Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie

Evaluation of Continuous Integration Solutions for Kieker

Nakharin Donsuypae, Robin Finkbeiner, Sebastian Harner

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Thomas F. Düllmann, M.Sc.
Commenced:	March 14, 2017
Completed:	September 14, 2017
CR-Classification:	I.7.2

Abstract

Nowadays software development is becoming more and more complex. To ensure that quality and functionality of the software is guaranteed extended testing is required. Testing is a very time consuming and often repetitive task. Continuous Integration (CI) aims to solve this by automating the whole build and test process. This research study focuses on the analysis of different Continuous Integration solutions in terms of functionality, usability and suitability for the open source project Kieker, an application performance monitoring tool. Kieker already utilized SnapCI and Jenkins for Continuous Integration. Unfortunately SnapCI, a hosted service by Thoughtworks was discontinued in May 2017. By analyzing different CI tools based on requirements previously defined we chose three solutions that would be suitable. To further evaluate these three, a prototypical setup has been implemented. Based on these findings a recommendation was given to the Kieker team that would satisfy the requirements.

Kurzfassung

Heutzutage wird Softwareentwicklung immmer komplexer. Um die fortlaufende Qualität und Funktionalität der Software sicherzustellen, bedarf es häufigeres Testen. Testen ist ein sehr zeitintensiver und oft repetitiver Prozess. Continuous Integration (CI) versucht dies zu loesen, indem es ganze Erstellungs- und Testprozesse automatisiert. Diese Fachstudie hat als Ziel die Analyse verschiedener CI Lösungen, bezogen auf Funktionalität, Benutzbarkeit und Tauglichkeit für das Open Source Projekt Kieker, ein Application Performance Monitoring Werkzeug. Kieker benutzt bereits SnapCI und Jenkins für Continuous Integration. Leider wird SnapCI, ein gehosteter Service von Thoughtworks, ab Mai 2017 nicht mehr angeboten. Durch das Analysieren verschiedener CI Lösungen basierend auf den Anforderungen wurden drei potentielle Lösungen ausgewählt. Um diese drei Optionen weiter zu untersuchen wurde eine protoypische Konfiguration aufgesetzt. Basierend auf den Ergebnissen wurde eine Empfehlung an das Kieker Team ausgesprochen, welche die Anforderungen erfüllt.

Contents

Chapter 1

Introduction

Today software developing is becoming more and more complex. To ensure that quality and functionality of the software is guaranteed one needs to do extended testing. Testing is a very time consuming and often repetitive task. Continuous Integration aims to solve this by automating the whole build and test process. This research study will focus on the analysis of different Continuous Integration solutions in terms of functionality, usability and suitability for the open source project Kieker.

1.1 Motivation

Continuous Integration is a promising piece in the process to improve code quality and functionality. One benefit of CI is the possibility of fast and early feedback on a product increment. CI removes the risk of big integrations by testing and integrating every code contribution to the code base. Therefore CI lowers the risk of a product failing customer expectations. Above-mentioned benefits are one of the reasons why Kieker is using a CI solution.

Kieker is an open source tool for analyzing internal behavior of large-scale software systems. "[It] provides complementary dynamic analysis capabilities, i.e., monitoring and analyzing a software system's runtime behavior [...]" [Kie]. Kieker utilized SnapCI and Jenkins for Continuous Integration. Unfortunately SnapCI, a hosted service by Thoughtworks was discontinued in May 2017. To further use CI in Kieker, a new solution is needed. There is a large number of different CI solutions, that are divided into self-hosted and hosted-service tools. But not all tools are suitable for the project.

1.2 Goals

The goal of this research study is to recommend a suitable CI solution for Kieker that replaces SnapCI. The proposed solution has to satisfy the requirements defined in the following sections. Selected candidates will be evaluated regarding possible costs, complexity and hosting. The best candidates will be prototypically implemented to test whether or not they are suited for the Kieker project stack.

1.3 Structure

This research study is structure as followed:

Chapter 2 – Continuous Integration and Delivery: Introduction to the concepts and components of Continuous Integration and Delivery.

Chapter 3 – Requirements: Overview of mandatory and optional features for a working Continuous Integration Solution.

Chapter 4 – Tools: Short description and evaluation of each inspected tool.

Chapter 5 – Test Setup: Prototypical implementation of three potential Continuous Integration solutions.

Chapter 6 – Evaluation: Comparison of the three chosen Continuous Integration solutions.

Chapter 7 – Conclusion: Final conclusion and recommendation of one solution for the Kieker project.

Chapter 2

Continuous Integration and Delivery

In this chapter we introduce the concepts of continuous integration and delivery. We elaborate their benefits and differences. In the first place, we define core components of continuous integration and continuous delivery solutions. As there are many different terms for similar concepts it is important to establish a common understanding of components used. Finally, we provide a definition for continuous integration and delivery.

2.1 Core Components

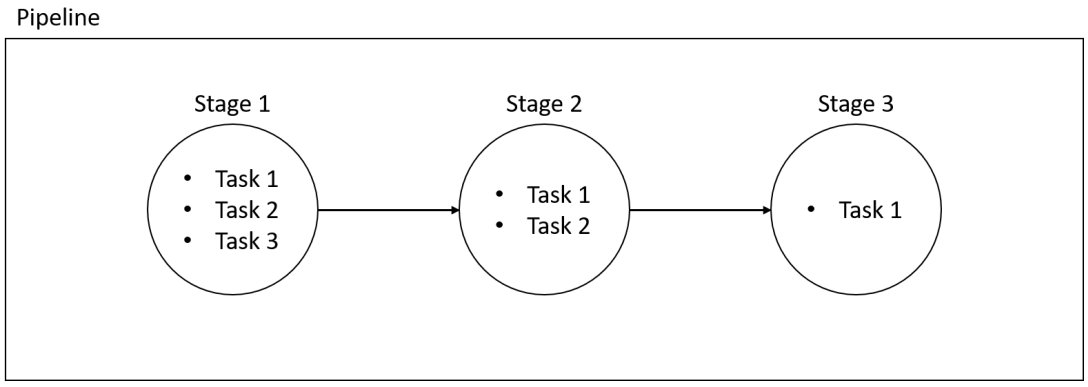

Figure 2.1: Continuous Integration Setup

Task

A task is the execution of a single well defined step with dependent resources available to it. For example, checking out the code base from the version control or executing defined unit tests with gradle.

