Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 219 (2023) 736–743

Procedia Computer Science

www.elsevier.com/locate/procedia

CENTERIS - International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies 2022

Modern Build Automation for an Insurance Company Tool Selection

Moritz Lange[a], Kim Chi Tran[a], Alexander Grunewald[a], Arne Koschel[a,*], Anna Pakosch[a], Irina Astrova[b]

[a] *Hochschule Hannover, University of Applied Sciences and Arts, Hannover, Germany*
[b] *Department of Software Science, School of IT, Tallinn University of Technology, Tallinn, Estonia*

Abstract

In this paper we describe the selection of a modern build automation tool for an industry research partner of ours, namely an insurance company. Build automation has become increasingly important over the years. Today, build automation became one of the central concepts in topics such as cloud native development based on microservices and DevOps. Since more and more products for build automation have entered the market and existing tools have changed their functional scope, there is nowadays a large number of tools on the market that differ greatly in their functional scope. Based on requirements from our partner company, a build server analysis was conducted. This paper presents our analysis requirements, a detailed look at one of the examined tools and a summarized comparison of two tools.

Keywords: build automation; build server; DevOps; evaluation; CI/CD

1. Introduction

Over time software became increasingly more complex and distributed. Proportionally, it became more and more difficult for developers to compile, link, package and test software projects with their dependencies manually. Especially with the rise of cloud native development using microservices (cf. CNCF: https://www.cncf.io/) and the large number of involved artifacts, build automation became nearly a 'must have'.

* Corresponding author. Tel.: +49-511-9296-0.
E-mail address: arne.koschel@hs-hannover.de

10.1016/j.procs.2023.01.346

For this reason, build automation tools like Make, Ant, Maven, and Gradle have emerged. Based on this, build servers have been developed. A build server is a system that checks out the current project status from a version control system (VCS) at certain times or events and then builds the project using a build automation tool described above.

If the build server responds to each check-in to the VCS, it is called Continuous Integration (CI). Build servers usually perform other tasks than just building the software, such as code analysis or automated testing to detect execution or quality issues early and ensure the stability of the software. Some build servers can even publish the built software. A distinction is made between publishing in a staging environment (Continuous Delivery (CD)) and publishing in a productive environment (Continuous Deployment (CD)) [1].

With the emergence of topics such as microservices and DevOps, build automation and in particular CI/CD is experiencing a new upswing. Small deployment units that are constantly built, tested, integrated and deployed, help significantly to enable continuous releases of – often more stable – software. As a result, a large number of build servers are currently available, which differ widely in their functional scope [1].

This article is based on a cooperation with a research partner from the insurance industry. While they operate successfully a medium scale service-oriented architecture (SOA), as well as mainframe and SAP applications, for newer demands – such as intelligent insurance risk predictions, and spontaneous insurances – they aim at microservices based solutions. In previous work [18] we had a look at microservices for such purposes. We also aim at working towards a 'Microservice Reference Architecture for Insurance Companies (RaMicsV)' jointly with our insurance industry partners [19], which gives an overall context for our work here.

However, especially with microservices a strong demand for a modern build system with good CI/CD support arrises. (Also) for that purpose the research partner is currently modernizing its build system, but is confronted with the large variety of tools. As part of our project work, a comparison of modern build servers was conducted. The results of this comparison, with – due to space limitations – an in depth look at (only) one evaluated build server, are the main contributions of this article (see [20] for a second in depth build tool evaluation of ours, which of course builds on the same evaluation process and uses the same evaluation criteria as here in this article).

The remainder of this article is organized as follows: Section 3 discusses our approach and shows how we have selected and compared different build server solutions. Section 4 evaluates in depth one particular build server, namely Jenkins, using the criteria from Section 3. Section 5 explicitly compares two evaluated build servers, and Section 6 gives a conclusion and some outlook to future work.

