BACHELOR THESIS

Nick Koops

Continuous Integration und Continuous Delivery - Ein Vergleich zwischen Jenkins und Dagger

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Nick Koops

Continuous Integration und Continuous Delivery - Ein Vergleich zwischen Jenkins und Dagger

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Stefan Sarstedt
Supervisor: Prof. Dr. Lars Hamann

Submitted on: 24. März 2023

Nick Koops

Title of Thesis

Continuous Integration and Continuous Delivery - A comparison between Jenkins and Dagger

Keywords

Continuous Integration, Continuous Delivery, Jenkins, Dagger, Software engeneering survey

Abstract

Agile practices with continuous integration and continuous delivery (CI/CD) pipeline approaches have increased the efficiency of projects. There are established (Jenkins) and newly developed (Dagger) CI/CD tools on the market. Jenkins pipelines are executed in the environment Jenkins is running in, making it complex to provide a consistent build environment across multiple machines. Dagger builds a layer on top of the CI/CD platform by running inside a Docker environment and thus decoupling the build environment from the build machine. This thesis compares Jenkins and Dagger to each other. Dagger improves on Jenkins by unifying and simplifying the process of extending pipelines, reusing components and making pipelines executable and debuggable locally. Dagger is deficient in the majority of the functionalities that are available through Jenkins plugins, the documentation is inadequate, Dagger is not usable in some edge cases and there is no experience of using Dagger in a large scale commercial application.

Kurzzusammenfassung

Agile Methoden erhöhen mithilfe von Continuous Integration und Continuous Delivery Pipelines (CI/CD) die Effizienz der Entwicklung von Software Projekten. Auf dem Markt gibt es etablierte (Jenkins) und neu entwickelte Tools (Dagger). Jenkins Pipelines werden in derselben Umgebung ausgeführt, in der Jenkins läuft, wodurch es komplex ist, eine konsistente Build Umgebung über mehrere Maschinen hinweg bereitzustellen. Dagger bildet eine Schicht oberhalb der CI/CD Plattform, in dem es einer docker Umgebung läuft, wodurch es die Build Umgebung von der Build Maschine entkoppelt. Dagger baut auf etablierten CI/CD Tools auf, in dem das Erweitern, Wiederverwenden, und lokale Ausführen der Pipeline Teil seiner Architektur sind. Im Vergleich zu Jenkins hat Dagger hat ein Defizit an Funktionen, die Dokumentation ist inadäquat, Dagger ist in Sonderfällen nicht einsetzbar und es gibt keine Erfahrung mit dem Einsatz in kommerziellen Großprojekten.

Contents

List of Figures

1 Introduction

This chapter introduces the motivation behind the use of agile methods and CI/CD pipelines in software development. Additionally it outlines the aim of the thesis by presenting the research question addressed in this thesis.

1.1 Motivation

Todays software development process is increasingly driven by unpredictably and rapidly evolving markets, constantly changing customer requirements, and frequent changes in information technology [45]. Agile methods aim to address these challenges by allowing a shorter time-to-market, more frequent iterations, and therefore a higher flexibility in responding to changing requirements and quicker implementation of customer feedback [36]. The agile manifesto names customer satisfaction "through early and continuous delivery of valuable software" [35] its highest priority. The employment of CI/CD pipelines allows teams to shorten the software delivery cycle by automating integration, delivery and production deployment [45]. Todays market of CI/CD tools is filled with options targeting different use cases, platforms and team structures [52]. Some of the tools, like Jenkins [23] and GitLab [49], have been around for more than a decade, while there are newer developments like Dagger [29], which has been released in 2022.

1.2 Aim of the thesis

This thesis compares an established CI/CD tool, Jenkins, to a newly developed tool, Dagger, as well as outline and discuss their advantages and disadvantages to answer the following question RQ:

> **Does Dagger satisfy developers' unmet needs and help to address issues they have with Jenkins?**

First, the fundamental concepts of CI/CD and Docker are introduced. Then a survey is conducted to get an understanding of what functionalities developers are missing in established CI/CD tools. Then, a Jenkins and a Dagger pipeline are developed, each used to build a separate open source software project. The gained experience, as well as literature, is then used to compare Jenkins and Dagger, outlining each tool's up- and downsides. Finally the thesis is summarized and RQ is answered.

2 Fundamentals

This chapter introduces and discusses the terms continuous integration (CI) and continuous delivery (CD). Furthermore, an example implementation of CI/CD in the form of a CI/CD pipeline is shown. It also explains the fundamentals of Docker, which are needed in order to understand Dagger.

2.1 Concepts of CI/CD

2.1.1 Continuous Integration

Continuous software delivery, responding to changing requirements and finally measuring progress by working software are key principles in the agile manifesto [35]. CI Software helps an agile development team to integrate their software frequently, usually once or multiple times per day [48, 37]. Fowler [37] describes the following practices of CI: An integral part of CI is maintaining a single source repository, containing everything required for a build including tests, install scripts, third party libraries, etc. The repository should also contain a mainline branch, on which current development branches are based. A large coverage with automated tests makes finding and fixing bugs more efficient. Methods like TDD [34], in which tests are written before the functional code making the tests pass, support self-testing code. Self testing code is also found under the name "continuous testing" [51]. The aforementioned aspects are important for build automation, in order to enable builds on virgin machines with just a single build command. Modern build tools, like gradle [27], meet those requirements by allowing to manage project dependencies, build and test the project etc. These automated builds on a separate integration machine are triggered by commits on the mainline, producing a tested and validated artifact. In order to keep a stable development base, broken builds on the mainline need to be fixed immediately. Keeping builds short, ideally within ten minutes, helps to get immediate feedback on changes to the repository. The state of the mainline build should be displayed to every team member.

2.1.2 Continuous Delivery

According to Fowler [38] CD builds on CI by pushing artifacts produced by CI into "increasingly production-like environments" [38] to ensure compatibility with the production environment. This is done by an automated deployment pipeline. The key benefits are reduced deployment risks, since just small incremental changes are deployed, which means fewer, easier to fix problems appear. Small increments also means earlier and more frequent user feedback to find out the value of a given change in software. Furthermore the development team's progress can be measured by deployments, giving them a "believable" [38] metric. The aim of CD is to make the current development build deployable in production at any time.

2.2 CI/CD pipeline

Software delivery is done by an automated build pipeline, modeling build steps, different environments and quality gates. Quality gates help to orchestrate the build process by ensuring, that in case one step fails to execute successfully, the subsequent build steps are not executed [51]. The pipeline will differ from one project to another in terms of type and number of stages.

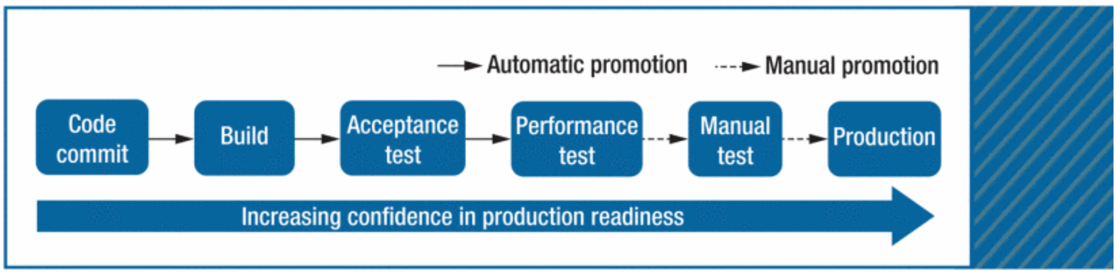


Figure 2.1: An example CI/CD pipeline [33]. Promotion: Advancing the build from one stage to the next. Quality is ensured by quality gates.

2.2.1 Code commit

The commit stage compiles code and runs unit tests. It is triggered automatically, as soon as code is committed to the repository and therefore provides instant feedback on the code to the developer. This stage's quality gate is fulfilled by successful compilation and execution of unit tests. If gate is fulfilled, the pipeline automatically advances to the next stage. If not, the developer needs to fix the code [33].

2.2.2 Build

The build stage runs the unit tests again to generate a test code coverage report, runs integration test, static code analyses and builds artifacts for the release. The artifacts are uploaded to a repository, which manages the releases for deployment and distribution. These artifacts are used by all subsequent stages. In case any errors occur, the pipeline will stop and notify the developer, otherwise the pipeline will continue [33].

2.2.3 Acceptance test

Acceptance tests ensure, that the software meets the specified requirements. A production-like environment, where the software is deployed to, is automatically created and configured by the pipeline. Then the acceptance tests are run in this environment. In case any errors occur, the pipeline will stop and notify the developer, otherwise the pipeline will continue [33].

2.2.4 Performance test

This stage measures the performance impact the code changes have. An environment is set up by the pipeline, in which the performance tests are run. The results are centrally monitored by a tool such as Nagios, New Relic, etc. [51]. This stage gives the developer immediate feedback, whether their code changes harm the software's performance. Fixing possible performance issues at this early stage in the development process is much cheaper, than fixing them before a release [33].

2.2.5 Manual tests

In some scenarios, i.e. when testers perform explorative tests or a specific feature can only be tested by humans, manual tests are necessary. The pipeline automatically creates an artifact in the correct configuration and environment and notifies the testers with the necessary information to access the application. Once the manual tests are performed successfully and the artifact has passed all quality gates, the artifacts are considered a release candidate. They are now ready to be deployed to production [33].

2.2.6 Production

The artifact can now be deployed to production with the click of a button [33].

2.3 Docker

In addition to the basics of CI/CD, basic knowledge of Docker is required to understand the foundation of Dagger, as well as how Jenkins is hosted. Docker was developed from the need of "computational reproducibility" [32]. Docker achieves computational reproducibility by building on technologies from operating system research including Linux containers (LXC), virtualization of the operating system and a hashed-based git-like versioning system.

2.3.1 Docker images

Docker images work similarly to virtual machines, since they have all dependencies necessary to run the image already installed, configured and tested. A key difference between Docker images and virtual machines is that Docker images share the Linux kernel with the host machine. Consequently any Docker image must be based on a Linux system with Linux-compatible software. When the host machine is not running Linux, i.e. on Windows or macOS, Docker runs inside a virtual Linux machine. This makes Docker more lightweight and higher performing compared to a fully virtualized machine, in case of running on an Linux OS. This made Docker particularly attractive to commercial applications, i.e. in a microservices architecture [41].

2.3.2 Dockerfiles

Dockerfiles are the blueprint to a Docker image. Dockerfiles are a script defining how to build an image. Dockerfiles are written in a simple Domain Specific Language (DSL). Dockerfiles offer multiple advantages:

- Virtual machine images can be many gigabytes large, since they contain a complete OS. Dockerfiles are just a a small plain text file, which is easily portable.
- Small plain text files are well suited for version control systems such as git, which makes any changes to the Dockerfile trackable.
- Dockerfiles provide a human readable summary of all dependencies, environment variable, etc. required to execute the code. This approach also eliminates the need to separately document all dependencies.
- The Dockerfile contains every dependency down to the OS. When built with *docker build* the resulting image will usually not differ from one machine to another, especially not when every dependencies' version is set explicitly.
- A Dockerfile can contain checks and tests after the installation and configuration of software, which verify the successful setup.
- The resulting image can be extended and customized by directly editing the Dockerfile.

2.3.3 Integration

Compared to a virtual machine Docker does not replace a user's existing toolchain, but is optimized to run a single application. Therefore a developer can continue to use a local IDE, but execute the code inside a standardized container ensuring an identical environment and thus portability and reproducibility. This means, that the software can be easily deployed in the cloud or on another developer's machine with no more effort than installing Docker, since Docker manages the packaging and execution in a way, so that it works identically across different machines, while exposing the required interfaces for networking, volumes, etc. For example a developer might want to execute their code on a more powerful server compared to their local machine, in which case the developer can export a snapshot of a running container and thus run a container an identical environment on a different machine.

3 Survey

In order to be able to answer RQ it is necessary to get an understanding of what functionalities developers are missing in established CI/CD tools. Therefore a survey is performed as proposed by [47] in five steps: Survey definition, survey design, implementation, execution and analysis. This chapter describes and discusses these steps generally, as well as in the context of RQ. Finally the survey's results are presented, analyzed and summarized.

3.1 Definition

The survey is an online survey interviewing software developers. The survey's aim is to get an understanding of what the respondents have used CI/CD tools for in the past, currently use them for, what advantages and disadvantages they see in the tools they are experienced with, wether they are familiar with newly developed CI/CD tools and their suggestions to improve new CI/CD tools compared to tools they currently use.

3.2 Design

For every online survey the respondents have to be selected in a process called sampling. The selected respondents have to fill in a questionnaire. This section discusses sampling and designing the questionnaire.

3.2.1 Sampling

Sampling means selecting a set of respondents from the superset under study, which is called population. Only a subset is selected because of the limited time, and in case of a large-scale survey money, budget. There are two main methods for sampling.

Systematic sampling

Firstly there is probability sampling, where a systematic approach is used to select a subset of respondents. In systematic sampling the entire population has to be known in order to select members of the population is such a manner, that every member has statistically seen an equal chance of being selected. This means, that it is possible to derive results from the sample that are representative for the whole population with minimal error. Therefore the sampling error mainly depends on sample size, the population size and the variety within the population. Hence the sample size has to be selected according to the tolerated sampling error.

Non-systematic sampling

Secondly there is non-systematic sampling, which is applied when systematic sampling is not feasible, i.e. when there are unknown individuals in the population or in a small-scale survey with a too small population. For reference, [47] considered surveys with a sample size of 102 to 865 samples as small-scale surveys, where they chose a non-systematic approach. Non-systematic approaches imply, that error between population and the sample are unknown. Therefore the characteristics of the sample have to evaluated afterwards. There a number of techniques for non-systematic sampling. One is convenience sampling, where the researcher chooses the most convenient respondents for the survey. This can be refined further by characterizing groups of the population and performing convenience sampling within the groups.

Choice of sampling method

[47] differentiates between two types of online surveys. The first type is a personalized survey, where every respondent is personally invited. Most of the time there is an access control (i.e. username and password or a personalized access code), ensuring only invitees can participate in the survey. It is possible to perform these kinds of surveys with both systematic and non-systematic approaches.
The second type are self-recruited surveys, where respondents find out about the survey by themselves, i.e. by popups on a website or e-mail newsletters. These surveys lead to a non-systematic approach by definition.
This survey uses a personalized non-systematic approach with convenience sampling.

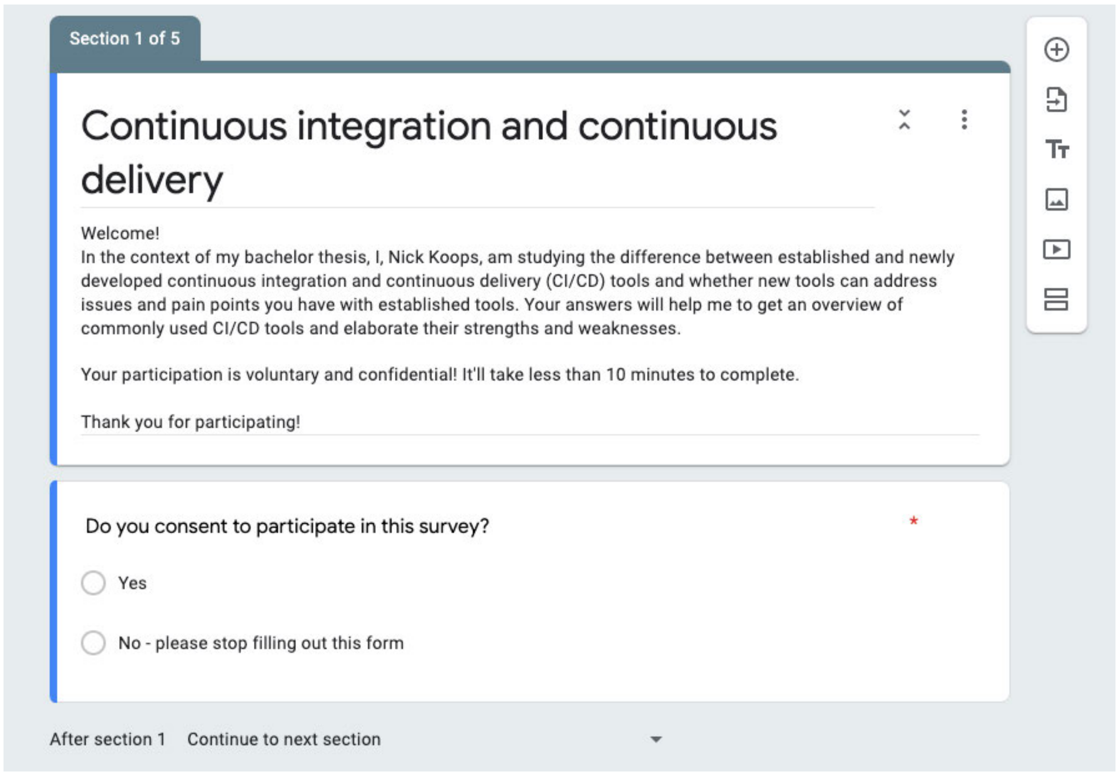
Since the scope for this theses is very limited, the small sample size is very small with three to eight respondents, who will be chosen by availability at the time of writing.

