	100.00 %	100.00 %