Stages

A stage is a collection of tasks. One stage executes all containing tasks. If every task has completed successfully the stage is finished. Each stage runs in an isolated environment. Stages can be dependent or independent from previous stages. Dependent ones need to be run in sequential order whereas independent stages can also be executed in parallel.

Pipeline

At a high level the pipelines primary function is to coordinate and execute every required step before a software increment can be added to the existing code base. The combination of stages result in a pipeline. The pipeline is responsible for managing transitions between stages, e.g. transferring artifacts from one stage to another.

Build

An instance of execution of a pipeline is called a build. A build can either succeed or fail. Each stage defined by the pipeline is executed, and so long as all Stages succeed, the build succeeds. If a stage fails, the current execution of the pipeline terminates and the build fails.

Worker

To be able to execute tasks in a pipeline one eventually needs multiple different environments. As it is not feasible to run every environment on the same system we introduce workers, separated hardware/virtual machines that are preconfigured to support a dedicated environment. This allows for a distributed CI solution which is scalable and parallelizable. Also, workers can be used to run multiple stages at the same time to decrease build times.

2.2 Continuous Integration

The overall objective of Continuous Integration is to automatically integrate new code into the existing code base. Continuous Integrations verifies that your software works with every new change - and you know the moment it breaks and can fix it immediately. Ideally malfunctioning code is never added to the code base. Additionally Continuous Integration can be utilized to validate and ensure software quality. A project that is to be used in Continuous Integration, has to fulfill several requirements. First, it must be possible to build the code base programmatically, e.g by using build tools like gradle or maven. As Humble [HF10] states build scripts should be treated like the code base, versionized and tested. Ideally, the complete Continuous Integration solution is documented and configured as code for both the build tool and the configuration of the pipeline. This is also known as Infrastructure as Code. Second, the project has to be managed in a version control system such as git or svn.

2.3 Continuous Delivery

Continuous Delivery takes Continuous Integration one step further. The goal is to automate the whole deployment process so that new product versions, that have passed the Continuous Integration build, are delivered to a production environment automatically. By automating the repetitive and often tedious task of software deployment, time spent deploying is decreased. This can lead to more releases and a faster feedback loop from stakeholders. Another benefit is that every member of the team could possibly trigger a deployment of the project. There is no more need for a dedicated role responsible for deployment. Overall this creates a release process that is repeatable, reliable and predictable [HF10].

Chapter 3

Requirements

In this chapter we collect and define requirements for our Continuous Integration solution. Even though these requirements are specifically modified to fit Kieker's development stack/process they are generally applicable to different projects. We differentiate between mandatory and optional requirements.

3.1 Mandatory

The following requirements are essential to have a usable CI solution for Kieker.

Today in software development a version control system is essential. To use these systems to our advantage the proposed solutions should have a good connection to our version control system. This allows us to exactly configure the granularity in which software increments are built. Additionally we can configure which events(commits, pull requests, tag, etc.) trigger a build. To follow the base principles of CI we want to test as many software increments as possible.

As Kieker can be used with Microservice Applicatons making it available as a Docker container is needed. To make the setup of each build step as easy and reproducible as possible we want to use Kieker's Docker container as our base image. Therefore, a proposed CI solution is required to support Docker container usage. The usage of containers guarantees perfect reproducibility and transparency. Furthermore we are able to utilize Docker to use every testing framework needed. In theory Docker provides excellent manageability of different build configurations due to usage of Dockerfiles to define our containers.

Moreover we would like to provide the configuration of our CI solution as easy and portable as possible. Infrastructure as code enables us to utilize version controlled

configurations and to deploy a solution on other systems without much overhead. In case of breaking changes, infrastructure as code makes it possible to restore a previous working state of the pipeline.

To be able to model a pipeline with multiple independent steps a proposed solution is required to support multiple stages. Each stage is executed in a new clean environment to avoid unwanted side effects during builds. Additionally multiple stages it allows us to run independent stages in parallel. And in case of a failure it is visible on which stage and step of the build the failure occurred. Ideally it should be possible to define the published artifacts for each stage. For some use cases it is required to pass specific artifacts from one stage to another.

As we do not always want to run the same set of stages for each branch it should be possible to define different pipelines for specific branches. Above mentioned features are required to be a fitting solution for the Kieker project environment.

3.2 Optional

The following requirements are features that are useful but not necessarily needed for Kieker.

Kieker already uses Github as a Version Control and Collaboration system so it is preferable to have a integration of the Repository in the Continuous Integration solution. For example to automatically detect and build pull requests. Additionally it would be convenient to have a integration of the CI solution in GitHub. That way feedback of one build can directly be displayed in Github, i.e. blocking the pull request if the build failed. The proposed solution should also support the concept of workers. The possibility to run multiple builds at the same time allows for a faster feedback loop. And in case of multiple developers working on the project simultaneously one does not have to wait for the build of another developer to finish. Depending on the hosting format of the solution we can easily allow for concurrent builds if we add additional computing power or workers.

Chapter 4

Tools

We compared ten different tools regarding the requirements described above. In this section we are going to give a quick overview of each tool regarding the above mentioned requirements. As it is not feasible for the scope of this work to build a prototypical implementation for each tool, we selected a subsection of three tools which we will discuss in more detail later. The goal is to give a recommendation that fits the Kieker environment best. The tools are classified into two groups based on their hosting. We evaluated the tools only focusing on their documentation and feature set as of May 2017.