2. Related work

In this section we will have a brief look at some work related to ours, thus in particular at some general overview but also at particular tool evaluations for CI/CD.

Meyer gives in [21] an overview on CI and its tools', which is helpful for a general look at the topic. More strategically oriented, Jin and Servant discuss in [22] strategies to improve CI.

A systematic overview on continuous integration, delivery and deployment, which includes approaches, tools, challenges, and practices is shown by Shahin, Babar, and Zhu in [23]. Singh, Gaba, M. and B. Kaur provide a cloud platform oriented comparison of different CI/CD tools in [24]. Probably closest to our work, although from 2015, is [25] by Rai et al. They look at Jenkins with a comparative scrutiny of various software integration tools.

Naturally however, none of the articles follows the particular evaluation steps and criteria as they were emphasized by our partner company, which is a good example for (at least the German) insurance business. This, thus, is the major contribution of our work here, although might be generalizable and be of value for at least some other insurance companies as well.

3. Comparison – Our Approach

In order to find suitable solutions for the partner to modernize its build system, our comparison approach is divided into five steps. They are explained below in chronological order.

Step 1: Restricting the Subjects for Comparison. After a training phase we have created a list of six build servers with a rough overview of each one. Afterwards we presented the list to the partner and in several discussions we reduced the number of subjects to three.

Jenkins [2,3,4,5], which is currently used by the partner, Travis CI [7,8,9,10,11,12,13], and GitLab CI [14,15,16,17] were selected for the comparison. Due to space limitations in this article we will only present our in depth evaluation results for Jenkins. Our in depth GitLab CI evaluation is presented in [20] and the in depth evaluation of Travis CI shall be presented in future work.

Step 2: Analysis of the Research Subjects. After the build servers were identified, we assigned a project participant to each tool to take a closer look.

Step 3: Development of Comparison Criteria. After each project participant has become an expert with a specific build server, we worked out a number of possible criteria to compare the products. Together with the partner, we discussed and prioritized the criteria. We have identified the following four categories of comparison criteria by which the build servers are compared and analyzed:

- Internals: In this category we roughly look at the architecture of each system and we check, how the build process works internally.
- Pipelines: This category deals with the topic of pipelines. It is analyzed how a developer can model the build process in each tool, i.e. how pipelines can be created. Furthermore, the pipeline-relevant features (e.g. an artifact store, image store or caching in the pipeline) of each tool are discussed in this category.
- Additional Features: This category deals with the features that go beyond the functionality of a build server. An example could be a built-in Wiki or a ticketing system.
- Platforms: This category analyzes supported platforms on which the build server can be run. We also investigate whether different versions of the software are available and how they differ.
- License and Pricing: The last category of comparison deals with the license and the costs associated with the operation of the software.

Step 4: Development of a Reference Scenario. After analyzing the build servers according to the categories mentioned above, we developed a reference scenario in cooperation with the project partner. The scenario was created based on the partner's previous considerations for modernizing the build process. The result is a sequence of tasks which have to be implemented by the system using specific technologies. Each system must be able to map the following process to meet the partner's requirements:

1. Reacting to a commit on a git server.
2. Building the project with Maven.
3. Testing the project with Maven.
4. Create a Docker image with the executable jar file.
5. Push the created Docker image to a Docker registry.
6. Initiate image deployment on a Kubernetes cluster.

Step 6 is optional, because if the image is in a Docker registry, the deployment can be initiated manually without much effort and the implementation of Continuous Deployment is not necessary. Within this work the above workflow was implemented with the examined build servers. A 'Hello World' Spring Boot 2.0 web application served as an example.

Step 5: Concluding Comparison of the Analyzed Subjects. Finally, all project participants came together and presented the results to each other. A final comparison was made and a recommendation for the partner was worked out.