3.2.2 Questionnaire design

Designing a questionnaire is the second important aspect in a survey. As defined by [47] the order and skip-patterns are just as important as the quality of the questions themselves. Skip-patterns are possible branching conditions within the questionnaire, designed to provide different paths through the questionnaire, depending on the answers the respondent gave. For example, the answer to a question might influence the answer to a subsequent question or make the respondent leave out an answer. Therefore a clear definition all possible paths in the questionnaire is necessary.

3.3 Implementation

The survey will be implemented with a tool, that allows the creation of online forms with different types of questions such as multiple choices or free text answers and rating statements on scale. Also, the tool needs to be able to provide the data, that has been filled in by the respondents. There are several tools on the market, which provide these functionalities [39]. Google Forms [12] and Microsoft Forms [16] offer the required functionalities for free, which is why the choice is between these two options. Ultimately, the choice fell on Google Forms, because it allows one to export the form as PDF, which Microsoft Forms does not. A screenshot of the online editor of Google Forms can be seen in 3.1.


Section 1 of 5

Continuous integration and continuous delivery

Welcome!
In the context of my bachelor thesis, I, Nick Koops, am studying the difference between established and newly developed continuous integration and continuous delivery (CI/CD) tools and whether new tools can address issues and pain points you have with established tools. Your answers will help me to get an overview of commonly used CI/CD tools and elaborate their strengths and weaknesses.

Your participation is voluntary and confidential! It'll take less than 10 minutes to complete.

Thank you for participating!

Do you consent to participate in this survey? *

Yes

No - please stop filling out this form

After section 1 Continue to next section

Figure 3.1: A screenshot of the online editor of Google Forms. It has a graphical user interface, which allows the creation of forms with predefined modules, such as questions, text and images. The screenshot shows the introduction and the first question of the questionnaire found in A.1.

3.3.1 Development guidelines

[47] present guidelines they learned from their experiences in designing online surveys. Most of their guidelines do still apply to designing online surveys, for example the amount and type of questions. Others like readability, instructions on how to fill in an online survey and recovery of lost data in case a respondent's browser crashes are no longer a concern for the developer of the survey, since these guidelines are handled by the tool that is used to design and execute the survey.

Number of questions

One important aspect is the amount of questions asked. When there are too many questions, answering them takes more time. When the respondents feel, that the time is not well invested, they will not answer the questionnaire. To keep the questionnaire as short as possible, the questions should be restricted to the topic defined in 3.1. Restricting the amount of questions also helps to keep the complexity in the analysis down. In order to add variation to the answers, some personal questions should be added. [47] conducted multiple surveys with approximately 30 to 90 questions. They found, that respondents answered the surveys despite them having about 90 questions, because the respondents felt, they were learning enough to justify the time invested.

Type of questions

Also, the type of questions has to be considered. While closed questions with fixed answers, i.e. multiple choice questions, are easier to analyze, one also has to integrate open questions with empty text boxes in order to obtain important background information from the respondent. These open questions support the interpretation and quantitative results of the closed questions.

Order of questions

The order of questions is important to keep the respondent motivated, while answering the questions. Questions should be grouped by topic, to maintain a flow through the questionnaire. It is a good practice to start with questions, which engage the respondent to participate in the survey.

3.3.2 Implementing the guidelines

Since the aim of this survey, in the context of a bachelor thesis, is to get a general understanding of what a small set of respondents miss in established CI/CD tools, not a large scale quantitative analysis, the questionnaire will have 21 questions. This way, the complexity in designing and analyzing the results will be kept within the scope of the thesis. The questionnaire can be found at A.1.

3.4 Survey execution

To execute the survey it was given to six coworkers of mine. Three of them participated in the survey leading to a response rate of 50%. The respondents were provided with a link to the survey in Google Forms, where they filled out the survey. Google Forms also provides an overview of the given answers as well as an option to export the results to a spreadsheet. The number of answers per questions varies, since not every respondent answered every question.

3.5 Survey analysis

The overview and the spreadsheet are used to analyze the answers. The results are visualized and interpreted by deriving graphs, diagrams and quotes from the answers.

3.5.1 Background information

A color is assigned to each respondent: respondent 1 (blue), respondent 2 (yellow) and respondent 3 (orange).

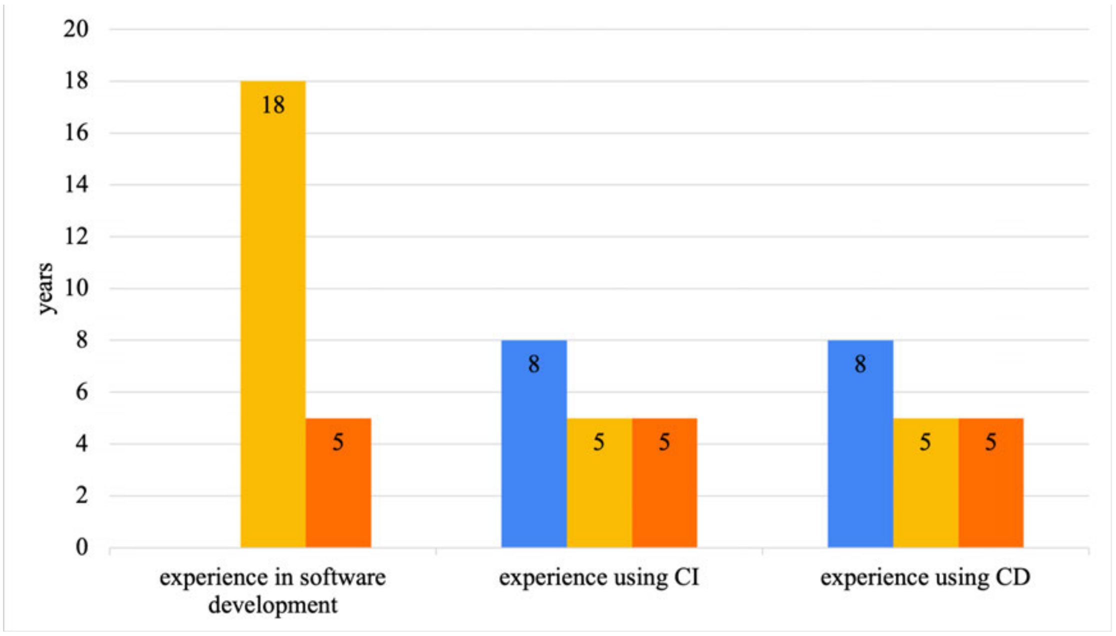


Figure 3.2: Answers regarding the respondents' experience in software development and CI/CD. Each bar represents a respondent's answer.

These answers imply two things: Firstly, respondent 2 did not use CI/CD for the first 13 years of development. Respondent 3 only has experience in software development using CI/CD. Secondly, all respondents use CI and CD in combination with each other, but not independently. Since major companies such as Facebook and Flickr started using CD (and therefore CI) in 2005 and 2009 respectively [48] and influential computer scientists [37] defined CD in 2013, many small and medium sized companies have adopted CI/CD in the last decade. Consequently all respondents use CI/CD in combination. Less experienced developers, such as respondent 3, use CI/CD for all of their career.

3.5.2 Past project specific questions

All of the respondents have worked with CI/CD in previous projects in the context of mobile application development. Respondents 1 and 2 only have experience with Jenkins. Respondent 3 has experience with Jenkins, as well as GitLab and GitHub.

What did you like about the tool(s)?

The respondents mentioned the following upsides of Jenkins: "large community", "big number of plugins", available at no cost and versatility. Participant 3 highlighted the benefits of the Jenkins pipeline compared to a traditional Jenkins job. Participant 3 mentioned GitHub's and GitLab's "everything in one application approach" for Devops, i.e. both GitHub and GitLab include a tool to review pull requests.

What didn't you like about the tool(s)?

Concerning Jenkins, all respondents mentioned the unintuitive web interface, which lacks behind the competition. Respondent 3 also mentioned the lack of tools to debug a Jenkins pipeline.

In which way did the tool(s) make your work more efficient?

All respondents highlighted the instant availability of test results, which in return improves code quality. Participant 2 also mentioned centralized builds. Although the respondents only referred to Jenkins, GitHub and GitLab, the mentioned improvements to their workflows apply to all CI/CD tools as mentioned in 2.1.1.

Did the tool(s) hinder you in any way?

All of the respondents answered, that maintenance can take a lot of time. Respondent 3's reason is time consuming debugging, since in the case of Jenkins it cannot be done locally.

3.5.3 Current project specific questions

All of the respondents currently use CI/CD tools in the context of mobile application development. They also use Jenkins exclusively, since its the default a the respondents' company and they are already familiar with Jenkins.

Why did you choose to use CI/CD for this application?

All of the respondents agreed that they chose to use CI/CD to deploy more frequently, increase confidence in build quality and results and create more visibility for the team. Respondents 1 and 2 also answered to reduce the time spent on setting up how to build and push artifacts by automating these steps using CI/CD. Furthermore respondent 2 added, that CI/CD tools help to document tests and archive old builds.

In case you use more than one CI/CD tool at a time, why do you use more than one?

None of the respondents see a benefit in using more than one CI/CD tool at a time, other than having a backup.

3.5.4 Is your current CD pipeline sufficient for your needs?

All respondents answered, that Jenkins is sufficient for their current workflows.

Respondent 1 answered, that an improved Xcode [24] integration would improve the workflow for building iOS applications.

3.5.5 New CI/CD specific questions

These questions are specifically aimed at the respondents interest in newly developed CI/CD tools and their suggestions for improvement compared to CI/CD tools they are experienced in.

Have you read about or do you have any experience with recently developed CI/CD tools, i.e. Dagger?

Only respondent 3 answered this question with yes, therefore respondents 1 and 2 never looked into new CI/CD tools.

In which way can newly developed CI/CD tools help you improve the pain points you have in your current workflow, if at all?

Respondent 3 suggested, that new tools should be able to execute builds in the cloud, eliminating the need for a local server. Cloud execution is already available for products like CircleCI [6]. Respondent 1 suggested, that newly developed CI/CD tools should reduce the need to script yourself in order to reduce the complexity of the pipeline. Approaches with pipelines consisting of customizable building blocks, such Buddy [2], promise to reduce the pipeline's complexity already exist, although they are not covered in this thesis because of time constraints.

3.6 Summary

This section set out to get an understanding of how experienced software developers have used and still use CI/CD tools in their workflows and what suggestions they have to be improved in newly developed CI/CD tools. Therefore a survey with a questionnaire was constructed as proposed by [47]. The survey was sampled with the population in mind, then designed according to [47] and [54], implemented using Google Forms and finally executed. The survey was answered by three coworkers with an average experience of six years in using CI/CD in their workflows. All of the respondents have experience using Jenkins, one also has experience using GitHub and GitLab. On the one hand, the respondents emphasized Jenkins large community, large selection of plugins and its versatility while being available for free. One respondent also mentioned GitLab's and GitHub's additional devops functionalities. The respondents answered, that CI/CD helps them to improve their code quality, deploy more frequently, create visibility for the entire development team, document tests and archive old builds. On the other hand, the respondents criticized Jenkins' unintuitive web interface and difficult to debug pipelines. The respondents agree, that maintaining CI/CD can be time consuming. Generally, the respondents agree that Jenkins is sufficient for their current workflows. Only one respondent has looked into newly developed CI/CD tools, although another respondent suggested, that new tools should be able to execute builds in the cloud.

4 Development

This chapter is the mainstay of this thesis. It compares Jenkins and Dagger to another by using each CI/CD tool build a suitable software project. Each section introduces the tool, a software project, which is built by CI/CD tool and the CI/CD pipeline including its language used to build the project. Finally, Jenkins and Dagger are compared to each other.

4.1 Jenkins

Jenkins is an open source platform, initially created to automate the build and testing process. The Jenkins build system is written in Java and highly extensible and customizable due to its plugin architecture. Therefore Jenkins is able to cover many possible scenarios and requirements, enabled by thousands of plugins developed by its open source community [31].

4.1.1 Open Source project

To evaluate Jenkins the open source Android app Foodium [46] is used. The Android app is built using Kotlin and uses modern Android development tools, such as Coroutines and Flow [19]. It is built upon Android architecture components in an MVVM architecture style [8].

4.1.2 Hosting

In order to run any build using Jenkins there has to be a running Jenkins instance. To provide a consistent and portable environment with all necessary Android build tools, Jenkins runs inside of a Docker container, which was introduced in 2.3. There are official

Jenkins Docker images available on Docker hub [17], which already include the required JDK to run Jenkins. In addition to a JDK an Android application requires an Android SDK to be built. The Android SDK has to be downloaded and configured manually. This is done inside of the Dockerfile:

```
 1 FROM jenkins/jenkins:jdk11
 2 USER root
 3
 4 # Download Android SDK
 5 ENV SDK_URL="https://dl.google.com/android/repository
 6    /commandlinetools-linux-8512546_latest.zip" \
 7    ANDROID_HOME="/usr/local/android-sdk" \
 8    ANDROID_VERSION=30 \
 9    ANDROID_BUILD_TOOLS_VERSION=30.0.2
10
11 RUN mkdir "$ANDROID_HOME" .android \
12    && cd "$ANDROID_HOME" \
13    && curl -o sdk.zip $SDK_URL \
14    && unzip sdk.zip \
15    && rm sdk.zip \
16    && mkdir "$ANDROID_HOME/licenses" || true \
17    && echo "24333f8a63b6825ea9c5514f83c2829b004d1fee" >
          "$ANDROID_HOME/licenses/android-sdk-license"
18 # Install Android Build Tool and Libraries
19 RUN $ANDROID_HOME/cmdline-tools/bin/sdkmanager --update
       --sdk_root="$ANDROID_HOME"
20 RUN $ANDROID_HOME/cmdline-tools/bin/sdkmanager
       --sdk_root="$ANDROID_HOME"
       "build-tools;${ANDROID_BUILD_TOOLS_VERSION}" \
21    "platforms;android-${ANDROID_VERSION}" \
22    "platform-tools"
```

Figure 4.1: Dockerfile to setup Jenkins, download and configure the Android SDK.

The *FROM* keyword in 4.1 line 1 defines an official Jenkins image with JDK version 11 as the basis for this Docker image. 4.1 lines 5-9 define environment variables, required for the download and installation of the Android SDK. First, the directory structure is created (see 4.1 line 11-12). Subsequently, the Android SDK is downloaded (see 4.1 line 13) as a ZIP file. After the file is unzipped (see 4.1 line 14), the ZIP file is no longer needed and therefore deleted (see 4.1 line 15). For legal reasons a license has to be accepted by creating an *android-sdk-license* file containing a license key (see 4.1 line 17). Subsequently, the Android SDK tools are updated (see 4.1 line 19) and finally installed (see 4.1 line 20-22). The Docker build command *docker build Dockerfile* builds a Docker image from the Dockerfile 4.1. Once the Docker image is built, it can be run using *docker run*. The Jenkins instance is now up and running.