The first group is self-hosted, the operators themselves are responsible for the setup, configuration and maintenance of the Continuous Integration solution, and the other group is hosted, which means that the complete infrastructure is supplied by an external service provider. The service provider takes care of the installation, configuration and actualization of the CI software. Developers only need to configure required pipelines.

4.1 Hosted CI Solutions

The main benefit of hosted solutions is the reduction in setup complexity. The service provider takes care of the often complicated and troublesome installation and operation of the tool. This gives developers the freedom to focus on the actual implementation of the pipeline. By using hosted solutions one is able to set up a working Continuous integration pipeline in a short time. For simple project structures it is possible for one developer to have a working solution up and running in a short time. But on the other hand we observed that hosted-solutions often are not able to model complex scenarios.

Buddy

Buddy [Bud] is a Docker-based Continuous Integration and deployment tool. It supports the concept of pipelines and stages. Buddy can be used as a hosted solution with five different pricing categories ranging from 49$ to 299$ per month. Most of the mandatory requirements are fulfilled by Buddy. But Buddy misses parallel stages and the possibility to download build or stage artifacts.

It is also possible to use Buddy for free on own machines. But the free version has some functional limitations regarding the number of users (only up to ten users) and permission configuration (everyone is an administrator) which makes it unsuitable for the Kieker environment.

CircleCI

Circle Ci 2.0 [Cir] is a comprehensive Continuous Integration and deployment tool. Circle satisfies all mandatory and optional requirements for Kieker. Pricing varies with the number of possible parallel builds and number of containers. Starting with a free tier for one concurrent build and ranging from 150$ for one container and four concurrent builds to 750$ per month. Circle Ci provides a good support for any language that runs on Linux.

Codeship

Codeship [Cod] provides two different versions, Pro and Basic. Codeship Basic has no Docker support which makes it unusable for the Kieker project. The feature set of Codeship Pro is very comprehensive but is missing stages. For Open Source projects Codeship is free. Otherwise pricing ranges from 75$ per month to roughly 1000$ per month.

Shippable

Shippable [Shi] offers a continuous integration service with Docker support. It has integrated Github and Bitbucket support and Shippable nearly satisfies all requirements for Kieker. At this time there is no option available to download artifacts or run parallel stages. Prices vary and Shippable offers pricing categories starting from a free version up to at least 500$ a month.

TravisCI

TravisCI [Tra] is a hosted continuous integration and deployment tool and is used by Github as a default CI solution. TravisCI offers a trial version for a specific duration. After the trial version prices start from 69$ per month up to 489$. Pricing categories vary and with each upgrade the the number of concurrent jobs increases. Furthermore TravisCI provides a free tier for open source projects. TravisCI doesn't support multiple pipelines yet and the staging functionality is currently in beta. It is possible to configure TravisCI with a YAML file, adding .travis.yml to the root directory of the repository.

Wercker

Wercker [Wer] is a hosted continuous integration tool with a user-friendly graphical user interface. It offers a free Community Edition, with the option of two concurrent jobs, as well as a virtual pipelines edition for 350$ a month and an enterprise edition where the prices aren't listed. Since Wercker is a Docker-Native platform we can use Kieker's docker image directly in our builds. It satisfies all requirements for Kieker and is therefore a possible replacement for SnapCI.

4.2 Self-hosted CI Solutions

Concourse

Concourse is a self-hosted open source Continuous Integration and deployment tool. The development is sponsored by Pivotal [Piv], they use Concourse for their own projects and in-house solutions. Concourse is free to use and licensed under the Apache license. Concourse is suitable for very large and complex projects. The complete configuration of pipelines is via Infrastructure as Code. Furthermore Concourse provides excellent controllability and configurability via a command line interface.

GoCD

GoCD [Goc] is a self-hosted solution from Thoughtworks. As it is developed by the same company that provided SnapCI there is a quick conversion of the pipeline configuration. Even though GoCD meets all requirements infrastructure as code is poorly documented which will make it hard to configure the pipeline in the end.

Drone

Drone [Dro] is a lightweight, powerful continuous delivery platform built for containers. Drone is packaged and distributed as a Docker image and can be downloaded from Dockerhub. Drone is the only product we analyzed that is still in beta. This means that there are still changes to come. But based on the documentation it meets most of the requirements that are crucial for a Kieker Continuous Integration solution. Even their documentation website has currently a bug which is a good indicator on the status of the tool.

Jenkins

Jenkins [Jend] is an open source Continuous Integration solution that uses a plugin system to extend functionality. This allows for nearly endless features. So it is no surprise that Jenkins meets all requirements. With version 2.0 it is now even possible to have your pipeline configuration as Infrastructure as Code. Jenkins has a big open-source community and is probably still the most used CI solution on the market.

Additionally, Kieker is already using Jenkins for nightly builds. That means that there exists a server infrastructure we can use.

Tool	Github Integration (Optional)	Docker Support	Multiple Pipelines	Staging	Infrastructure as Code	Agents	Concurrent Builds	Hosted/Self	Artifact Download	Parallel Stages	Interaction with Repository
Buddy	✓	✓	✓	✓	✓	✓	✓	hosted	✗	✗	✓
CircleCi	✓	✓	✓	✓	✓	✓	✓	hosted	✓	✓	✓
Codeship	✓	✓ (Pro)	✓	✗	✗	✓	✓	hosted	✓	✓ (Pro)	✓
Concourse	✓	✓	✓	✓	✓	✓	✓	self-hosted	✓	✓	✓
Drone	✓	✓	✗	✓	✓	✓	✗	hosted / self hosted	✓	✗	✓
gocd	✓	✓	✓	✓	✓	✓	✓	self-hosted	✓	✓	✓
Jenkins 2	✓	✓	✓	✓	✓	✓	✓	self-hosted	✓	✓	✓
Shippable	✓	✓	✓	✓	✓	✓	✓	hosted	✗	✗	✓
TravisCi	✓	✓	✗	Beta	✓	✓	✓	hosted	✓	✓	✓
Wercker	✓	✓	✓	✓	✓	✓	✓	hosted	✓	✓	✓

Figure 4.1: Overview Tools

Chapter 5

Test Setup

We chose three solutions based on our analysis that we think would fit as a CI solution for Kieker. To narrow our decision down to one we did a test setup and configuration with each of the three tools. We recreated the pipeline defined in SnapCI as shown in Figure 5.1.