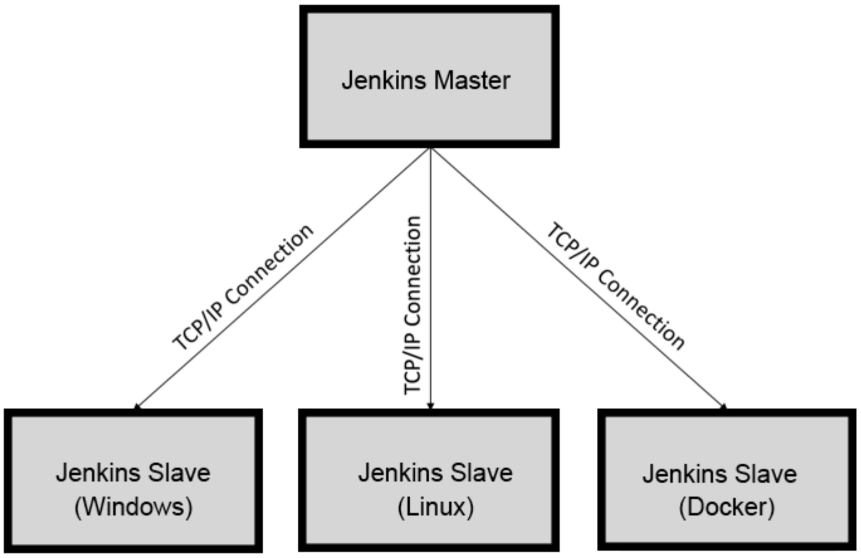

Fig. 1. Jenkins Master/Slave Architecture.

4. CI/CD Tool Evaluation: Jenkins

Jenkins is a Java-based open source CI tool, which is used by teams of all sizes and for projects in a wide variety of languages and technologies. Compared to other CI tools Jenkins has a significant market share and built up a huge community over the years. Jenkins can be controlled and configured through a simple and intuitive web user interface: the Jenkins dashboard. The tool is flexible and easy to adapt to custom requirements, because its functionality can be extended through hundreds of plugins provided by the community. For example, plugins can ensure the integration of different VCS, build tools or code quality metric tools. The community, which is described as a large, dynamic, reactive and welcoming bunch, is one of the reasons why Jenkins is so popular. Driven by the community the development pace of Jenkins and its plugins is fast: New features, bug fixes and updates of plugins are released on weekly bases (see [2, pp. 2–3]).

4.1. Internals

This section describes the architecture of Jenkins and how builds are executed. Jenkins uses a master/slave architecture shown in Figure 1. There is a main master server, on which Jenkins is installed. Its tasks are scheduling build jobs, dispatching them to the slaves, monitoring the slaves and recording and presenting the build results. Jenkins slaves are the main job-executing instances, though the master can also execute build jobs directly. A slave is a Java executable that runs on a remote machine, which can therefore run on a variety of operating systems. Slaves communicate with the master instance over a TCP/IP connection (see [2, p. 305]).

Slaves can be created manually through the Jenkins dashboard. If Jenkins is used in a cloud environment like AWS, Azure, Google Cloud or VMWare, slaves are managed automatically by the cloud [3]. A build job for execution on a slave can also be created with the dashboard. A build job in Jenkins can be triggered by different events like a Git commit, time-events or manually through the dashboard. When running a job, Jenkins clones the corresponding project from the defined VCS, e.g. Git or Subversion. Next, the build steps are executed. They are the basic building blocks of a job. In a default Jenkins installation steps to invoke Maven and Ant, as well as running OS-specific shell or Windows batch commands, can be added to the job. Further steps can be added by installing appropriate plugins. After executing build steps, post-build actions like archiving generated artifacts, reporting on test results or notifying people about the build results are executed (see [2, pp. 81-113] [3]).