4.1.3 Pipeline

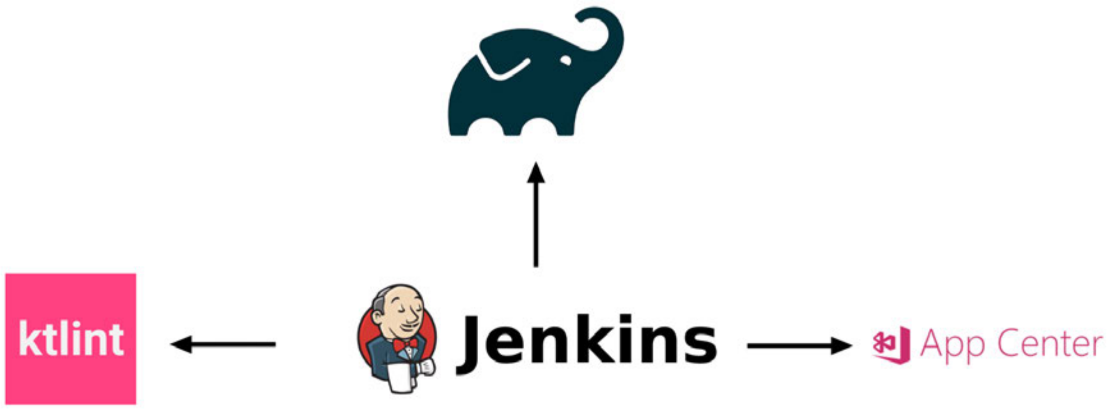
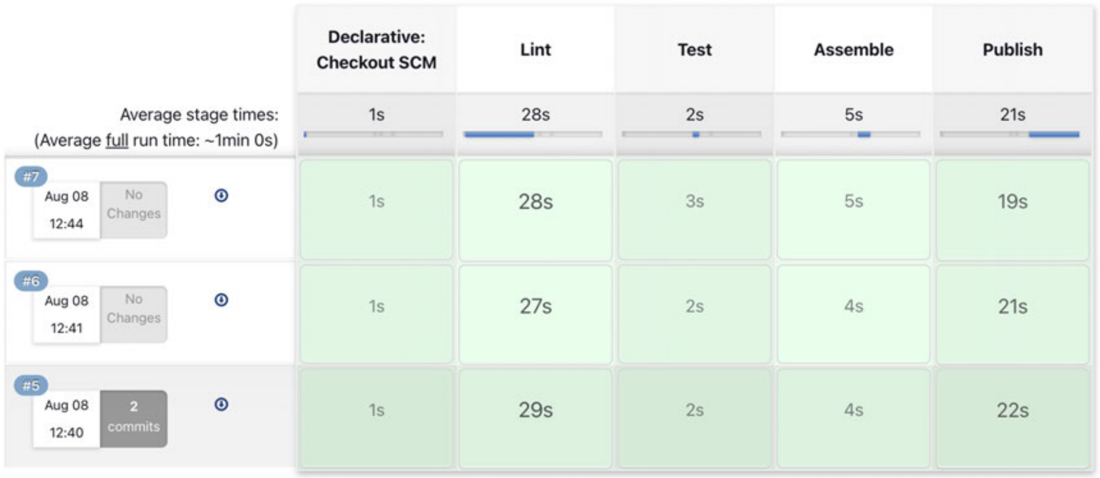


Figure 4.2: Jenkins as an orchestration tool



	Declarative: Checkout SCM	Lint	Test	Assemble	Publish
Average stage times: (Average full run time: ~1min 0s)	1s	28s	2s	5s	21s
#7 Aug 08 12:44 No Changes	1s	28s	3s	5s	19s
#6 Aug 08 12:41 No Changes	1s	27s	2s	4s	21s
#5 Aug 08 12:40 2 commits	1s	29s	2s	4s	22s

Figure 4.3: Jenkins CI/CD pipeline. This pipeline builds an Android app in five stages. First it checks out the repository, secondly runs a linter [40], thirdly runs unit tests, fourthly assembles the Android app and finally publishes the app to Microsoft App Center [28].

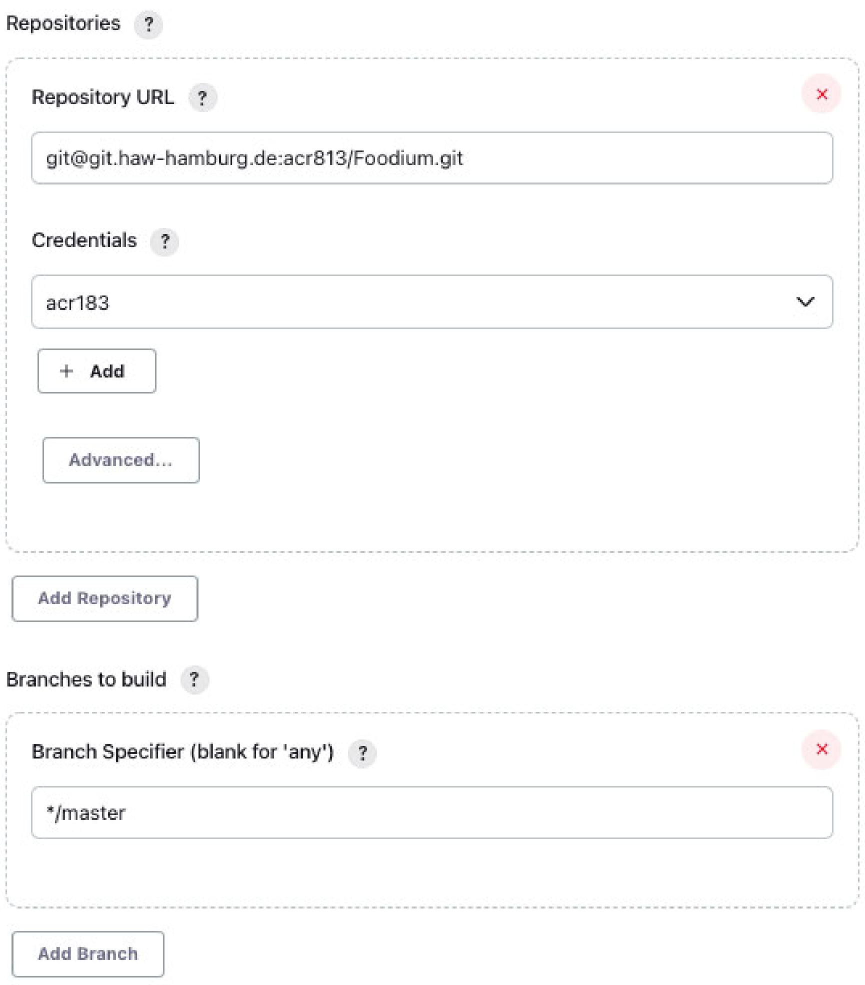
In this project Jenkins orchestrates ktlint [42], build and test process using Gradle [27]. Finally the Android application is published to Microsoft App Center [28] (see 4.2).

```
pipeline {
   agent any

   stages {
      stage('Lint') {
         steps {
            sh './gradlew lint'

            archive(includes: '*/lint*.html')

            publishHTML(target: [
                  allowMissing    : false,
                  alwaysLinkToLastBuild: false,
                  keepAll         : true,
                  reportDir       : 'app/build/reports',
                  reportFiles     : 'lint-results.html',
                  reportName      : "Lint Report"
            ])
         }
      }

      stage('Test') {
         steps {
            sh './gradlew test'

            archive(includes: '*/index.html')

            publishHTML(target: [
                  allowMissing    : false,
                  alwaysLinkToLastBuild: false,
                  keepAll         : true,
                  reportDir       :
                     'app/build/reports/tests/testReleaseUnitTest',
                  reportFiles     : 'index.html',
                  reportName      : 'Test Report'
            ])
         }
      }

      stage('Assemble') {
         steps {
            sh './gradlew assembleRelease'
         }
      }

      stage('Publish') {
         environment {
            APPCENTER_API_TOKEN = credentials('APPCENTER_API_TOKEN')
         }
         steps {
            appCenter apiToken: APPCENTER_API_TOKEN,
                  ownerName: 'Nick.Koops-haw-hamburg.de',
                  appName: 'Foodium',
                  pathToApp: '*/app-release-unsigned.apk',
                  distributionGroups: 'Collaborators'
         }
      }
   }
}
```

Figure 4.4: Jenkins pipeline in Groovy DSL

The CI/CD phases are chained using a pipeline (see 4.3) within a single workflow definition. The pipeline is defined in a Jenkinsfile using the Groovy DSL as seen in 4.4. The stages are enclosed in a pipeline block (see line 1), defining the beginning and ending of the pipeline. Line 2 executes the pipeline on any available Jenkins agent.

SCM checkout


Repositories

Repository URL: git@git.haw-hamburg.de:acr813/Foodium.git

Credentials: acr183

Branches to build

Branch Specifier (blank for 'any'): */master

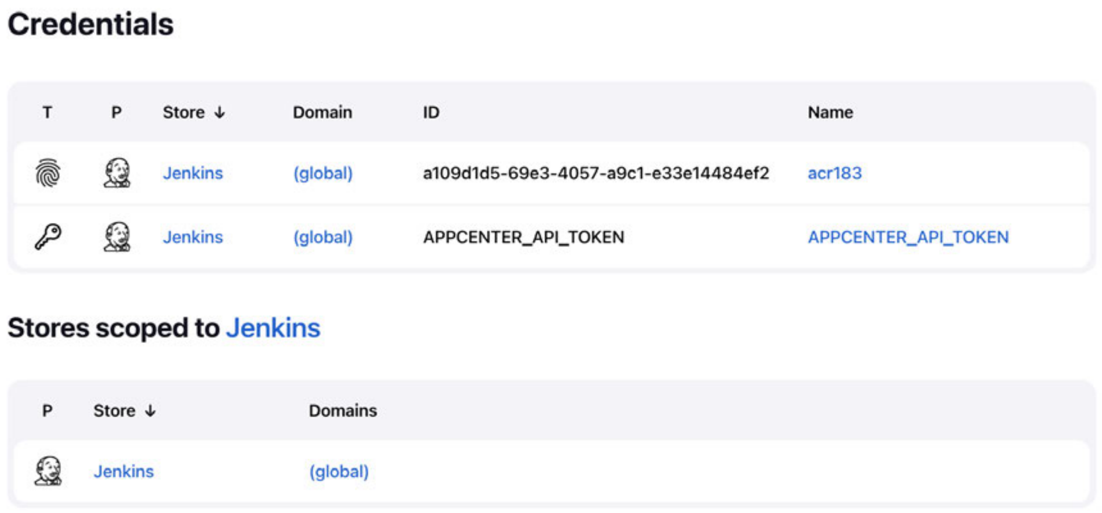
Figure 4.5: SCM checkout in the Jenkins build configuration

As seen in 4.3, the first stage is the Source Control Management (SCM) checkout. The repository URL points to the GitLab repository[1] in which a fork of the Foodium app is

[1]https://git.haw-hamburg.de/acr813/Foodium

located. The URL to the GitLab repository and the credentials for authentication are configured inside the web UI (see 4.5). The SSH username and private key are stored in Jenkins using Jenkins credentials plugin (see 4.6) [18]. Jenkins credentials can be used to store

- Secret text - a token such as an API token (e.g. a GitHub personal access token),
- Username and password,
- Secret file - which is secret content in a file,
- SSH Username with private key - an SSH public/private key pair,
- Certificate - a PKCS#12 certificate file and optional password, or
- Docker Host Certificate Authentication credentials.



Credentials

T	P	Store ↓	Domain	ID	Name
		Jenkins	(global)	a109d1d5-69e3-4057-a9c1-e33e14484ef2	acr183
		Jenkins	(global)	APPCENTER_API_TOKEN	APPCENTER_API_TOKEN

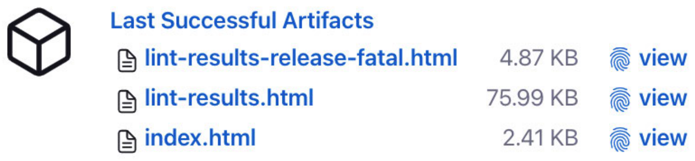
Stores scoped to Jenkins

P	Store ↓	Domains
	Jenkins	(global)

Figure 4.6: Jenkins credentials configuration. The upper credentials contain the SSH username and private key for the GitLab repository. The second credentials are a secret text containing the Microsoft App Center API token, a unique identifier for the application generated by App Center, needed for the upload. Both credentials are stored globally inside Jenkins.

Lint

The stages of the pipeline are defined within the stages block (see 4.4 line 4). Each individual stage contains the steps to be executed (see 4.4 line 6-18). After the SCM checkout, as defined in 2.2, the pipeline performs a static code analysis using ktlint (see 4.4 line 7). The linter is executed via a Gradle task. The linter puts out a HTML file containing the resulting report. In line 9 the lint report is saved permanently using the archive plugin, so they can be viewed and downloaded later. Archived build artifacts are kept as long as the build itself [18]. Once the report HTML file is archived, it is published to the pipeline's overview page, making it accessible to the development team (see 4.7). The HTML is published with the HTML Publisher plugin (see 4.4 line 11-18) [13].



Last Successful Artifacts

lint-results-release-fatal.html	4.87 KB	view
lint-results.html	75.99 KB	view
index.html	2.41 KB	view

Figure 4.7: Published lint and test report HTML files

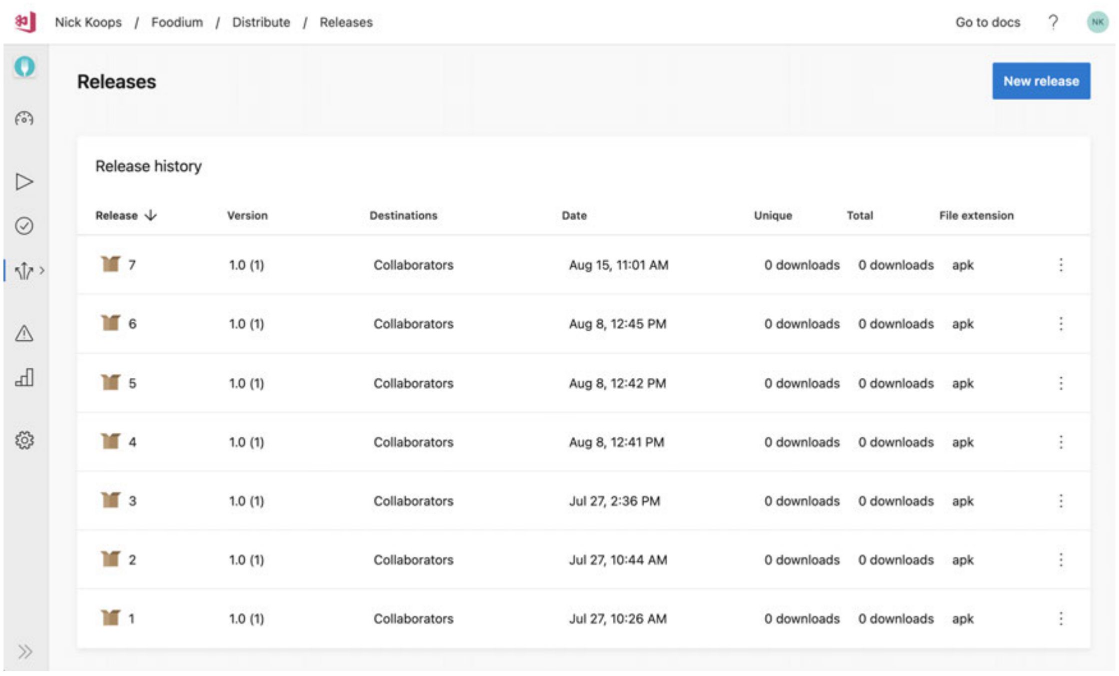
Test

The test stage (see 4.4 line 22-37) is equivalent to 4.1.3 apart from the Gradle task being executed and the HTML file being uploaded. The Gradle task test in line 24 runs all unit tests and results in an index.html containing the test report. The test report HTML is published (see 4.7) and archived in the same way as the lint report.

Assemble

Once the lint and test quality gates are fulfilled the Foodium application itself is build using the Gradle command in 4.4 line 41. The Gradle command `assembleRelease` assembles the release variant of the Android Application. Its artifact is an Android package (APK), which is an archive containing the contents of an Android app that are required at runtime. This is the file, wich can be installed on Android devices [8].