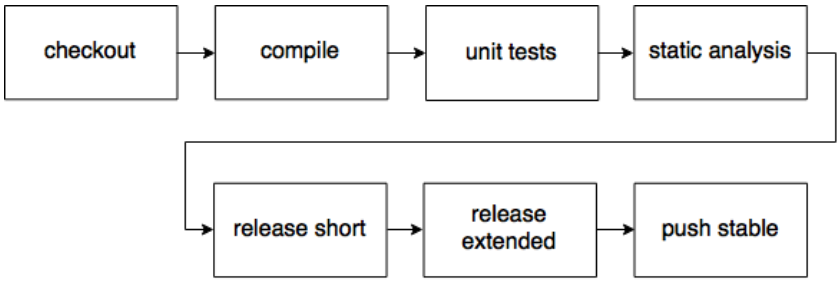

Figure 5.1: Pipeline Test Setup

5.1 Jenkins

The first solution we chose is Jenkins. Starting from version 2.0 it supports stages and by adding plugins we covered the rest of the features that were missing in the base tool. For example we used the Git [Jenb] and Github [Jenc] plugins to add a connection to version control and a Github integration that automatically detects pull requests and branches. Additionally, we added the plugin BlueOcean [Jena] which introduces an alternative User Interface that freshens up the quite old one from Jenkins. But it does not replace it because it only improves the presentation of build status and projects. The whole configuration can still only be done in the old interface or with a Jenkinsfile that

is stored in the repository. How you exactly use a Jenkinsfile, we will go into detail later. The huge amount of plugins and the strong community behind Jenkins were two of the reasons why we chose it. Additionally the amount of plugins provide nearly every feature that we would need and a solution for nearly every problem that occurred.

5.1.1 Installation

Another big reason why we chose Jenkins is that the Kieker project already used Jenkins for its nightly builds and we would have nearly no work with setup and maintenance. But usually the setup would include setting up a machine with Java to run Jenkins and possibly additional machines to act as workers in this case for Docker.

5.1.2 Setup

As mentioned above it is possible to configure the pipeline in Jenkins two different ways, using the user interface or by defining it in a Jenkinsfile. Using a file allows us to keep track of changes to the configuration of the pipeline. But there are even two types of pipeline definition, scripted and declarative. Figure 5.2 and Figure 5.3 show the differences of the two types.

The scripted Jenkinsfile allows a definition without much overhead. One defines the environment that the pipeline should run in and if wanted a dedicated worker. After that one can start to define the stages and tasks of the pipeline.

```
node('kieker-slave-docker') {

   stage ('Checkout') {
      sh 'echo "Checking out Workspace"'

      checkout scm
   }

   stage ('1-compile logs') {
      sh 'docker run --rm -v ' + env.WORKSPACE + ':/opt/kieker
          kieker/kieker-build:openjdk7 /bin/bash -c "cd /opt/kieker; ./gradlew -S
          compileJava compileTestJava"'
   }
}
```

Figure 5.2: Scripted Jenkinsfile defining a pipeline

The declarative one on the other hand allows a more readable definition of a pipeline. Here one can substitute blocks, like a docker definition, for a shell call that would have to been called instead.

```
pipeline {
  agent {
     label 'kieker-slave-docker'
  }

  environment {
    DOCKER_BASE = "docker run --rm -v ${env.WORKSPACE}:/opt/kieker
        kieker/kieker-build:openjdk7 /bin/bash -c "
  }

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('Compile') {
      steps {
        sh DOCKER_BASE + '"cd /opt/kieker; ./gradlew -S -p kieker compileJava
            compileTestJava"'
        //stash 'everything'
      }
    }
  }
}
```

Figure 5.3: Declarative Jenkinsfile defining a pipeline

After defining the pipeline one has to connect Jenkins with Github. The Github and BlueOcean plugin allows for direct connection via a personal token generated by the owner of the repository. The branches with a Jenkinsfile then get automatically built based on the pipeline defined.

5.1.3 Build Execution

There are two ways to start a build, manually from the web interface or automatically by committing a code change or creating a pull request. Figure 5.5 shows the representation of one build in the Jenkins BlueOcean interface. The stages are separated and one can for example look up the logs of each stage individually. If a build fails it is directly shown where.

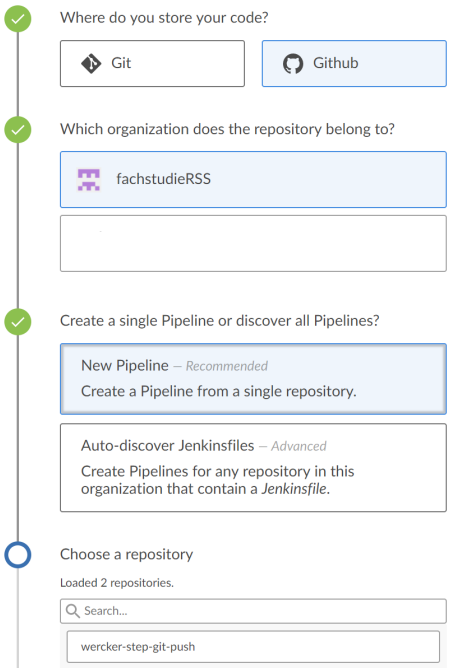

Figure 5.4: Adding a Github project with BlueOcean

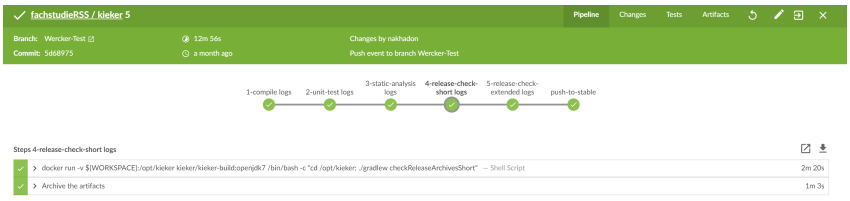

Figure 5.5: Representation of a successful run build.