4.2. Pipelines

Jenkins supports implementing continuous delivery pipelines with the help of a suite of plugins. A pipeline can be defined by writing Groovy code in the web UI or into a Jenkinsfile, which is stored in the project's VCS. While the syntax is basically the same in both cases, providing a Jenkinsfile is generally considered best practice. The actual code for a pipeline can be written using two types of syntax - Declarative and Scripted. While Declarative is a more recent feature of Jenkins Pipeline, which presents a more simplified, declarative syntax, Scripted is a procedural domain specific language (DSL) built with Groovy. Furthermore, Scripted pipelines can be more expressive and flexible, because they provide more functionality of the Groovy language like flow control. However, both have fundamentally the same pipeline sub-system underneath and only differ in syntax and flexibility. For simplicity, this paper examines only the declarative syntax type.

Below we show the Jenkinsfile used to implement the reference scenario with a declarative syntax:

```
pipeline {
  agent any
  options {
    skipStagesAfter Unstable()
  }
  stages {
    stage ( 'Checkout') {
      steps {
        git 'https://example.com/git/exampleProject.git'
      }
    }
    stage ( 'Build') [
      steps {
        sh '$fmvnHomeg/bin/mvn -B -D skipTests package'
      }
    }
    stage ( ' Test ' ) {
    steps {
      sh '$fmvnHomeg/bin/mvn surefire:test -B'
      }
    }
    stage ( ' Docker Build ' ) {
    steps {
      sh 'docker build . -t myapp'
      }
    }
    stage ( ' Docker Push ' ) {
      steps {
        sh 'docker push apps/myapp'
      }
    }
  }
}
```

- *pipeline* defines a block containing all content and instructions for executing the entire pipeline [4].
- *agent* instructs Jenkins to allocate a slave that executes either the pipeline or a specific stage in the Jenkins environment depending on where the agent section is placed. It can be defined at the top-level inside the pipeline block or inside the stage at each stage-level. The keyword *any* selects any available slave [4].
- *options* defines general options for the pipeline. In case of a stage failure subsequent stages should be skipped.
- *stage* contains a steps section, an optional agent section, or other stage-specific directives. It is basically the main section to define the work done by the pipeline [4].

- *steps* defines a series of one or more steps to be executed in a given stage directive. It contains the commands for execution of specific tasks within the pipeline [4].
- *git* and *sh* are steps for the execution of commands [4].

Every pipeline should contain the components: *pipeline, agent, stage* and *steps*. The *git* and *sh* statements are specific for this example. Additional features like the integration of an artifact-store are provided through the plugin-mechanism of Jenkins, which is excluded for simplicity. By adding features via the Jenkins dashboard, new commands can be used in the *steps*-section (see [2, pp. 285–304], [4]).

4.3. Additional Features

Jenkins is a highly expandable platform for CI/CD. The plugin mechanism provides hundreds of extensions for a Jenkins installation. Due to the community more and more plugins are released on weekly bases. Additional features like notifications, issue tracking and documentation can easily be added through the Jenkins plugin manager in the dashboard. Furthermore, Jenkins can also be used to publish built software to environments like Kubernetes. On one hand there are plugins for pretty much every cloud-based Kubernetes environment like Google Compute Engine or Microsoft Azure, on the other hand there are also plugins for self-hosted Kubernetes instances. In conclusion the additional features of Jenkins are limitless, because plugins for desired features can be written by everyone. For this reason Jenkins can support a complete DevOps workflow: Development, IT-Operations and quality assurance aspects can be brought together within the Jenkins environment (see [5]).

Another feature, which needs to be mentioned, is Jenkins X, which is a new distribution of Jenkins, that combines Jenkins with built in Docker and Kubernetes support. That means, that no more plugins are required to implement Docker and Kubernetes to the environment. A new feature, that Jenkins X automatically creates, is a Git repository aswell as the Dockerfile, Helm chart and the Jenkinsfile needed for a project, whenever a project is imported. Since Jenkins X is a fairly new distribution, this paper is focused on the original Jenkins. However, Jenkins X should nevertheless be investigated in future work to test, if it is a viable alternative to Jenkins for the research partner.