Publish



Releases

Release history

Release	Version	Destinations	Date	Unique	Total	File extension
7	1.0 (1)	Collaborators	Aug 15, 11:01 AM	0 downloads	0 downloads	apk
6	1.0 (1)	Collaborators	Aug 8, 12:45 PM	0 downloads	0 downloads	apk
5	1.0 (1)	Collaborators	Aug 8, 12:42 PM	0 downloads	0 downloads	apk
4	1.0 (1)	Collaborators	Aug 8, 12:41 PM	0 downloads	0 downloads	apk
3	1.0 (1)	Collaborators	Jul 27, 2:36 PM	0 downloads	0 downloads	apk
2	1.0 (1)	Collaborators	Jul 27, 10:44 AM	0 downloads	0 downloads	apk
1	1.0 (1)	Collaborators	Jul 27, 10:26 AM	0 downloads	0 downloads	apk

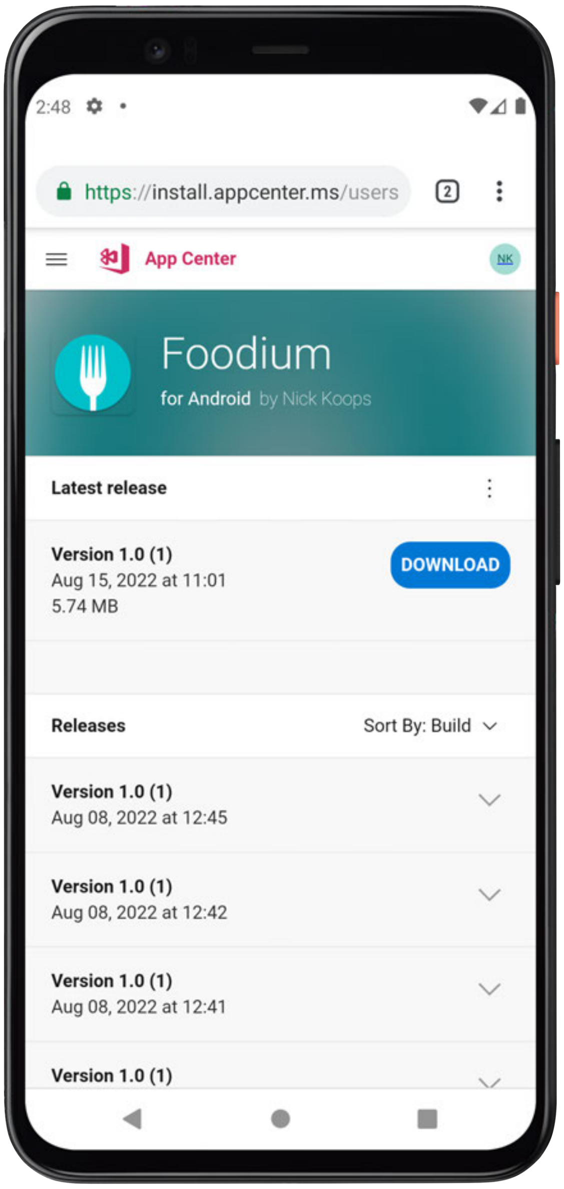
Figure 4.8: Releases of the Foodium Android app in Microsoft App Center

Subsequently the APK file has to be published to a platform where testers can download it for possible manual tests, as defined in 2.2. Microsoft App Center is a platform built for distributing mobile applications, such as Android and iOS apps, to selected groups of internal and external testers. Testers are invited by email and can be assigned to one ore more groups, as seen in 4.8 [28]. In order to upload an application to App Center, App Center generates unique API tokens, which are used to identify the application in App Center. The API token is stored in Jenkins as a secret text with the aforementioned Jenkins credentials plugin. As seen in 4.4 line 46-48 the App Center API token is retrieved from credentials and assigned to an environment variable called `APPCENTER_API_TOKEN`. To upload the application via Jenkins there is an App Center Jenkins plugin [26], which is used in 4.4 line 50-55. Its arguments are

- the apiToken,
- ownerName - owner of the application,
- pathToApp - path of the artifact to be uploaded and

- distributionGroups - groups to which the application is distributed.

In this case the previously assembled APK is uploaded and distributed to a default distribution group called Collaborators, although more specific groups of testers can be added, i.e. internal and external testers.



Version 1.0 (1)
Aug 15, 2022 at 11:01
5.74 MB

Version 1.0 (1)
Aug 08, 2022 at 12:45

Version 1.0 (1)
Aug 08, 2022 at 12:42

Version 1.0 (1)
Aug 08, 2022 at 12:41

Version 1.0 (1)

Figure 4.9: Downloading the application from App Center on an Android phone

App Center provides a web UI with all available builds for the requested application. The application can be downloaded by testers and installed on an Android device (see 4.9).

4.2 Dagger

Dagger is a portable development kit for CI/CD with the objective to create CI/CD pipelines running in any Docker-compatible runtime. Dagger achieves this by unifying development and CI environments into a pipeline, that is independent from a CI environment and can therefore be ported any CI environment and even be run locally. This independence is made possible by two technologies:

- CUE - a configuration language developed as the result of Borg Configuration Language (BCL) [53].
- The configuration is executed in BuildKit [4]. BuildKit is the backend building images from Docker files [10].

Dagger does not replace a CI platform, but rather adds a portable layer on top of it. Therefore Dagger runs on all Docker capable CI products, such as GitHub actions, TravisCI, CircleCI, GitLab and Jenkins [7]. This sections first introduces the core concepts of the CUE language, then explains how a Dagger pipeline is used to build a sample project on a local machine and how the Dagger pipeline is executed in a CI/CD environment.

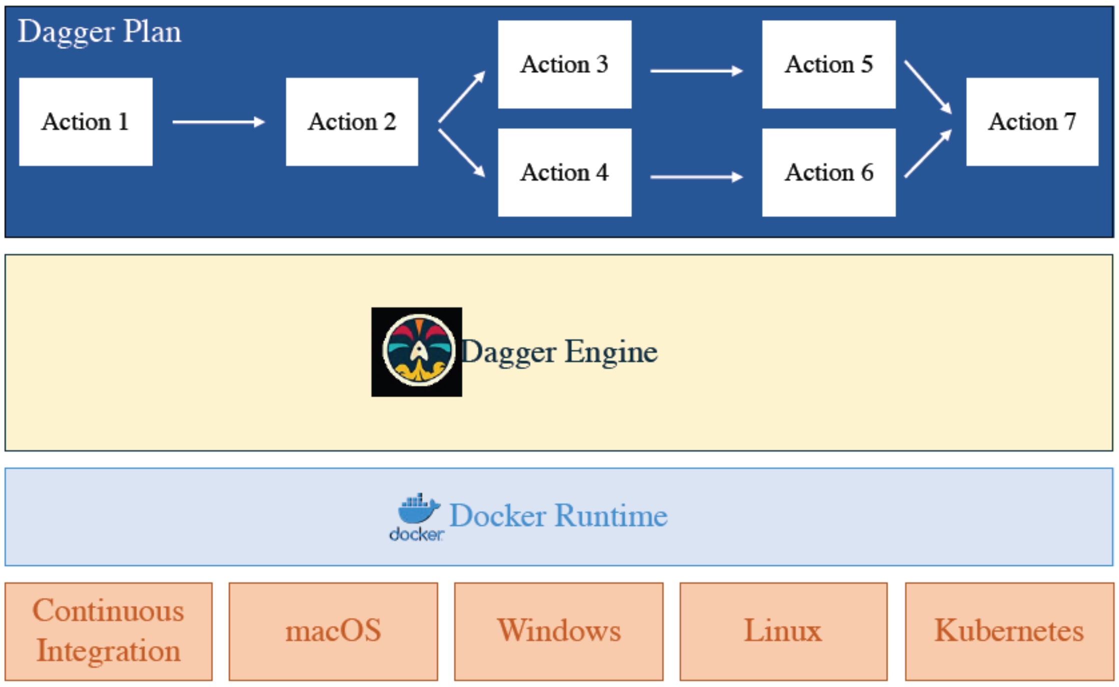


Figure 4.10: A layer model of Dagger. Docker runtime (light blue) can run on any compatible platform (orange). This may be a CI platform, such as Jenkins or CircleCI, an operating system, such as macOS or Linux, or on a container orchestration system, such as Kubernetes. The Dagger engine (yellow) runs on the Docker runtime. Dagger plans (dark blue) are executed by the Dagger engine. A Dagger plan is composed of Dagger actions (white) (see 4.2.2).

4.2.1 CUE

This section explains the concepts needed to understand the CUE language. The CUE configuration language consists of these core concepts [7]:

1. CUE is a superset of JSON
2. Types are values
3. Concrete values
4. Constraints, definitions and schemas
5. Unification

Further details can be found in the Dagger documentation [7].

CUE is a superset of JSON

This means, that every JSON can be expressed in CUE, but not everything in CUE can be expressed in JSON. As a consequence every valid JSON is valid CUE.

```
{
   "Alice": {
      "Name": "Alice Miller",
      "Age": 42
   }
}

Alice: Name: "Alice Miller"
```

Figure 4.11: Example of JSON as valid CUE code

The example given in 4.11 declares the top-level key *Alice* twice: First in a more verbose JSON style and secondly in the shorthand CUE style. The shorthand style can be used, when the declaration only targets one key within the object.

Types are values

In the context of Dagger fields and their values will be passed as output to other CI/CD platforms and within the Dagger plan from one action as output to another action as input. Since most CI/CD platforms expect these values to conform to a certain schema where a field has a type and is potentially constrained by functions such as min, max, enums, regular expressions, etc.

```
Alice: {
   Name: string // type as value
   Age: int // type as value
}

Alice: {
   Name: "Alice Miller"
   Age: 42
}
```

Figure 4.12: Field *Name* is defined as a *string* and *Age* as *int*.

4.12 defines the fields *Name* as a *string* and *Age* as an *int*. Since *string* and *int* are not within quotes, they are not concrete values. CUE now enforces these values to conform to the defined type, i.e. an *int* value cannot be assigned to the field *Name*.

Concrete values

In CUE fields are validated against a well-defined schema. In case defined fields are not optional and do not have a concrete value, CUE throws an error.

```
Alice: {
   Name: string
   Age: int
}

Alice: {
   Name: "Alice Miller"
   // Age is incomplete, since no concrete value is defined
}
```

Figure 4.13: No concrete value is given for *Age*

Definitions

Definitions ensure that each object implementing the definition satisfies the defined schema.

```
#Person: {
   Name: string
   LastName: string
   Email: string
   Age?: int
}

Alice: #Person & {
   Name: "Alice"
   LastName: "Miller"
   Email: "alice@miller.com"
   // Age ist optional
}
```

Figure 4.14: An example of a definition *#Person* constraining the *Person* object

4.14 defines a *#Person* definition, as indicated by the #. This constrains the *Person* object to a set of specific fields and their values. By default objects conforming to a definition are closed, so they only contain fields specified in the definition. Fields can also be optional, denoted by the *?* after the name of the field. Definitions are not exported to the final output. In order to get concrete output, an object (in this case *Alice*) has to conform to a definition (*#Person*). The definition is then unified with the object using the *&* operator.

Unification

Unification allows CUE to simultaneously define constraints and compute concrete values.

```
1  import (
2    "strings" // import builtin package
3  )
4
5  #Person: {
6    // further constrain to a min and max length
7    Name: string & strings.MinRunes(3) & strings.MaxRunes(22)
8
9    // basic regex testing the correct syntax for an e-mail
10   Email: =~"^([a-zA-Z0-9]*)@([a-zA-Z0-9]*).([a-zA-Z0-9]*)$"
11
12   // constrain to natural ages
13   Age?: int & >0 & <140
14 }
15
16 Alice: #Person & {
17   Name: "Alice Miller"
18   Email: "alice@miller.com"
19   Age: 42
20 }
21
22 // output in YAML:
23 Alice:
24   Name: Alice Miller
25   Email: alice@miller.com
26   Age: 42
```

Figure 4.15: Unifying the *#Person* with the *Alice* object

The output of the unification is the product of unifying the constraints and types of *#Person* definition with the concrete values of the *Alice* object. The unified object can the be output into several formats, i.e. YAML (see 4.15 lines 23-26).

4.2.2 Dagger pipeline

A Dagger pipeline is built using CUE and its features explained in 4.2.1. Every Dagger configuration starts with a plan, which is derived from *dagger.#Plan* [7] (see 4.16 line 11). Consider this plan 4.16 for the evaluation of Dagger.

```
package main

import (
      "dagger.io/dagger"
      "dagger.io/dagger/core"
      "universe.dagger.io/alpine"
      "universe.dagger.io/bash"
      "universe.dagger.io/docker"
)

dagger.#Plan & {
      //Write the output of the gradle build to the client dev machine
      client: {
            filesystem: {
                  "./build": write: contents: actions.build.gradle.export.directories."/build"
            }
            env: PAT: dagger.#Secret
      }

      actions: {
            build: {
                  // core.#Source lets you access a file system tree (dagger.#FS)
                  // using a path at "." or deeper (e.g. "./foo" or "./foo/bar") with
                  // optional include/exclude of specific files/directories/globs
                  checkoutCode: core.#Source & {
                        path: "."
                  }
                  // Build an alpine image with gradle and bash installed
                  base: alpine.#Build & {
                        packages: {
                              "gradle": _
                              "bash": _
                        }
                  }
                  // Pull image with OpenJDK from Docker Hub
                  jdk: docker.#Pull & {
                        source: "eclipse-temurin:11"
                  }
                  // User docker.#Run to export openjdk dir from jdk container
                  javaHome: docker.#Run & {
                        input: jdk.output
                        export: {
                              directories: "/opt/java/openjdk": _
                        }
                  }
                  // Copy the openjdk contents to the alpine gradle image
                  copyJava: docker.#Copy & {
                        input: base.output
                        contents: javaHome.export.directories
                              ."/opt/java/openjdk"
                  }
            // Finally copy the source code into the image
            image: docker.#Copy & {
                input:  copyJava.output
                contents: checkoutCode.output
            }
            // Runs a bash script in the input container
            // in this case `gradlew`
            // export the /build directory to write to client machine
            gradle: bash.#Run & {
                input: image.output
                script: contents: """
                    ./gradlew build
                    """
                export: directories: "/build": _
            }
        }

        test: {
            test: bash.#Run & {
                input: build.gradle.output
                script: contents: """
                    ./gradlew test
                    """
                export: directories: "/build/test-results": _
            }
        }

        push: {
            // pull docker image with JRE
            base: docker.#Pull & {
                source: "eclipse-temurin:11.0.16.1_1-jre"
            }
            // copy artifacts into the docker container
            baseWithArtifacts: docker.#Copy & {
                // copy artifacts
                _artifacts: core.#Source & {
                    path: "."
                    include: ["*.jar"]
                }
                input:  base.output
                contents: _artifacts.output
            }
            // set docker config
            imageToPush: docker.#Set & {
                input: baseWithArtifacts.output
                config: {
                    workdir: "/"
                    expose: "8080" : _
                    entrypoint: ["java", "-jar", "blog-0.0.1-SNAPSHOT.jar"]
                }
            }
```

```
103                   // push docker image to GitHub
104                   push: docker.#Push & {
105                         image: imageToPush.output
106                         dest:
107                         "ghcr.io/nickkoops/spring-boot-kotlin:latest"
108                         auth: {
109                               username: "nickkoops"
110                               secret: client.env.PAT
111                         }
112                   }
113             }
114       }
115 }
```

Figure 4.16: Dagger Plan

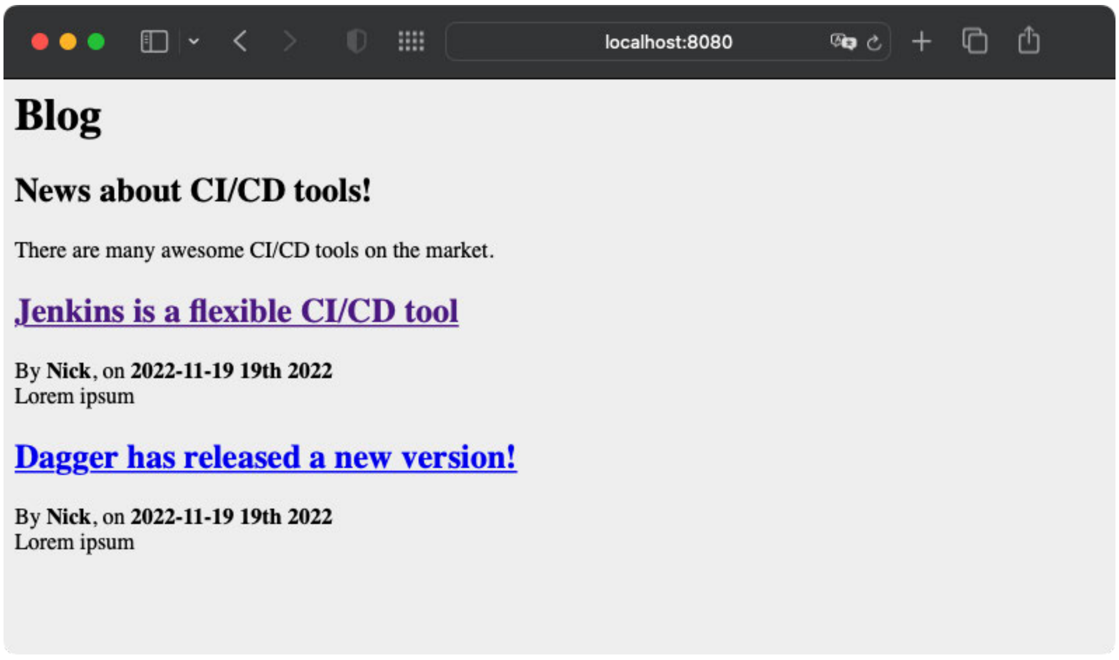
In general, a plan can:

- read from and write to the clients *filesystem* (see 4.16 line 14),
- read *env* variables (see 4.16 line 17) and
- declare *actions*, such as *build* (4.16 line 21), *test* (4.16 line 69) and *push* (4.16 line 79).

This plan writes the output of gradle build from the docker image to the client machine (see 4.16 line 14). Additionally an environment variable *PAT* (Personal Access Token) is read from the client machine (see 4.16 line 17). The content of the environment variable is used to authorize pushing a Docker image to GitHub (see 4.16 line 110). Actions are declared inside of the action block (4.16 line 20). The plan declares three actions: *build* (4.16 line 21), *test* (4.16 line 69) and *push* (4.16 line 79). The aim of this plan is to build a web application using Gradle, run its tests and finally build a deployable Docker image containing the application and the required environment.

Build action

The *build* action builds the demo project, which is a simple web application [22], using gradle.