5.1.4 Issues

As using Jenkinsfiles is a fairly new possibility to define the pipeline there is no centralized documentation on it. Additionally many plugins do not support it as of right now. Another problem we occurred with the combination of Jenkins and Docker is that currently there is no working solution for clearing the workspace before checking out new code changes. This causes to fill up the storage of the used machine and also might cause Jenkins to build the wrong version of the repository.

5.2 Concourse

The second tool we chose is Concourse. In our opinion Concourse is a very clean realization of a Continuous Integration tool. All configuration is done via a simple and programmer friendly command line tool. The Web interface only provides a graphical representation of configured pipelines and nothing more. Concourse uses a very flexible and simple concept to define pipelines. Every pipeline consists of three basic building blocks Tasks, Resources and Jobs. A Task is the execution of a script. If the scripts exits with 0 a task succeeds otherwise it fails. This corresponds with our definition of a task. Resources are entities used to check for new versions and/or pull and push new versions of the defined Resource. Concourse knows nothing about concepts like git instead it provides developers with the abstraction of resources. This abstraction is very powerful and developers are able to integrate all kinds of content into the pipeline and are not restricted by predefined resource types. Concourse provides a set of resources out of the box for essential functionality e.g a git resource. Also there are many resource types developed and maintained by the community. The last concept used are Jobs which are basically Stages. Every pipeline in Concourse consists of these three building blocks. In Figure 5.6 a excerpt of the pipeline is depicted. The stages shown are connected by dependencies to a resource. Succeeding stages are only executed if each previous stage has completed successfully.

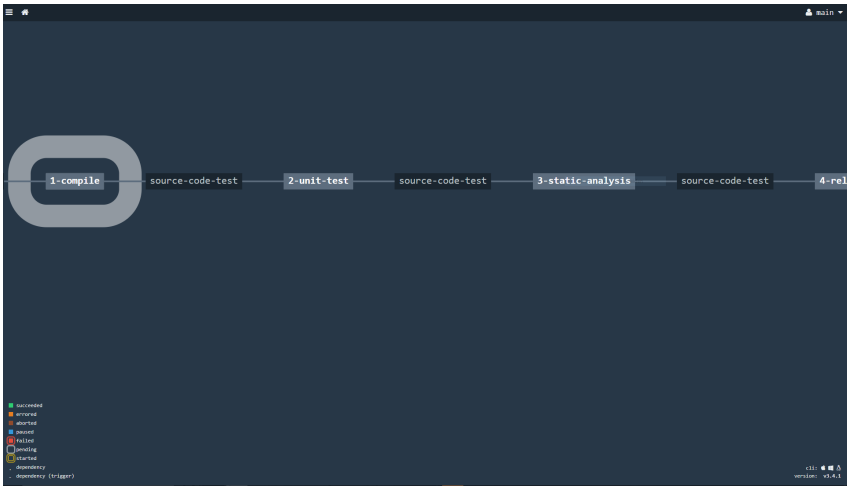

Figure 5.6: Graphical representation of a pipeline in the web interface.

5.2.1 Installation

The installation of Concourse is very straightforward. Concourse can be installed using standalone binaries or via Docker Compose. The host system is required to have Docker

and Docker-Compose installed. For our test setup we used the provided docker-compose file. One only has to configure the URL used to reach the web node and the credentials used for authentication. To be able to give different user permissions Concourse supports the definition of teams. Out of the box Concourse comes with a single team called main. This team has administrative permissions. More teams can be added via the CLI. To start the service only the docker-compose up command has to be executed.

```
$ fly set-team -n my-team --basic-auth-username ci --basic-auth-password changeme
$ fly login -n my-team
```

Figure 5.7: How to create multiple Teams

Also configuration of connected workers is done via the docker-compose file. To add more workers one has to add a new entry to the file and set the required parameters like shown in Figure 5.8 again the CLI tool can be used to check if workers are running.

```
concourse-worker:
  image: concourse/concourse
  privileged: true
  links: [concourse-web]
  depends_on: [concourse-web]
  command: worker
  volumes: ["./keys/worker:/concourse-keys"]
  environment:
    CONCOURSE_TSA_HOST: concourse-web
```

Figure 5.8: Add multiple workers.

5.2.2 Setup

In this subsection we are going to describe which steps are required to recreate the pipeline defined in SnapCI. First one has to connect to the selected Concourse instance via the CLI tool. The definition of a pipeline is done with a YAML file and describes required resources jobs and executed tasks. Listed below in Figures 5.9 to 5.12 are one example for each building block. First required resources are defined. Shown in 5.9 is a very rudimentary way to connect to a git repository. Concourse obviously allows for more comprehensive connections to a repository using OAuth, Webhooks or SSL. Every resource configuration is required to have a name, a type specification and all required information to authenticate against the entity providing the content of the resource. In the given example we connect to a git repo with hardcoded credentials. To hide such sensitive information Concourse allows to configure parameters that can be set with the CLI like shown in Figure 5.10.

```
resources:
- name: source-code-test
  type: git
  source: &repo-source
    uri: https://github.com/fachstudieRSS/kieker.git
    branch: concourse
    username: concourseFachstudie
    password: Fachstudie@RSS_2017
```

Figure 5.9: Definition of a Resource

```
$ fly -t example set-pipeline --pipeline my-pipeline --config pipeline.yml
--var "private-repo-key=$(cat id_rsa)"
```

Figure 5.10: Set parameters with the CLI tool.