4.4. Platforms, License and Pricing

Jenkins is self-hosted whether on own infrastructure or in a cloud environment. It is available in two different versions: Long-Term Support (LTS) and Weekly. It is possible to switch between those two.

- LTS: This version is for companies, that have their own private branches of Jenkins for stabilization and internal customizations. It sticks to a release line, which changes less often and only receives important bug fixes, even if such a release line lags behind in terms of features [4].
- Weekly: New releases are delivered on weekly bases to provide new bug fixes and features. This version is for users like plugin developers, that need to be up-to-date at any given time (see [4]).

Currently both versions are available for Docker, FreeBSD, Gentoo, macOS, OpenBSD, OpenSUSE, Red Hat/Fedora/CentOS, Ubuntu/Debian, Windows and as a Generic Java package with a weekly version for Arch Linux as well (see [2, pp. 3], [4]).

Since Jenkins is published under the MIT license, any person can ‘use, copy, modify, merge, publish, distribute, sublicense and/or sell copies of the Software’ (see [6]). In other words: everybody can use it for free.

5. Evaluation Summary

We analyzed GitLab in depth in our previous work [20] independently under the specified criteria from Section 3. In this article we added the in depth analysis of a recent version of the Jenkins build tool (which – in older versions – is the current tool in use at our partner company). To tie up the analysis and to compare the build servers, Table 1 shows our summarized results for Jenkins and GitLab.

At first glance the internals of the build automations tools look differently, but they all use a similiar underlying master/worker architecture. While GitLab pipelines are defined declaratively using YAML files, Jenkins pipelines are

Table 1. Comparing Jenkins and GitLab

Criteria	**Jenkins**	**GitLab**
Internals	Master/Slave architecture	GitLab CI delegates builds to GitLab Runners
Pipelines	Groovy file (declarative or procedural style)	YAML file (declarative style)
Additional Features	Theoretically unlimited through community plugins	Many features built-in
Platforms	Self-hosted, Open source and cross-plattform	SaaS or Self-hosted, Open source and cross-platform
License and Pricing	Free to use with MIT License Paid support for special distributions	Community Edition free to use with MIT License Plans for Enterprise Edition

defined declaratively or procedurally with Groovy via Jenkinsfiles. The procedural style gives Jenkins users more control over the internal flow of the pipeline, but requires more expertise.

GitLab and Jenkins both support a full DevOps workflow. Jenkins through its plugin mechanism and GitLab through its many built-in tools.

There is no official SaaS solution for Jenkins. GitLab offers such a solution. Each examined tool can be run on premise.

Both tools offer versions that are available under the MIT license [6]. While Jenkins is completely open source and free to use, this is only partially true for GitLab.

6. Conclusion

Within the scope of this paper the market for build servers was examined. In cooperation with an industrial partner, three representatives were selected and a more in depth evaluation of them was undertaken. The detailed evalutaion of one of them and the brief evaluation results of two tools are presented in this article.

Comparison criteria were developed and grouped into categories. First the tools were examined independently. Then their similarities and differences were identified. To sum up things, there are many tools that vary largely in their functionality. It is difficult to make a statement about which tool is the best, because different tools address different needs.

In [20] we presented the in depth evaluation of GitLab and in this paper we compared it to the Jenkins based solution in Section 5. For our industry partner we recommend, that the functionality of the tools should be checked again against more detailed requirements. Jenkins as the currently used tool can meet all requirements known to us, but has to be equipped with a lot of plugins and configurations in order to do so.

Beside GitLab and Jenkins, our partners also wanted to look in depth at the Travis CI build tool solution, which we presented so far only briefly in [20]. Thus, we shall present an in depth evaluation of Travis CI and the final overall comparison of all three build tools in a future article of ours.