Blog

News about CI/CD tools!

There are many awesome CI/CD tools on the market.

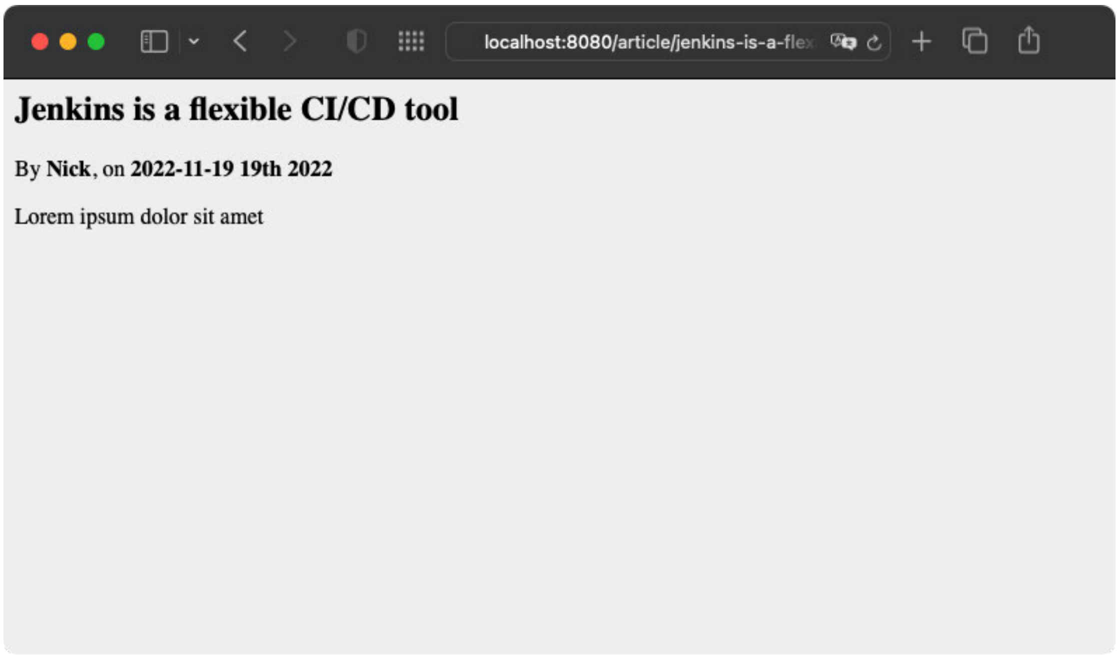
Jenkins is a flexible CI/CD tool

By **Nick**, on **2022-11-19 19th 2022**
Lorem ipsum

Dagger has released a new version!

By **Nick**, on **2022-11-19 19th 2022**
Lorem ipsum

Figure 4.17: A screenshot of the software used to evaluate Dagger. This is the blog's homepage.



Jenkins is a flexible CI/CD tool

By **Nick**, on **2022-11-19 19th 2022**

Lorem ipsum dolor sit amet

Figure 4.18: When the user clicks on an article on the homepage, the article opens in a new detailed page.

The application is a basic Spring Boot web application based on an open source project [22] written in Kotlin and built using Gradle. The application is a blog containing blog entries. The blog itself has a title, subtitle and two blog entries (see 4.17). As seen in 4.18 each blog entry has a title, headline, content and an author. The content of the blog is for demonstration purposes only and chosen freely. To build a Gradle project inside a docker container, the container needs to include a Java development kit (JDK), a Bash command line and Gradle itself. The Dagger *Alpine* package contains a definition *alpine.#Build* (4.16 line 29), which takes packages (in this case Gradle and Bash). Alpine is minimal Linux distribution, which is typically used as a base image for utilities and production application [1]. Through unification the *base* object now contains an Alpine docker image with Gradle and Bash already installed. The *docker.#Pull* definition (4.16 line 36) pulls an eclipse-temurin docker image. This image contains a JDK [11], which is subsequently exported from its container (4.16 line 40) and copied into the Alpine container (4.16 line 47). Finally the source code is copied from the host machine into the Alpine container using *docker.#Copy* (4.16 line 53). The source code is then compiled using Gradle via a Bash command (4.16 line 60). This step also exports the resulting build artifacts back onto the host machine (4.16 line 65).

Test action

The *test* action (see 4.16 line 69) gets the output from the *build.gradle* action as input (4.16 line 71) to execute the Gradle test task in the same environment as the code has been compiled in. Therefore Gradle does not have to recompile the code which in return shortens the *test* task's execution time.

Push action

The *push* action builds a Docker image containing a Java runtime environment (JRE) and the artifacts from the previous *build* action. Subsequently the image is pushed to a remote GitHub Container registry. The web application requires a JRE to be executed. Therefore the *push* action pulls a Docker image containing a JRE [11] (4.16 line 82). Subsequently the application artifact is copied from the build machine into the docker container. To achieve this, an *_artifacts* object implementing *core.#Source* is defined inside of the *baseWithArtifacts* object (4.16 line 87), to make object containing the source only visible inside of the object requiring the source code. The source code is then copied into the container using *docker.#Copy* (4.16 line 85). To execute the application on container start, the path to the application inside of the container is set as an entrypoint using *docker.#Set* (4.16 line 95). Since the web application uses port 8080, this port of the container gets exposed. Finally the Docker image is pushed to a remote GitHub Container registry using *docker.#Push* (4.16 line 104). The destination URL consists of *ghcr.io/OWNER/IMAGE_NAME:latest*, where *OWNER* is substituted with the name of the repository's owner, in this case myself, and *IMAGE_NAME* is substituted with the name of the Docker image, in this case *spring-boot-kotlin* (4.16 line 106). Authentication is done via the username and previously generated *PAT* (4.16 line 109 and 110), which is saved as an environment variable on the local machine. The PAT is a *dagger.#Secret*. Secrets are used to pass confidential information, such as passwords, API keys, SSH keys, etc. within the plan. Secrets are managed by the Dagger runtime and are never exposed as plaintext in logs, written into the filesystem or cached [7].

Local plan execution

To execute the plan locally, docker has to be installed and running [7]. The plan is executed via the command line with *dagger do* followed by the name of the action, in this case *build*.

```
$ dagger do build
[x] actions.build.gradle.script                         0.0s
[x] actions.build.jdk                                   1.7s
[x] actions.build.checkoutCode                          6.6s
[x] actions.build.base                                  2.0s
[x] actions.build.javaHome                              0.0s
[x] actions.build.javaHome.export                       0.0s
[x] actions.build.copyJava                              0.0s
[x] actions.build.image                                 0.6s
[x] actions.build.gradle                                102.7s
[x] actions.build.gradle.export                         0.4s
[x] client.filesystem."./build".write                   0.8s
```

Figure 4.19: Output from command *dagger do build*.

The output 4.19 lists the executed steps. The [x] to the left of the executed steps denotes, that is has been executed successfully. The time in seconds to the right of the executed step is the execution time. Some steps have a 0.0s execution time, since they have been cached in prior executions.

Plan execution in a CI/CD environment

Dagger depends on a Docker environment. Therefore the CI/CD environment has to support Docker. There is a list of currently supported CI/CD environments on the Dagger website [7]. CircleCI was chosen, since it is listed as Dagger compatible, offers free execution in the cloud and a configurable built in Docker environment [6]. The configuration is contained in a *.circleci/config.yml* file within the repository. This CircleCI configuration is based on the configuration on the Dagger website.

```
version: 2.1

jobs:
   install-and-run-dagger:
   docker:
      - image: cimg/base:stable
   steps:
      - checkout
      - setup_remote_docker:
         version: "20.10.14"
      - run:
         name: "Install Dagger"
         command: |
         cd /usr/local
         wget -O - https://dl.dagger.io/dagger/install.sh | sudo sh
         cd -
      - run:
         name: "Run Dagger"
         environment:
            PAT: ghp_sasjlvb820qpzher234chvhi34b5b3kbn225
         command: |
         dagger do push --log-format plain

workflows:
   dagger-workflow:
   jobs:
      - install-and-run-dagger
```

Figure 4.20: The CircleCI configuration used for this project.

This configuration contains one job *install-and-run-dagger* (4.20 line 4). This job is executed inside a *cimg/base:stable* Docker environment, which is an Ubuntu based Docker image, created to serve as the base for CI/CD builds (4.20 line 6). It has a minimum set of required tools, such as Git and Docker [5]. After the repository checkout (4.20 line 8), the *setup_remote_docker* step (4.20 line 9) creates a remote Docker environment in which subsequent *docker run* commands are executed. The *Install Dagger* step downloads and executes the Dagger installation script via a *docker run* command (4.20 line 13). The next *Run Dagger* (4.20 line 18) step runs the Dagger plan the same way as on a local machine (4.20). The *–log-format plain* option prints the log in a verbose format. The *PAT* environment variable (4.20 line 20) is required for pushing the final Docker image into the GitHub Container registry as explained in 4.2.2. PAT's value is altered to not expose the actual value publicly.

4.3 Comparison

As mentioned in the introduction to this chapter, the Jenkins and Dagger are compared two each other by these criteria in order to answer RQ:

- portability,
- reusability,
- extendability and,
- scalability.

The criteria are derived from each tool's strength promoted by their developers: Dagger particularly emphasizes its portability and reusability [7]. Jenkins emphasizes its extendability due to its plugin architecture [31]. Scalability is included to highlight the lack of developer experience and literature with Dagger compared to Jenkins.

4.3.1 Portability

[44, 25] define portability as an ability of a software to run on different computing systems, such as hardware, software or a combination of the two. In the context of CI/CD systems its the ability to execute the pipeline on different machines being built on different hardware and operating systems. Jenkins itself can be executed on any system,

which has a Java runtime [31]. However, Jenkins relies on the system it is executed on to provide the tools necessary for the pipeline build, i.e. build tools. The Jenkins pipeline syntax offers a *tools* section, which can automatically install a specified version of *maven*, *jdk* and *gradle* [18]. Since the syntax is limited to these three tools, any other tool, such as the Android SDK Platform-Tools required to build the Android app without Android Studio installed [8] as explained in 4.1, has to be installed separately. To provide a consistent environment for Jenkins, Jenkins can be run inside a Docker container, as explained in 4.1.2. This way, the necessary build tools, environment variables, etc. are held consistent when switching systems.
Dagger solves the issue of inconsistent build environment in two ways: Firstly, every Dagger plan is executed inside of a Docker environment. Consequently this environment stays consistent, even when switching systems, although dependencies on the local machine declared inside of the *client* field, i.e. environment variables, (see 4.2.2) have to be provided by the system. Secondly, since Dagger does not replace a CI/CD platform, but rather builds on top of it, a Dagger plan can be ported from a Docker compatible CI/CD platform A to another Docker compatible platform B, without rewriting any of the plans' code. Relying on Docker also means being limited to software, which can be executed inside a Docker container. This brings inherent limitations, such as not being able to build with proprietary and platform specific build systems, i.e. Xcode on macOS [24]. Consequently macOS and iOS applications cannot be build using Dagger.

4.3.2 Reusability

In the general context of software development reusability means using existing software to develop new software and thereby copying a module to be used in other projects than the one it was initially developed for [43]. This section focuses on the use case of reusing code, by moving it out of the pipeline and into a separate repository. Jenkins has a technology called Shared Libraries. Shared Libraries are designed to share common parts and patterns of pipelines between them, in order to reduce redundant code. The Jenkins documentation [18] outlines how to use Shared Libraries. Shared Libraries can run any methods in Java, Groovy, Jenkins internal APIs, Jenkins plugins or third-party libraries. To put it briefly, the purpose of Shared Libraries is to enhance the Jenkinsfile, by reducing its' complexity and encapsulate data into separate files, classes, etc., that would otherwise be part of it. A Shared Libraries' functionality is limited to the Jenkinsfile, which means a Shared Library cannot do anything, that the Jenkinsfile cannot do on its own. Shared Libraries are usually in a separate source code repository from the main Jenkinsfile using the Shared Library. A Shared Library can be retrieved using the SCM plugin and then be used in the main Jenkins pipeline. Shared Libraries are accessible as global variables to the pipeline. Therefore every global variable must be in a *vars/* directory.

```
1 def call(String name = 'human') {
2   echo "Hello, ${name}!"
3 }
```

Figure 4.21: A global variables' name must be in camelCase. In this example it is *var/-greet.groovy*. A global variable must also implemented the method *call()* (see line 1).

This global variable is now exposed to the pipeline.

```
 1 @Library('greet') _
 2 
 3 pipeline {
 4    agent any
 5 
 6    stages {
 7       stage('Example') {
 8          steps {
 9             greet('Nick')
10          }
11       }
12    }
13 }
```

Figure 4.22: A pipeline using the global variable *var/greet.groovy*.

This method can now be called in the Jenkinsfile (see 4.22 line 9). Examples beyond this use case can be found in the Jenkins documentation [18].
Dagger uses modules containing packages to encapsulate code from a plan. An example of packages can be seen at 4.16 line 8, where *universe.dagger.io* is the module and *universe.dagger.io/docker* is one of the packages contained in the module. Imported definitions can then be used in the plan (see 4.16 line 40). Consider this exemplary package:

```
package personal

import(
 "universe.dagger.io/alpine"
 "universe.dagger.io/bash"
)

#Run: {
   _img: alpine.#Build & {
      packages: bash: _
   }

   bash.#Run & {
      always: true
      input: _img.output
   }
}
```

Figure 4.23: Declaring a package in an external repository

```
 1 package main
 2
 3 import (
 4    "dagger.io/dagger"
 5    "github.com/your-username/personal" // import personal package
 6 )
 7
 8 dagger.#Plan & {
 9    actions: {
10       // reference #Run definition from personal package imported
            above
11       hello: personal.#Run & {
12          script: contents: "echo \"Hello!\""
13       }
14    }
15 }
```

Figure 4.24: Importing a package from an external GitHub repository

Packages can also be retrieved from external repositories (4.24 line 5). In this example a package called *personal* contains a *#Run* definition (see 4.23), which is referenced in 4.24 line 11. When running the command *dagger project update* inside of the root project directory, Dagger checks out the package from the remote repository and copies the files into the *rootProject/cue.mod/pkg/github.com/your-username/personal* folder. Once the package is imported, it can be used in the plan. This way, details of the implementation in the *personal* package stay hidden from the plan 4.24. Ultimately, both Jenkins Shared Libraries and Dagger packages achieve the same encapsulation of shared code from their pipelines, just adapted for the Groovy and CUE languages respectively.

4.3.3 Extendability

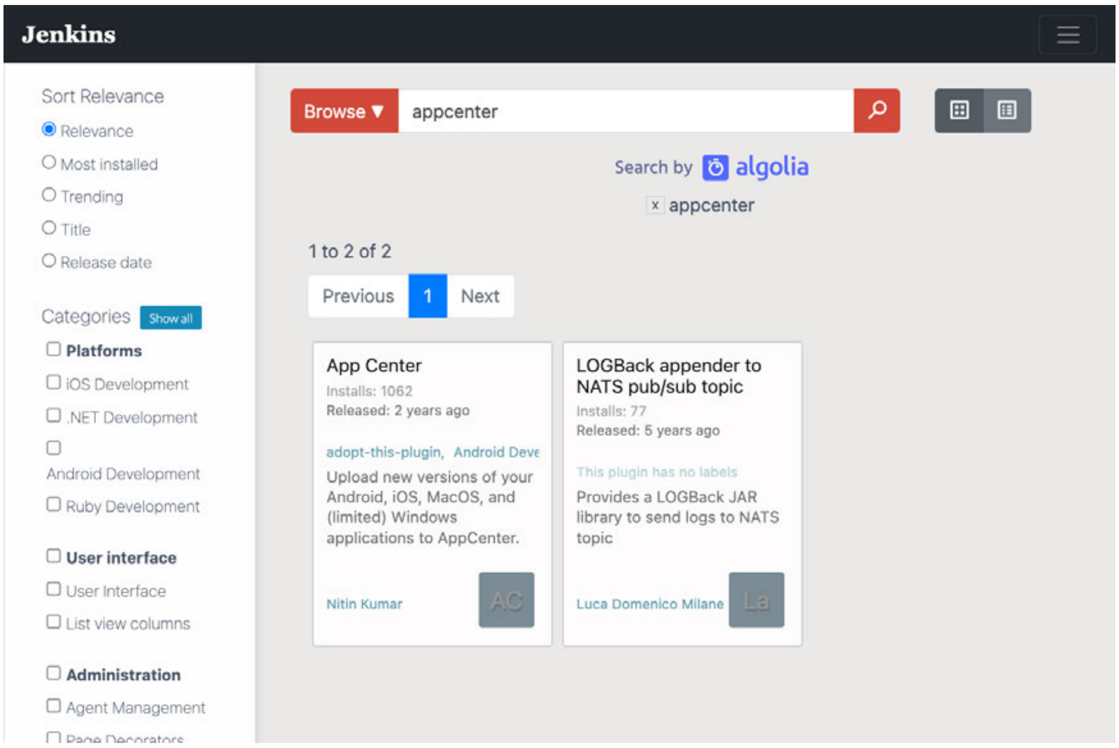
[25] defines extendability as the ease at which a system or component can be modified to increase its functionality. In the context of CI/CD this definition is only part of the equation, since its not only question of how extendable a CI/CD tool is, but also how to discover existing extensions and how to integrate them into a pipeline. Hence, Jenkins and Dagger are compared by the following aspects:

- Discovery of extensions - what are the steps to find an existing extension?
- Integration of extensions - how do you integrate an existing extensions into your own pipeline?
- Creation and distribution - how do you create an extension and how do you distribute it to be reused in other projects?