The next component of a pipeline are the jobs. In Figure 5.11 a excerpt of the job definition for our test scenario is listed. Each Job consists of a name and a plan. The plan defines the steps to be taken to complete the job. For example the second job is called "1-compile" and its plan goes as follows. If the resource "source-code-test" triggers a true event and the previous job ("checkout") has passed successfully execute the task "compile". If the task completes, trigger a true event for the next job. There are two ways to configure a task. Either also define it in the same file or point to a YAML file containing the task. In the example we reference a task definition stored under the given file path in the repository.

```
jobs:
- name: checkout
  plan:
  - get: source-code-test
    trigger: true

- name: 1-compile
  plan:
  - get: source-code-test
    trigger: true
    passed: [checkout]
  - task: compile
    file: source-code-test/ci/run_compile_task.yml

- name: 2-unit-test
```

Figure 5.11: Definition of a Job

Figure 5.12 shows the definition of this task. One can configure the used Docker image for each task and the script to be run. As well as the input the task should take. Again it is possible to write the executed script directly in the task file or reference to a file.

```
platform: linux

image_resource:
  type: docker-image
  source:
    repository: kieker/kieker-build
    tag: openjdk7
inputs:
- name: source-code-test

run:
  path: sh
  args:
  - -exec
  - |
    cd source-code-test
    ./gradlew -S compileJava compileTestJava
```

Figure 5.12: Definition of a Task

After the pipeline is configured one has to push the configuration to the desired Concourse instance using the CLI like shown in Figure 5.13.

```
$ fly -t main set-pipeline --config kieker_pipeline.yml --pipeline kieker_pipeline
```

Figure 5.13: Publish a configuration with the CLI tool.

Above mentioned are all steps required to publish a working pipeline to a Concourse instance.

5.2.3 Build execution

There are three different ways in Concourse to trigger a build of a pipeline. First, one of the defined resources detects a new Version and starts the execution of the pipeline. Resources are able to detect new Versions in several ways e.g polling or web-hooks. Second, a build can be started manually via the web interface and lastly one can trigger the pipeline with the CLI tool.

5.2.4 Issues

While setting up the experimental Concourse instance we encountered some issues. There are compability problems with Ubuntu 14.04 LTS and Docker that prevented Concourse to create new Volumes to store data required to run jobs. To resolve this problem the installed Ubuntu system has to be updated. We were not able to find a way to automatically detect new branches but are confident that such a feature could be implemented.

Overall, we believe that Concourse is a very good and comprehensive solution. Surely able to provide the functionality required by Kieker.

5.3 Wercker

Wercker is the only hosted solution we chose. Since the free version of Wercker satisfies the needs of Kieker we chose it as a potential hosted solution. It fulfilled every requirement we have introduced and also offered a very user-friendly graphical user interface. The pipeline configuration can be done with a wercker.yml file where it is possible to define stages and which later can be further configured on the Wercker website. Wercker also offers the option to connect to Github with a few clicks and choose the respective repository. In the upcoming sections we will not talk about talk installation since Wercker is a hosted solution and no installation is needed.

5.3.1 Setup

The setup of Wercker is, compared to a self-hosted solution, relatively simple. One has to add the wercker.yml file as seen in Figure 5.14 to the root of the repository and define the stages it has to run. It is also possible to use variables and later set them on the website which we will talk about later on. The next step is to configure the pipeline with an in-built editor on the Wercker website.

```
box:
   id: kieker/kieker-build:openjdk7

1-compile-logs:
  steps:
    - script:
       name: directory
       code: |
         ls -a
    - script:
       name: 1-compile-logs
       code: |
         ./gradlew -S compileJava compileTestJava

2-unit-test:
  steps:
   - script:
       name: 2-unit-test logs
       code: |
         ./gradlew -S test
```

Figure 5.14: Wercker yaml file

Wercker defines a stage as a pipeline. To add stages one has to click on the "add a new pipeline" button and define the stage name and the YAML pipeline name. After all

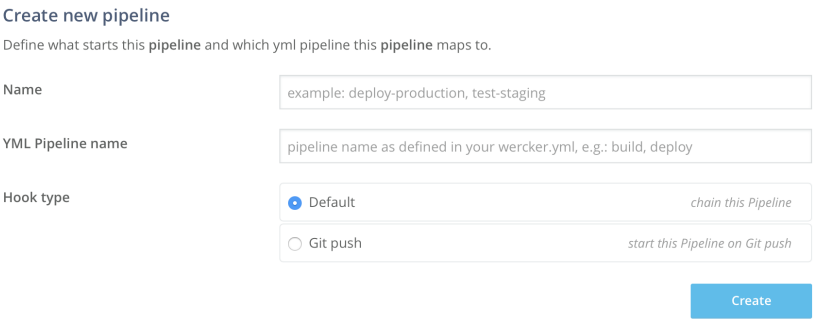

Figure 5.15: Add a new stage

stages have been added it is possible to use the workflow function offered by Wercker. Workflows are defined as as a pipeline and are a way to manage the automation of different stages. With one click it is possible to add the next stage and define which branches can be ignored as seen on Figure 5.16.