References

[1] P. M. Duvall, S. Matyas, and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk. Pearson Education, 2007.
[2] J. F. Smart, Jenkins: The Definitive Guide - Continuous Integration for the Masses. Sebastopol: O'Reilly Media, Inc., 2011.
[3] kohsuke. (2016) Use jenkins. [Online]. Available: https://wiki.jenkins.io/display/JENKINS/
[4] J. IO. Jenkins user documentation. [Online]. Available: https://jenkins.io/
[5] S. J. Bigelow. Jenkins-unterstuetzt-eine-umfassende-devopskultur. [Online]. Available: https://www.computerweekly.com/de/tipp/Jenkins-unterstuetzt-eine-umfassende-DevOps-Kultur
[6] Massachusetts Institute of Technology. The MIT License. [Online]. Available: https://github.com/jenkinsci/jenkins/blob/master/LICENSE.txt,

[7] Travis CI. (2019) Travis CI user documentation. [Online]. Available: https://docs.travis-ci.com/
[8] T. CI. (2019) Travis CI - test and deploy with confidence. [Online]. Available: https://travis-ci.com/plans
[9] Travis CI. (2019) Travis CI enterprise - Travis CI. [Online]. Available: https://docs.travis-ci.com/user/enterprise/
[10] T. CI. (2019) Free continuous integration platform for Github projects. [Online]. Available: https://github.com/travis-ci/travis-ci
[11] ——. (2019) The Travis CI blog: Setting up a CI/CD process on Github with Travis CI. [Online]. Available: https://blog.travis-ci.com/2019-05-30-setting-up-a-ci-cd-process-on-github
[12] Atlassian. (2019) Travis for Jira Jatlassian marketplace. [Online]. Available: https://marketplace.atlassian.com/apps/1220191/travis-for-jira
[13] T. CI. (2019) The Travis VI blog: Travis CI joins the Idera family. [Online]. Available: https://blog.travis-ci.com/2019-01-23-travis-ci-joins-idera-inc
[14] GitLab Inc. (2019) The first single application for the entire devops lifecycle. [Online]. Available: https://about.gitlab.com/
[15] ——. (2019) Gitlab CI/CD documentation. [Online]. Available: https://docs.gitlab.com/ee/ci/
[16] ——. (2019) Gitlab feature overview. [Online]. Available: https://about.gitlab.com/features/
[17] ——. (2019) Gitlab pricing. [Online]. Available: https://about.gitlab.com/pricing/
[18] C.v. Perbandt, M. Tyca, A. Koschel, I. Astrova, "Development Support for Intelligent Systems: Test, Evaluation, and Analysis of Microservices", in: SAI IntelliSys 2021. LNCS, vol 294. Springer, 2021.
[19] A. Koschel, A. Hausotter, R. Buchta, A. Grunewald, M. Lange, P. Niemann, "Towards a Microservice Reference Architecture for Insurance Companies", in IARIA SERVICE COMPUTATION 2021, The 13th Intl. Conf. on Advanced Service Computing, Online, Thinkmind, 2021.
[20] A. Grunewald, M. Lange, K.C. Tran, A. Koschel, and I. Astrova, "A Case for Modern Build Automation for Intelligent Systems", SAI IntelliSys 2022, to be published in Springer LNCS, 2022.
[21] M. Meyer, "Continuous Integration and Its Tools", in IEEE Software, vol. 31, no. 3, pp. 14-16, May-June 2014.
[22] X. Jin and F. Servant, "What Helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration", 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 213-225.
[23] M. Shahin, M. Ali Babar and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", in IEEE Access, vol. 5, pp. 3909-3943, 2017.
[24] C. Singh, N. S. Gaba, M. Kaur and B. Kaur, "Comparison of Different CI/CD Tools Integrated with Cloud Platform", 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence), 2019, pp. 7-12.
[25] P. Rai, Madhurima, S. Dhir, Madhulika and A. Garg, "A prologue of JENKINS with comparative scrutiny of various software integration tools", 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), 2015, pp. 201-205.