Although Shared Libraries and plugins have overlapping functionality, plugins are not limited to what the Jenkinsfile can do: Plugins can affect nearly every aspect of Jenkins' behavior, even core functionalities such as Job types, build queues, node monitors, etc. The most common use case of plugins is to offer a specific functionality with a well designed interface, which can be used to integrate the plugin into a Jenkinsfile, such as SCM implementations, i.e. Git and Subversion, or build steps, i.e. providing a convenient UI to view test results or send emails. Dagger does not differentiate between libraries and plugins, but rather uses CUE Modules, which contain CUE Packages, to share common parts between plans and actions [7].

Discovery of extensions

Jenkins plugins are hosted in a repository in the jenkinsci GitHub organization and can be publicly accessed through the plugins index [21].

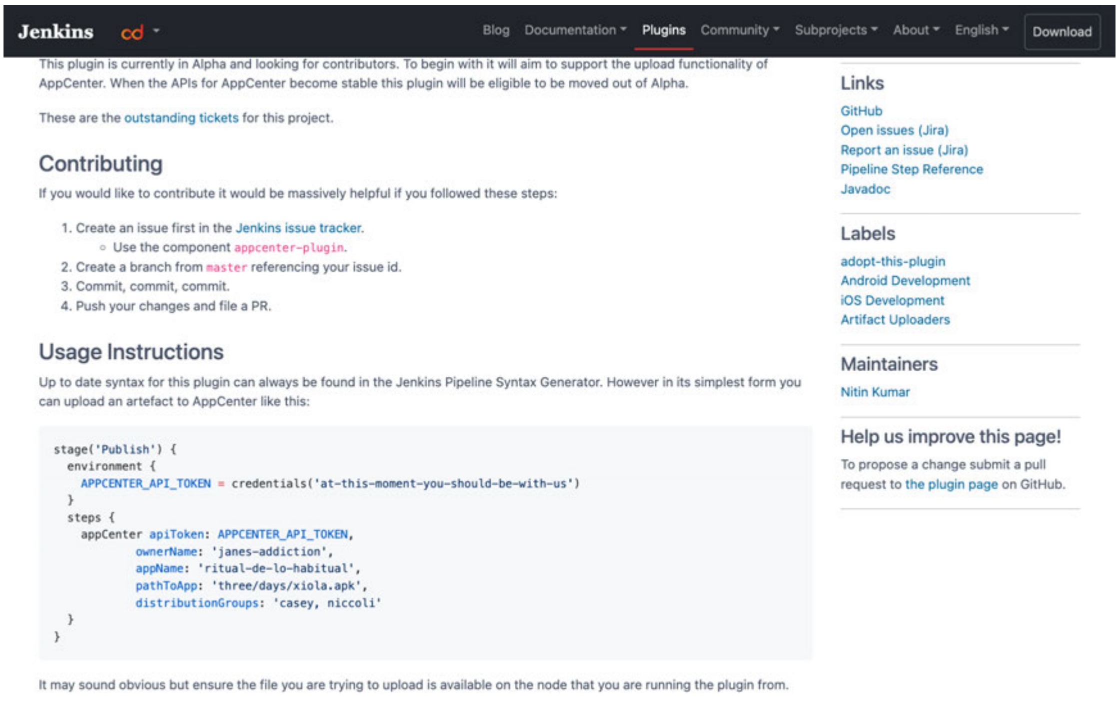


1 to 2 of 2

App Center
Installs: 1062
Released: 2 years ago
adopt-this-plugin, Android Deve
Upload new versions of your Android, iOS, MacOS, and (limited) Windows applications to AppCenter.
Nitin Kumar

LOGBack appender to NATS pub/sub topic
Installs: 77
Released: 5 years ago
This plugin has no labels
Provides a LOGBack JAR library to send logs to NATS topic
Luca Domenico Milane

Figure 4.25: The Jenkins plugins index webpage used to search for plugins. It offers filter and sort features to enhance the search functionality.



This plugin is currently in Alpha and looking for contributors. To begin with it will aim to support the upload functionality of AppCenter. When the APIs for AppCenter become stable this plugin will be eligible to be moved out of Alpha.

These are the outstanding tickets for this project.

Contributing

If you would like to contribute it would be massively helpful if you followed these steps:

1. Create an issue first in the Jenkins issue tracker.
   - Use the component `appcenter-plugin`.
2. Create a branch from `master` referencing your issue id.
3. Commit, commit, commit.
4. Push your changes and file a PR.

Usage Instructions

Up to date syntax for this plugin can always be found in the Jenkins Pipeline Syntax Generator. However in its simplest form you can upload an artefact to AppCenter like this:

```
stage('Publish') {
  environment {
    APPCENTER_API_TOKEN = credentials('at-this-moment-you-should-be-with-us')
  }
  steps {
    appCenter apiToken: APPCENTER_API_TOKEN,
            ownerName: 'janes-addiction',
            appName: 'ritual-de-lo-habitual',
            pathToApp: 'three/days/xiola.apk',
            distributionGroups: 'casey, niccoli'
  }
}
```

It may sound obvious but ensure the file you are trying to upload is available on the node that you are running the plugin from.

Links

GitHub
Open issues (Jira)
Report an issue (Jira)
Pipeline Step Reference
Javadoc

Labels

adopt-this-plugin
Android Development
iOS Development
Artifact Uploaders

Maintainers

Nitin Kumar

Help us improve this page!

To propose a change submit a pull request to the plugin page on GitHub.

Figure 4.26: The documentation page of the App Center plugin

4.25 shows an exemplary search for the App Center plugin used in 4.1.3. Every plugin has a documentation page (see 4.26), which is built after a template from the Jenkins documentation, containing maintainer information, changelogs, contributor documentation and usage instructions [18]. There is no standardized way to discover Shared Libraries, since they are typically kept privately, rather than being open source. The same is true for packages in Dagger. There is no standardized way in Dagger on how to discover existing packages, although you should start your search in the *dagger.io* and *universe.dagger.io* modules. The *dagger.io* contains the core Dagger functionalities, such as plans and git. The *universe.dagger.io* module contains community developed packages, such as Docker and Bash. Compared to Jenkins, there is only a package coding style [7], but no documentation template for packages, leading to very poorly, if at all, documented packages, i.e. the Bash and Alpine packages. In these cases the only way to understand the packages' interface and its functionality is to study the source code.

Integration of extensions

Jenkins plugins can either be installed through the plugin manager in the web UI or the Jenkins CLI. Once a plugin is installed, it can be used. An exemplary use of a plugin can be seen in 4.4 line 50 - 54, where the aforementioned App Center plugin is used in a pipeline step. Another example is the HTML Publisher plugin (see 4.4 line 11 - 18), publishing the build artifacts as seen in 4.7 to the web UI. Since there are very strict guidelines for documenting plugins in the jenkinsci GitHub repository, the integration of most plugins is straight forward. Since Dagger does not differentiate between plugins and libraries, an exemplary integration of packages is already documented in 4.3.2. As mentioned in 4.3.3, there are no guidelines on how to document Dagger packages, making it equally time consuming and complex to understand the code of a package before integrating it. Most of the time, taking a look at examples of other projects integrating the package helped to understand its interface and functionality. Once it is clear how to use a package, a package can be integrated into a plan as seen in 4.3.2.

Creation and distribution

The basis for a Jenkins plugin is generated from Maven Archetypes [15]. Maven Archetypes are templates from which other Maven projects are generated. Jenkins currently offers three plugin archetypes, an empty plugin skeleton, a plugin skeleton with an example configuration and a plugin with an exemplary build step. Once the basis is generated, the plugin can be extended. Like Jenkins itself, plugins are written in Java. A detailed description of how to extend the templates is given in the Jenkins documentation [18]. As mentioned in 4.3.3, Jenkins plugins are published in the jenkinsci GitHub repository. Every plugin in this repository has to be free and open source under an OSI-approved license [20] to ensure source code access and project continuity in case of changing maintainers. To distribute the plugin on GitHub, one has to have a GitHub user account. In order to release a plugin, you have to have a Jenkins community account, which will give you access to Jira issue tracker and the Maven repository. A plugin can either be released via CD in GitHub or manually.
In order to extend Dagger by adding packages to the *universe.dagger.io* module, one has to fork the Dagger GitHub project. From there on, one can perform the desired changes and finally create a pull request back into the Dagger GitHub repository. These pull requests should be small incremental changes, to ensure quick feedback. Additionally,

the pull request and its commits have to follow guidelines to ensure a consistent Git history [7]. This process is the same when contributing to Dagger itself, rather than just adding a package. This process also implies, that everyone, who installs Dagger locally, installs a copy of every package, even the ones not required for the project. Compared to Jenkins' process of creating plugins, Daggers' involves fewer barriers, i.e. it does not require a dedicated release process. This is, since the tools are in two different stages of their development: On the one hand Jenkins is a mature platform with over 2000 plugins available [14], which creates a necessity for well documented publish and release process for plugins. On the other hand, Dagger is in a very early development stage, which requires rapid development and quick feedback, in order to grow the platform.

4.3.4 Scalability

[41] use a CI/CD pipeline at Otto.de for every microservice to frequently and automatically deploy their applications in parallel. Otto.de is one of the largest online shops in Europe, with up to 1 million visitors per day. They chose a microservice architecture mostly for non-functional requirements, such as scalability, performance and fault tolerance. Their pipelines check out every commit and compile, package, deploy and test every commit in an integration stage. Subsequently, the container is deployed to the next stage, called testing, which runs load and integration tests. Before going live there is the pre-live stage, where all services are tested against each other in a suite of manual and automatic tests. In their experiences traditional CI servers, such as Jenkins, reach their limits due to the high number of deployments, since all pipelines across the microservices have to be kept up to date and configured similarly. Therefore, they developed an internal domain-specific language LambdaCD [3]. LambdaCD has a similar infrastructure as code approach compared to Dagger. LambdaCD pipelines are microservices responsible for building, testing and deploying an application. The pipeline runs in the same environment as the other microservices. The pipeline can also be run, tested and debugged locally without a CI platform, just as dagger. Unfortunately, as of the time of writing, Dagger is at such an early stage of development that there is no experience yet in terms of large-scale projects, such as described in [41]. Consequently, due to time constraints, this thesis does not cover Dagger in the context of large scale projects, although this is definitely a starting point for future work.

4.4 Summary

4.4.1 Jenkins

This chapter compared Jenkins to Dagger, by using each tool to build a open source software project. Jenkins is an open source CI/CD tool designed to be extendable and customizable. Jenkins can run on any machine with a Java runtime. Jenkins is operated via a web interface. In this chapter, Jenkins was used to build an Android application with a pipeline consisting of five stages Git checkout, linter, unit tests, application assembly and publishing. Each of the stages is a quality gate. The pipeline itself is written in Groovy DSL. Jenkins itself, as well as a pipeline, can be extended using plugins.

4.4.2 Dagger

Dagger is a portable development kit for CI/CD running in any Docker-compatible environment. Therefore Dagger's pipeline is independent from a CI environment and can be ported to any CI environment, such Jenkins, CircleCI, GitHub, etc., and can even be run locally without a CI environment. Dagger does not replace a CI environment, but rather builds a portable layer on top of it. Dagger's pipelines are called plans. A plan is written in CUE DSL. CUE allows developers to specify data of a type as concrete values satisfying set constraints. CUE also allows to both define and enforce a schema. In this case, Dagger was used to build a web application using Gradle. The proposed Dagger plan builds the application, runs unit tests and publishes a Docker image containing the application into a GitHub repository.

4.4.3 Comparison

When comparing Jenkins to Dagger, Jenkins uses the environment it is running in to execute the pipeline, which can be directly on the machine, in a virtual machine, in a Docker container, etc. This means it is more complex to keep the environment consistent locally and on a build machine, but also makes it possible to use build systems, which cannot be run inside a Docker container. In contrast to Jenkins, Dagger always executes its builds inside a Docker environment, keeping the build environment consistent regardless of the machine it is running on. A Jenkinsfile's complexity can be reduced by encapsulating code into Shared Libraries, which can be made accessible via SCM to multiple pipelines. Jenkins can also be extended by using plugins. Plugins can not only be used in a Pipeline, but every part of Jenkins, such as the web interface. Plugins are published under strict guidelines into a shared GitHub repository. They are discovered and downloaded from the plugins index. Dagger has a less complex approach to extend a plan and make actions reusable, due to its early stage of development: Dagger uses modules containing packages, which can either be in a local or remote repository. Dagger modules can also be made accessible publicly by becoming part of the Dagger universe. There are no guidelines for documentation, making it complex and time consuming to understand and integrate third party modules. Although Jenkins is used in large scale systems [30], Jenkins can reach its limits, especially in microservice applications with a large number of parallel deployments. There are no large scale projects using Dagger yet, although there is a comparable tool (LambdaCD), which is used to frequently build, test and deploy multiple microservice applications in parallel.

5 Conclusion

This chapter summarizes the prior chapters as well as reviewing and answering the initial question RQ. Finally, it gives a prospect into possible future work.

5.1 Summary of results

A survey was conducted to get an understanding of what functionalities developers are missing in established CI/CD tools. The survey shows that developers like that Jenkins is available for free, has a large community and an extensive list of available plugins, but miss an intuitive interface, local debugging functionality and find maintenance time consuming. Furthermore, new CI/CD tools should require less scripting and should be executable in the cloud. Two CI/CD pipelines using Jenkins and Dagger were developed. Jenkins is a well established open source CI/CD tool living from its highly customizable and extendable plugin architecture and is widely adopted, but is not up to date in terms of portability, usability and reaches its limits in complex applications. Dagger builds a layer on top of the CI/CD platform, runs in a Docker environment and can therefore be executed independently on any CI/CD platform as well as locally. Dagger learns from established tools by unifying and simplifying the process of extending plans, reusing components and making pipelines executable and debuggable locally. Being in an early stage of development, Dagger lacks most of the functionality available as plugins, is poorly documented in many places, is not usable in some edge cases and there is no experience of using Dagger in a large scale, commercial application.

5.2 Review of results

Dagger does not satisfy all unmet needs, but addresses issues found in Jenkins. Since Dagger in an early development stage, the number of available modules, compared to

Jenkins' large selection of plugins, is very limited, forcing developers to implement the needed functionality themselves and increasing, rather than decreasing the need to script. By building a layer on top of a CI/CD platform, Dagger is decoupled from the environment it is executed in. Thus, the web interface and the ability to be executed in the cloud depends on the CI/CD platform Dagger is run on, rather than Dagger itself. Being portable to any Docker environment, Dagger is also debuggable locally. Dagger reduces the required maintenance by streamlining the process of moving individual components of a plan into modules. The modules itself can be tested and reused in other plans. Therefore, the overall complexity and hence the required maintenance is reduced. On these grounds, Dagger is not yet ready to be used in large scale commercial applications. Although, this does not mean Dagger is not going to succeed. There are examples, i.e. Docker, which started as a niche product with very limited feature set and compatibility. Once there was demand for containerization in the industry, Docker's rate of development became rapid [50, 9]. Dagger has the best prerequisites for a similar course, since it builds on the already established Docker platform and can be run on established CI/CD platforms.

5.3 Future work

There are several starting points for future work in this thesis. One could be to conduct the survey with a significantly larger sample size, i.e. 1000 people from the target population [54], since the number of respondents is currently limited to three. This would increase the variety of respondents, their answers and the aspects they miss in established CI/CD tools, thus making the outcome of the survey more significant. Another starting point could be to include other CI/CD tools into the comparison, such as CircleCI, Buddy, TravisCI, etc., to provide a broader comparison between available CI/CD tools. Finally, Dagger has been developed extensively since the time of finishing this thesis: In addition to the CUE SDK, Dagger now offers SDKs in Go, Node.js, Python and GraphQL, which could be evaluated and compared to established CI/CD tools in the same way it was done for the CUE SDK.