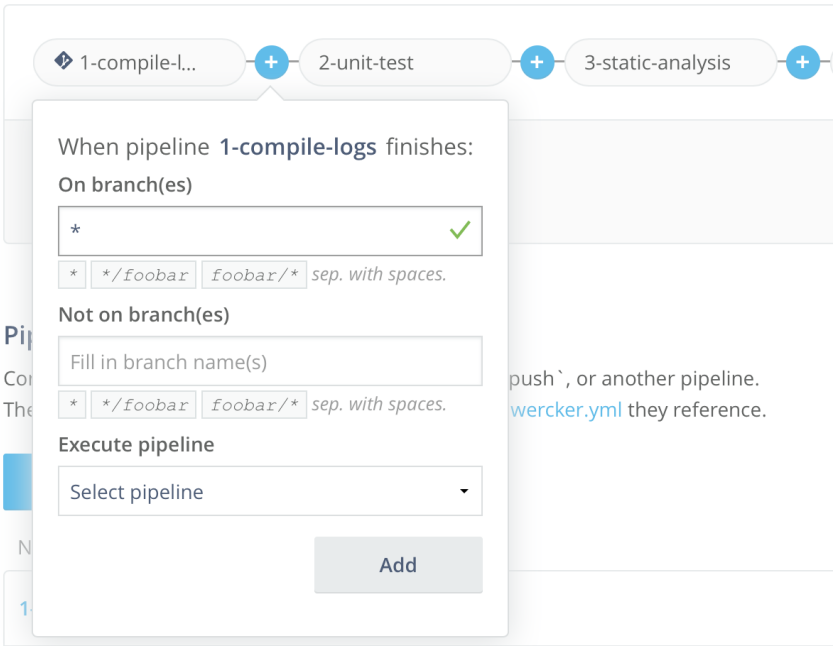

Figure 5.16: Normal Case: 1 Request & 1 Response.

Additionally, one can set variables in the wercker.yml file and set the value on the Wercker website. This feature is especially useful if you have to define a token and do not want to save it on the wercker.yml file.

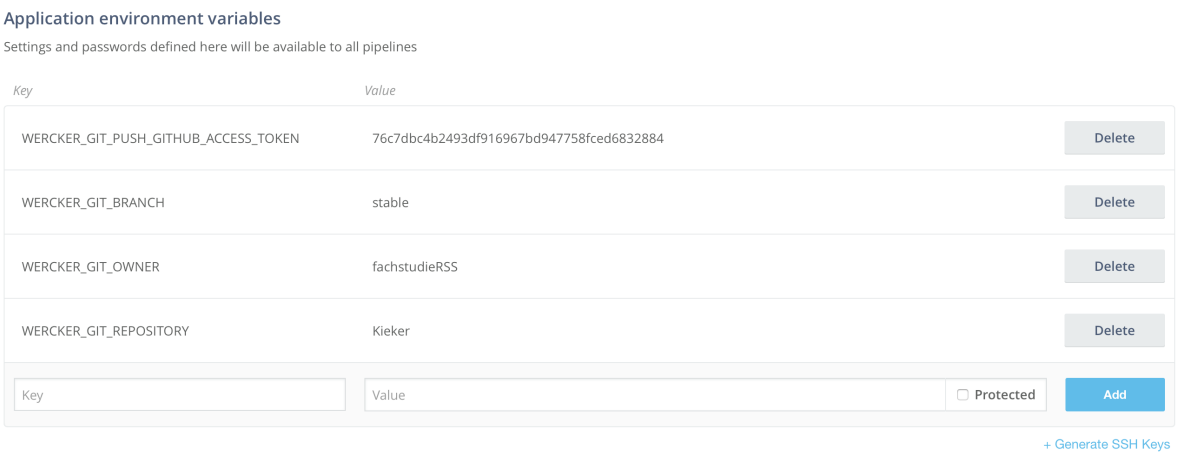

Figure 5.17: Set variables on the website

5.3.2 Build execution

The execution of a build of a pipeline can be done either automatically where a new version is detected and the execution of the pipeline starts or manually on the Wercker website with a button click. After the build has been triggered a new window pops up and the stages which have been completed or are running are shown.

5.3.3 Issues

Issues we have faced were the lack of possibility to download the stored build artifacts. Wercker shows that it stores build artifacts but there has been no functionality to download it. It was also not clear if the code base gets mounted or not. As we used a working Kieker Docker image, we needed to copy our current files into the Docker container. But as we found out, Wercker automatically starts up the defined Docker image with the code base inside, which is not documented.

In conclusion, we think Wercker is a satisfying tool and potential replacement for SnapCI with its user-friendly interface and simplicity.

Chapter 6

Evaluation

In this section we are going to compare and evaluate the three most promising tools. We will highlight individual benefits and argue how well each tool would fit in the Kieker environment. We took the following aspects into account. How difficult are configuration, setup and maintenance. How well each tool integrates into the Kieker development process and how expensive the solution is.

The first tool Wercker is the only hosted solution that made it into the final selection. One big advantage of it is that there is no setup and maintenance required. In our test setup we experienced that the configuration of a pipeline is very easy and the integration into Github works very well. The pipelines support every required Github integration, e.g. automated pull request and branch detection. Wercker was recently acquired by Oracle[Ora] which makes it one of the few tools backed by a major software company. We reckon this provides financial stability to the tool and lowers the probability of an unexpected end of service like we experienced with SnapCI from Thoughtworks. There is the risk that complex pipelines cannot be modeled in Wercker because it emphasizes simplicity. Furthermore the realization of Infrastructure as Code is very lukewarm. Stages are configured as code but the definition of a pipeline is realized via the web interface. This somewhat destroys the benefits of Infrastructure as Code. One more negative aspect is the pricing model. There is a big pricing surge if one requires more than two concurrent builds.

Jenkins and Concourse on the other hand are, compared to Wercker, relatively cumbersome to configure and maintain as we described in the previous section. As Concourse uses Docker-Compose, an automated orchestration tool to start and connect multiple containers, to have a running installation one only has to create one docker-compose file and run a single command. For Jenkins this process is slightly more complicated due to the plugin based architecture, e.g. for our use case one has to install several different plugins in order to have a working CI environment. As already mentioned before Kieker already utilizes Jenkins for nightly builds, so there is already a prepared installation that

could be used. This would reduce the effort required to setup and maintain a Jenkins installation. This is a major advantage compared to Concourse where a completely new installation is required.