Bibliography

[1] : *alpine*. – URL https://hub.docker.com/_/alpine. – Zugriffsdatum: 2022-10-02

[2] : *Buddy Docks*. – URL https://buddy.works/docs. – Zugriffsdatum: 2022-11-23

[3] : *BUILD PIPELINES AS CODE*. – URL www.lambda.cd. – Zugriffsdatum: 2022-11-07

[4] : *BuildKit*. – URL https://github.com/moby/buildkit. – Zugriffsdatum: 2022-10-20

[5] : *cimg/base*. – URL https://circleci.com/developer/images/image/cimg/base. – Zugriffsdatum: 2022-10-10

[6] : *CircleCI Docs*. – URL https://circleci.com/docs/. – Zugriffsdatum: 2022-10-10

[7] : *Dagger vs. Other Software*. – URL https://docs.dagger.io/. – Zugriffsdatum: 2022-09-05

[8] : *developers*. – URL https://developer.android.com/. – Zugriffsdatum: 2022-10-15

[9] : *Docker Blog*. – URL https://www.docker.com/blog/tag/docker/. – Zugriffsdatum: 2022-11-30

[10] : *docker docks*. – URL https://docs.docker.com/. – Zugriffsdatum: 2022-12-12

[11] : *eclipse-temurin*. – URL https://hub.docker.com/_/eclipse-temurin. – Zugriffsdatum: 2022-10-03

[12] : *Get insights quickly, with Google Forms.* – URL https://www.google.com/intl/en/forms/about/. – Zugriffsdatum: 2022-06-27

[13] : *HTML Publisher.* – URL https://plugins.jenkins.io/htmlpublisher/. – Zugriffsdatum: 2022-08-09

[14] : *Installation Trends JSON.* – URL https://stats.jenkins.io/plugin-installation-trend/. – Zugriffsdatum: 2022-11-05

[15] : *Introduction to Archetypes.* – URL https://maven.apache.org/guides/introduction/introduction-to-archetypes.html. – Zugriffsdatum: 2022-10-31

[16] : *Introduction to Microsoft Forms.* – URL https://support.microsoft.com/en-us/office/introduction-to-microsoft-forms-bb1dd261-260f-49aa-9af0-d3dddcea6d69. – Zugriffsdatum: 2022-06-21

[17] : *Jenkins Continuous Integration and Delivery server.* – URL https://hub.docker.com/r/jenkins/jenkins. – Zugriffsdatum: 2022-10-03

[18] : *Jenkins User Documentation.* – URL https://www.jenkins.io/doc/book/. – Zugriffsdatum: 2022-10-15

[19] : *Kotlin.* – URL https://kotlinlang.org/. – Zugriffsdatum: 2022-10-20

[20] : *Licenses and Standards.* – URL https://opensource.org/licenses. – Zugriffsdatum: 2022-11-05

[21] : *Plugins Index.* – URL https://plugins.jenkins.io/. – Zugriffsdatum: 2022-10-29

[22] : *Tutorial Spring Boot Kotlin.* – URL https://github.com/spring-guides/tut-spring-boot-kotlin. – Zugriffsdatum: 2022-11-16

[23] : *What is Jenkins?.* – URL https://www.cloudbees.com/jenkins/what-is-jenkins. – Zugriffsdatum: 2022-05-24

[24] : *Xcode.* – URL https://developer.apple.com/xcode/. – Zugriffsdatum: 2022-10-20

[25] IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990* (1990), S. 1–84

[26] : *App Center Plugin*. 2021. – URL https://github.com/jenkinsci/appcenter-plugin. – Zugriffsdatum: 2022-08-15

[27] : *What is Gradle?* 2021. – URL https://docs.gradle.org/current/userguide/what_is_gradle.html. – Zugriffsdatum: 2022-05-27

[28] : *Build-Test-Distribute Mobile Apps using App Center*. 2022. – URL https://www.azuredevopslabs.com/labs/vstsextend/appcenter/#:~:text=Visual%20Studio%20App%20Center%20is,one%20single%20integrated%20cloud%20solution.. – Zugriffsdatum: 2022-08-01

[29] : *INTRODUCING DAGGER: A NEW WAY TO CREATE CI/CD PIPELINES*. 2022. – URL https://dagger.io/blog/public-launch-announcement. – Zugriffsdatum: 2022-05-25

[30] Arachchi, S.A.I.B.S. ; Perera, Indika: Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. In: *2018 Moratuwa Engineering Research Conference (MERCon)*, 2018, S. 156–161

[31] Armenise, Valentina: Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. In: *Proceedings of the Third International Workshop on Release Engineering*, IEEE Press, 2015 (RELENG '15), S. 24–27

[32] Boettiger, Carl: An Introduction to Docker for Reproducible Research. In: *SIGOPS Oper. Syst. Rev.* 49 (2015), jan, Nr. 1, S. 71–79. – URL https://doi.org/10.1145/2723872.2723882. – ISSN 0163-5980

[33] Chen, Lianping: Continuous Delivery: Huge Benefits, but Challenges Too. In: *IEEE Software* 32 (2015), Nr. 2, S. 50–54

[34] Crispin, Lisa: Driving Software Quality: How Test-Driven Development Impacts Software Quality. In: *IEEE Software* 23 (2006), Nr. 6, S. 70–71

[35] Cunningham, Ward: *Principles behind the Agile Manifesto*. 2001. – URL http://agilemanifesto.org/principles.html. – Zugriffsdatum: 2022-05-10

[36] Dzamashvili Fogelström, Nina ; Gorschek, Tony ; Svahnberg, Mikael ; Olsson, Peo: The impact of agile principles on market-driven software product development. In: *Journal of Software Maintenance and Evolution: Research and Practice* 22 (2010), Nr. 1, S. 53–80. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spip.420

[37] Fowler, Martin: *Continuous Integration*. 2006. – URL https://martinfowler.com/articles/continuousIntegration.html. – Zugriffsdatum: 2022-05-24

[38] Fowler, Martin: *ContinuousDelivery*. 2013. – URL https://martinfowler.com/bliki/ContinuousDelivery.html#footnote-when. – Zugriffsdatum: 2022-05-27

[39] Guay, Matthew: *The 8 best online form builder apps*. 2022. – URL https://zapier.com/blog/best-online-form-builder-software/. – Zugriffsdatum: 2022-06-27

[40] Habchi, Sarra ; Blanc, Xavier ; Rouvoy, Romain: *On Adopting Linters to Deal with Performance Concerns in Android Apps*. S. 6–16. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA : Association for Computing Machinery, 2018. – URL https://doi.org/10.1145/3238147.3238197. – ISBN 9781450359375

[41] Hasselbring, Wilhelm ; Steinacker, Guido: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, S. 243–246

[42] Leitschuh, Jonathan: *Ktlint Gradle*. 2022. – URL https://github.com/JLLeitschuh/ktlint-gradle. – Zugriffsdatum: 2022-08-04

[43] Luer, Chris: Assessing Module Reusability. In: *First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '07)*, 2007, S. 7–7

[44] Matinlassi, M.: Evaluating the portability and maintainability of software product family architecture: terminal software case study. In: *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, 2004, S. 295–298

[45] Olsson, Helena H. ; Alahyari, Hiva ; Bosch, Jan: Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012, S. 392–399

[46] Patil, Shreyas: *Foodium*. 2021. – URL https://github.com/PatilShreyas/Foodium. – Zugriffsdatum: 2022-08-04

[47] Punter, T. ; Ciolkowski, M. ; Freimut, B. ; John, I.: Conducting on-line surveys in software engineering. In: *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, 2003, S. 80–88

[48] Savor, Tony ; Douglas, Mitchell ; Gentili, Michael ; Williams, Laurie ; Beck, Kent ; Stumm, Michael: Continuous Deployment at Facebook and OANDA. In: *Proceedings of the 38th International Conference on Software Engineering Companion.* New York, NY, USA : Association for Computing Machinery, 2016 (ICSE '16), S. 21–30. – URL https://doi.org/10.1145/2889160.2889223. – ISBN 9781450342056

[49] Sijbrandij, Sid: *A brief history of GitLab.* – URL https://about.gitlab.com/company/history/. – Zugriffsdatum: 2022-05-24

[50] Singh, Sachchidanand ; Singh, Nirmala: Containers & Docker: Emerging roles & future of Cloud technology. In: *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, S. 804–807

[51] Soni, Mitesh: End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. In: *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015, S. 85–89

[52] Taylor, David: *20 Best Continuous Integration(CI/CD) Tools in 2022.* 2022. – URL https://www.guru99.com/top-20-continuous-integration-tools.html. – Zugriffsdatum: 2022-05-24

[53] Verma, Abhishek ; Pedrosa, Luis ; Korupolu, Madhukar ; Oppenheimer, David ; Tune, Eric ; Wilkes, John: Large-Scale Cluster Management at Google with Borg. New York, NY, USA : Association for Computing Machinery, 2015 (EuroSys '15). – URL https://doi.org/10.1145/2741948.2741964. – ISBN 9781450332385

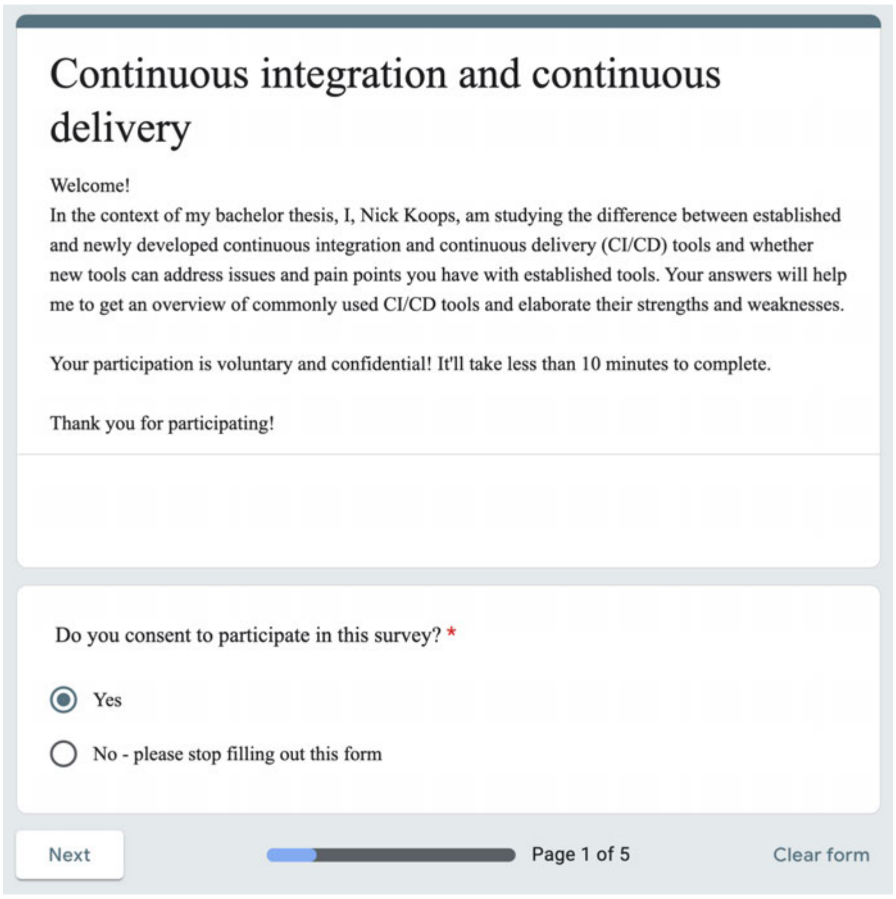
[54] Zhang, Yang ; Vasilescu, Bogdan ; Wang, Huaimin ; Filkov, Vladimir: One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* New York, NY, USA : Association for Computing Machinery, 2018

(ESEC/FSE 2018), S. 295–306. – URL https://doi.org/10.1145/3236024.3236033. – ISBN 9781450355735

A Appendix

A.1 Survey

A.1.1 Survey section 1


Continuous integration and continuous delivery

Welcome!
In the context of my bachelor thesis, I, Nick Koops, am studying the difference between established and newly developed continuous integration and continuous delivery (CI/CD) tools and whether new tools can address issues and pain points you have with established tools. Your answers will help me to get an overview of commonly used CI/CD tools and elaborate their strengths and weaknesses.

Your participation is voluntary and confidential! It'll take less than 10 minutes to complete.

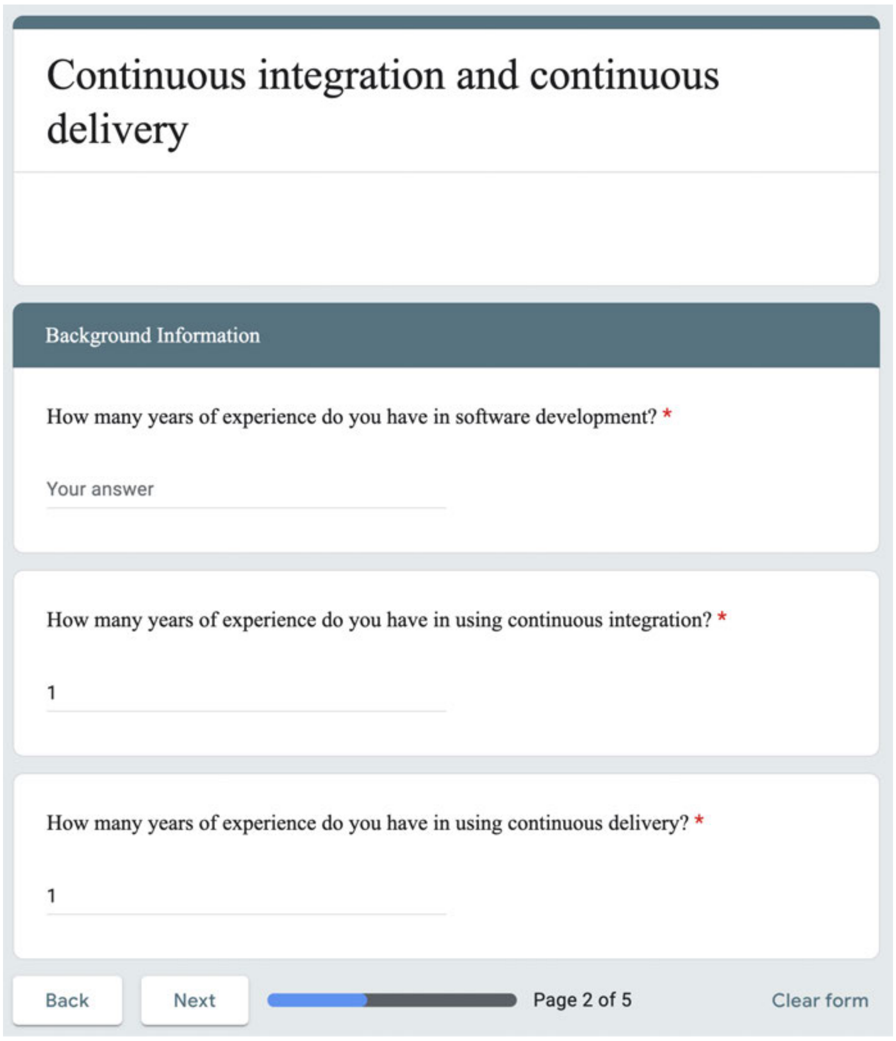
Thank you for participating!

Do you consent to participate in this survey? *

[✓ Yes]

No - please stop filling out this form

A.1.2 Survey section 2



Continuous integration and continuous delivery

Background Information

How many years of experience do you have in software development? *

Your answer

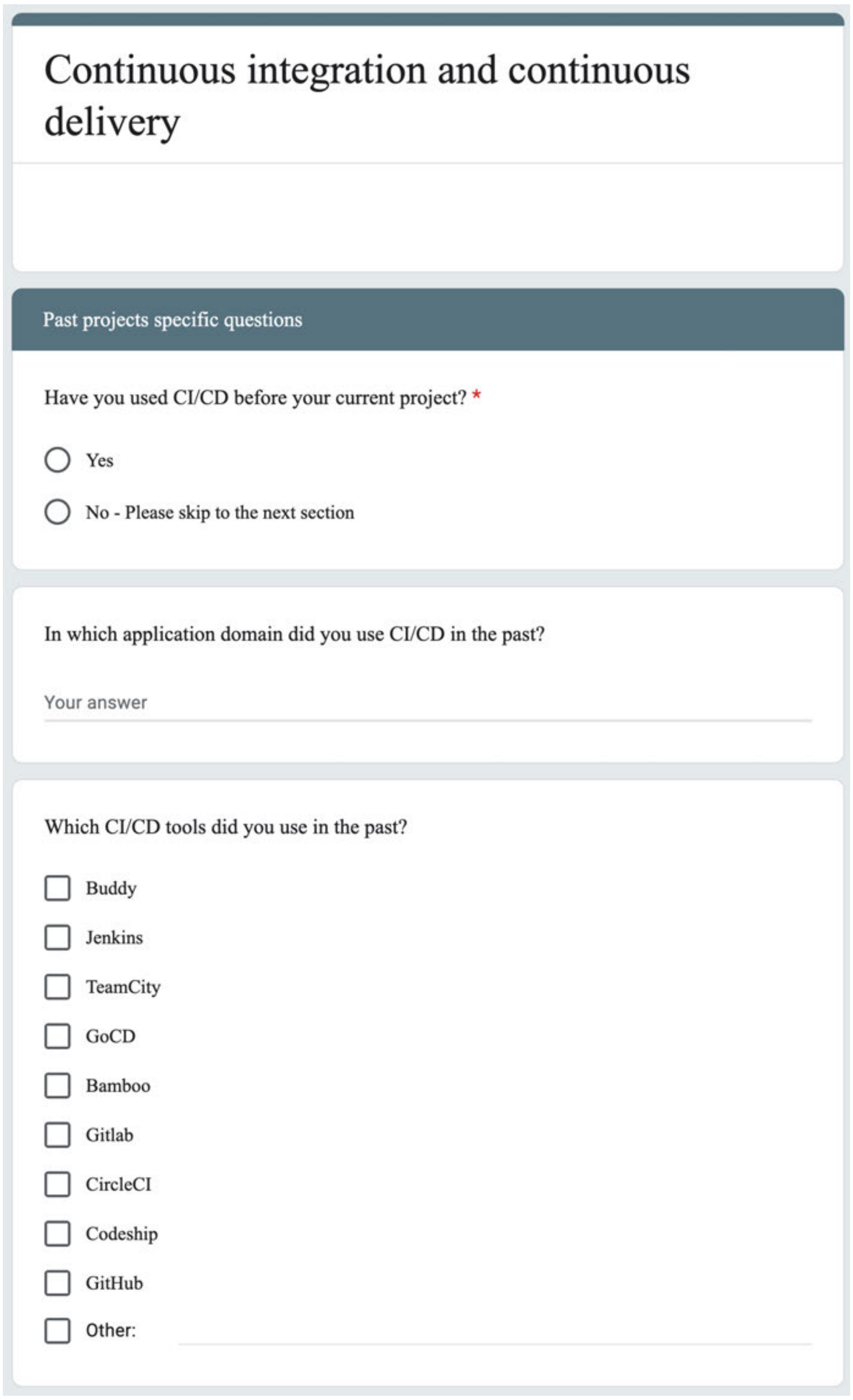
How many years of experience do you have in using continuous integration? *

1

How many years of experience do you have in using continuous delivery? *

1

A.1.3 Survey section 3



Continuous integration and continuous delivery

Past projects specific questions

Have you used CI/CD before your current project? *

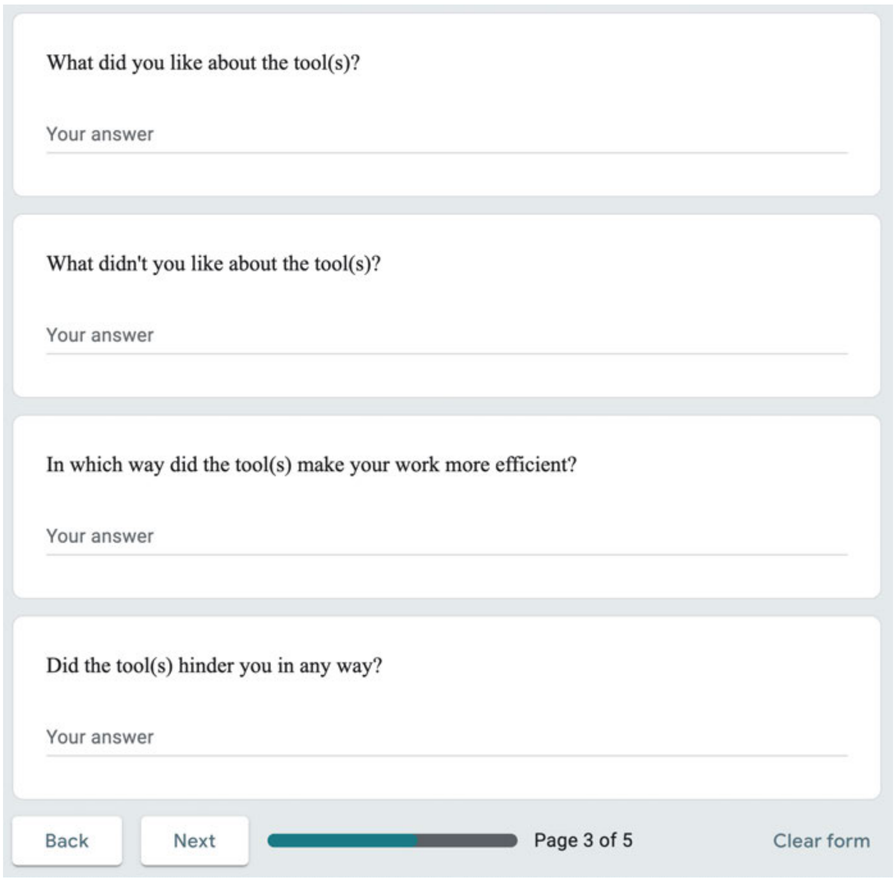
- ◯ Yes
- ◯ No - Please skip to the next section

In which application domain did you use CI/CD in the past?

Your answer

Which CI/CD tools did you use in the past?

- ☐ Buddy
- ☐ Jenkins
- ☐ TeamCity
- ☐ GoCD
- ☐ Bamboo
- ☐ Gitlab
- ☐ CircleCI
- ☐ Codeship
- ☐ GitHub
- ☐ Other:



What did you like about the tool(s)?

Your answer

What didn't you like about the tool(s)?

Your answer

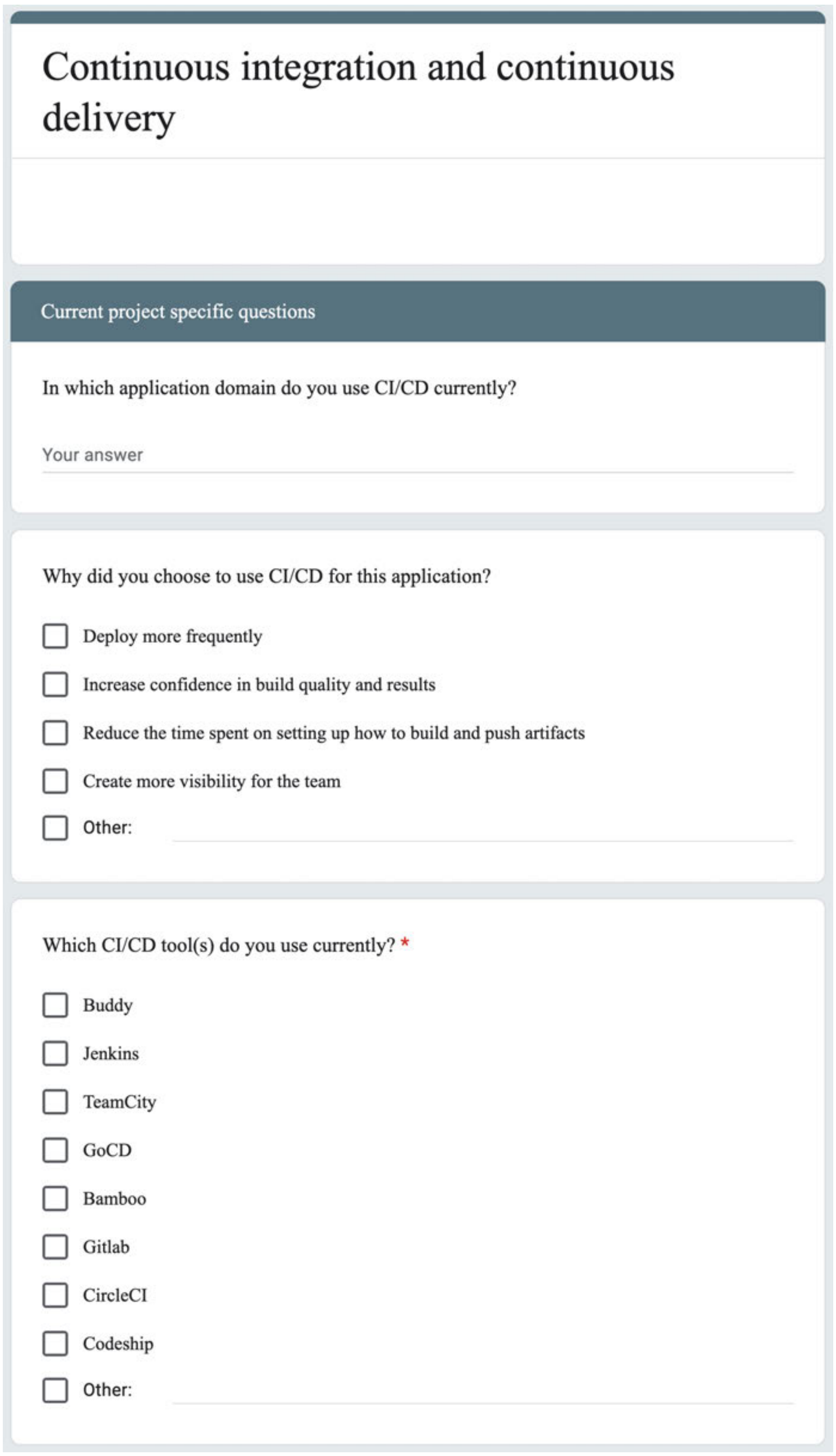
In which way did the tool(s) make your work more efficient?

Your answer

Did the tool(s) hinder you in any way?

Your answer

A.1.4 Survey section 4



Continuous integration and continuous delivery

Current project specific questions

In which application domain do you use CI/CD currently?

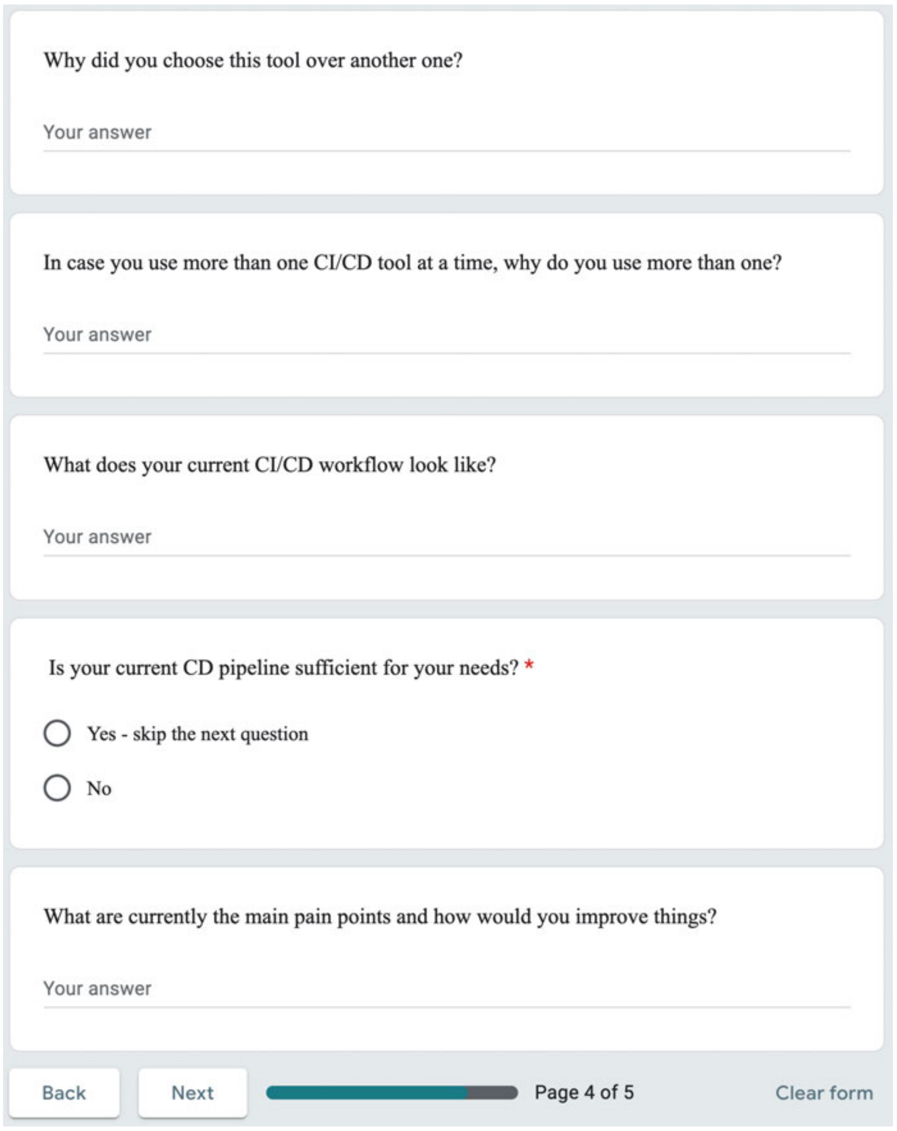
Your answer

Why did you choose to use CI/CD for this application?

- ☐ Deploy more frequently
- ☐ Increase confidence in build quality and results
- ☐ Reduce the time spent on setting up how to build and push artifacts
- ☐ Create more visibility for the team
- ☐ Other:

Which CI/CD tool(s) do you use currently? *

- ☐ Buddy
- ☐ Jenkins
- ☐ TeamCity
- ☐ GoCD
- ☐ Bamboo
- ☐ Gitlab
- ☐ CircleCI
- ☐ Codeship
- ☐ Other:



Why did you choose this tool over another one?

Your answer

In case you use more than one CI/CD tool at a time, why do you use more than one?

Your answer

What does your current CI/CD workflow look like?

Your answer

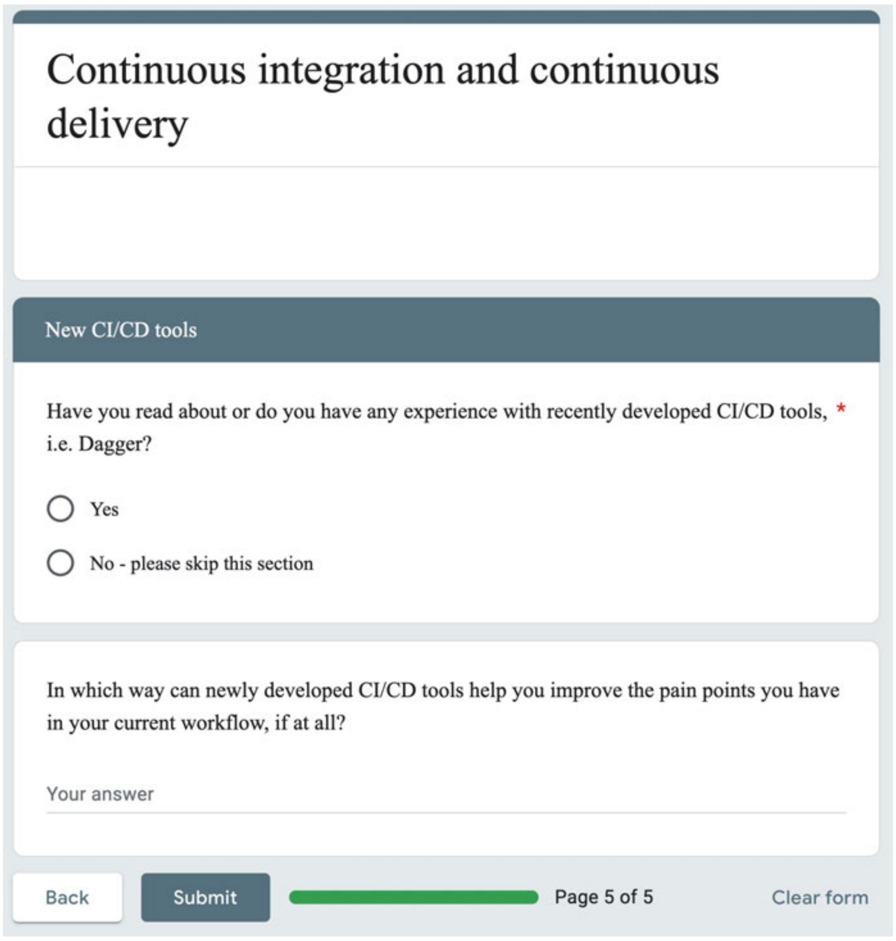
Is your current CD pipeline sufficient for your needs? *

- ○ Yes - skip the next question
- ○ No

What are currently the main pain points and how would you improve things?

Your answer

A.1.5 Survey section 5



Continuous integration and continuous delivery

New CI/CD tools

Have you read about or do you have any experience with recently developed CI/CD tools, i.e. Dagger? *

◯ Yes

◯ No - please skip this section

In which way can newly developed CI/CD tools help you improve the pain points you have in your current workflow, if at all?

Your answer

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

[redacted]

Ort	Datum	Unterschrift im Original