Furthermore one can assume that the system administrators and Kieker contributors are already familiarized with Jenkins. Concourse on the other hand is a completely new environment and both, developers and administrators have to learn how to operate the new system. Both tools utilize Infrastructure as Code to define a pipeline. To run a build Jenkins utilizes a groovy file that defines the pipeline. Such a file has to be placed in the repository. Accordingly to change the pipeline a commit has to be pushed to the repository. This makes testing a pipeline quite time consuming because every little change has to be in a commit. Concourse on the other hand works with a command line interface that allows to directly push a file that defines the pipeline. Each change can be made without touching the repository in any way. To keep track of the configuration one can commit the configuration file to the repository after every tested change. Further, the Command Line Interface provided by Concourse allows developers and administrators to dial into a running pipeline to pull system logs and other information helpful for debugging. Concourse utilizes the CLI and configuration files for every task and is therefore the purest implementation of Infrastructure as Code. There can be no configuration that is not documented in code somewhere and no one can make changes unnoticed.

In contrast to Wercker, Concourse focuses more on adaptability, configurability and allows developers to model complex pipelines. This leads to a more complex pipeline configuration. For example to integrate Github into the system one has to manually define the resources to connect the pipeline. Wercker and Jenkins already provide preconfigured solutions.

We really like the fact that the CLI tool from Concourse is very comprehensive. From configuration to debugging and manual execution everything is combined into a single interface. As developers are used to write textual commands, using a CLI feels very intuitive. The classic Jenkins UI on the other hand is in our opinion quite confusing. But with version 2.0 Jenkins introduced the BlueOcean plugin, an alternative UI that improves the user experience of the pipeline representation. In contrast to the complex and cluttered Jenkins UI Wercker convinces with a simple UI that fits its use case. As Wercker emphasizes on simplicity the UI is reduced to the most important settings needed. But there are also shortcomings in this approach, for example there is no possibility to download build artifacts.

Another important aspect is the anticipated stability of each solution to prevent another unexpected termination. As Concourse and Jenkins are open source they are depending on active contributers to improve and further develop the tools. The community behind Jenkins is significantly larger than Concourse's, as one can tell by the number of plugins

available for example. A big community also lowers the risk of product abandonment. As Concourse is backed by a smaller community and is still relatively unknown, the probability of discontinuation is higher compared to Jenkins. Additionally a big community is capable to give better support. In case of problems it is very likely that someone in the community has already solved it.

Our last aspect are the costs to maintain the solutions. In contrast to Wercker, Concourse and Jenkins only have operational costs. There are no charges for the provided software.

To sum up, all three solutions have unique features that make them distinct from each other. Concourse is a pure Infrastructure as Code solution configured over the CLI. Jenkins is a popular tool with strong support from its community and Wercker is easy to maintain and use with low initial overhead.

Chapter 7

Conclusion

Finally, we draw a conclusion and give a recommendation for the best fitting tool.

In our opinion none of the three tools has a big advantage over the others. As we chose three tools that differ in complexity we would recommend Jenkins in the current situation. As Kieker already uses a working installation of Jenkins for the nightly builds, this is giving it the edge over the other two solutions. Additionally, Jenkins offers with its plugins the option to customize the solution as needed. Concourse on the other hand is slightly more complex and rather fits a bigger project, due to the configuration and modification one can do. These kind of in-depth configurations are not necessary for Kieker. Lastly, Wercker emphasizes on simplicity which means in this case that essential features like the artifact download are missing. Furthermore Wercker does not support full Infrastructure as Code for the definition of the pipeline.

Nevertheless, depending on the use case another solution besides Jenkins can also be suitable for the project. It really depends on the complexity of the pipeline and the infrastructure someone has access to. Even though we chose Jenkins in this case, we would recommend any of these three solutions for a fitting use case.

Appendix

Bibliography

[Bud] *Buddy: Build, Test & Deploy Code in Seconds*. https://buddy.works/. (Accessed on 09/04/2017) (cit. on p. 10).

[Cir] *Continuous Integration and Delivery - CircleCI*. https://circleci.com/. (Accessed on 09/04/2017) (cit. on p. 10).

[Cod] *Continuous Integration, Deployment & Delivery with Codeship*. http://codeship.com/. (Accessed on 09/04/2017) (cit. on p. 10).

[Dro] *Continuous Delivery | Drone*. https://drone.io/. (Accessed on 09/04/2017) (cit. on p. 12).

[Goc] *Open Source Continuous Delivery and Automation Server | GoCD*. https://www.gocd.org/. (Accessed on 09/04/2017) (cit. on p. 12).

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321601912, 9780321601919 (cit. on p. 5).

[Jena] *Blue Ocean*. https://jenkins.io/projects/blueocean/. (Accessed on 09/07/2017) (cit. on p. 15).

[Jenb] *Git Plugin - Jenkins - Jenkins Wiki*. https://wiki.jenkins.io/display/JENKINS/Git+Plugin. (Accessed on 09/07/2017) (cit. on p. 15).

[Jenc] *GitHub Plugin - Jenkins - Jenkins Wiki*. https://wiki.jenkins.io/display/JENKINS/GitHub+Plugin. (Accessed on 09/07/2017) (cit. on p. 15).

[Jend] *Jenkins*. https://jenkins.io/. (Accessed on 09/04/2017) (cit. on p. 12).

[Kie] *Kieker | Application Performance Monitoring and Dynamic Software Analysis*. http://kieker-monitoring.net/. (Accessed on 09/10/2017) (cit. on p. 1).

[Ora] *Oracle Buys Wercker*. https://www.oracle.com/corporate/acquisitions/wercker/index.html. (Accessed on 09/06/2017) (cit. on p. 27).

[Piv] *Pivotal* -. https://pivotal.io/. (Accessed on 05/09/2017) (cit. on p. 12).

[Shi] *DevOps and CI/CD automation simplified | Shippable*. https://www.shippable.com/. (Accessed on 09/04/2017) (cit. on p. 10).

[Tra] *Travis CI - Test and Deploy Your Code with Confidence*. https://travis-ci.org/. (Accessed on 09/04/2017) (cit. on p. 11).

[Wer] *Wercker Home*. https://www.wercker.com/. (Accessed on 09/04/2017) (cit. on p. 11).

All links were last followed on.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature