MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

# Jenkins Virtual Pipeline

Bachelor's Thesis

TOMÁŠ HAVLÍČEK

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

# Jenkins Virtual Pipeline

Bachelor's Thesis

TOMÁŠ HAVLÍČEK

Advisor: RNDr. Adam Rambousek, Ph.D.

Department of Machine Learning and Data Processing

Brno, Spring 2023

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

# Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Havlíček

**Advisor:** RNDr. Adam Rambousek, Ph.D.

## Acknowledgements

I would like to thank my advisor RNDr. Adam Rambousek, Ph.D., for the continuous support and feedback. I would also like to thank Mgr. Jiří Vaněk for great technical insights and for answering all my questions regarding the thesis. Last but not least, I would like to thank my family and friends for their support during the course of my studies.

# Abstract

The target of this thesis is to implement a plugin for an automation server Jenkins. When investigating whether Jenkins build went right or wrong, it usually only takes a few lines to get the wanted information about the build. However, the important lines can get lost in the full log. The plugin should allow users to create custom marks, match these marks with lines in the log, and visualize the result so that the user can easily access the exact place of the given mark. Furthermore, the plugin should have more supporting features, for example, for working with bigger logs. This should improve the user experience when searching for expected information in a Jenkins log. As breaking down one big log into small defined parts somewhat mimics the behavior of Jenkins Pipeline, the plugin is called Virtual Pipeline.



# Keywords

Jenkins, Continuous Integration, Continuous Delivery, Plugin, Java, Jelly, Pipeline, Maven, Java, User Experience, Freestyle Project

# Contents

**Introduction** **1**

**1 Jenkins** **2**
1.1 Project Jenkins . . . . . 2
1.2 Continuous Integration/Continuous Delivery . . . . . 2
1.3 Jenkins structure and user interface . . . . . 3
1.3.1 Jenkins overview . . . . . 3
1.3.2 Project types . . . . . 3
1.3.3 Console logs . . . . . 4

**2 Jenkins Development and Extensions** **6**
2.1 Design . . . . . 6
2.1.1 Open source project . . . . . 6
2.1.2 Extensibility . . . . . 6
2.2 Development . . . . . 7
2.2.1 Development environment . . . . . 7
2.2.2 Creation of a plugin . . . . . 8
2.2.3 Deploying a plugin . . . . . 9
2.2.4 Testing a plugin . . . . . 10
2.2.5 Jelly . . . . . 10

**3 Comparison to similar features** **12**
3.1 Blue Ocean . . . . . 12
3.2 TeamCity feature . . . . . 13
3.2.1 Build chain . . . . . 13
3.3 Other Plugins . . . . . 14
3.3.1 Log Parser Plugin . . . . . 14
3.3.2 Log File Filter Plugin . . . . . 14

**4 Plugin design and implementation** **15**
4.1 Design and model . . . . . 15
4.1.1 Use cases . . . . . 15
4.1.2 Functional requirements . . . . . 15
4.2 Plugin steps . . . . . 16
4.2.1 Configuration . . . . . 17
4.2.2 During building . . . . . 17

4.2.3 After building . . . . . 18
4.2.4 Multiple builds . . . . . 18
4.3 Implementation dilemmas and problems . . . . . 18
4.3.1 How to display logs in Jenkins interface . . . . . 19
4.3.2 Adressing lines on another page . . . . . 19
4.3.3 Displaying image on Build page and Project Page 20
4.4 Implementation . . . . . 21
4.4.1 Plugin structure . . . . . 21
4.4.2 Classes . . . . . 22
4.4.3 Cache JSON format . . . . . 27
4.4.4 Form input . . . . . 28
4.4.5 Jelly scripts . . . . . 28
4.4.6 Used libraries . . . . . 29

**5 Conclusion** **30**

**Bibliography** **31**

# List of Figures


1.1 Example of Jenkins logs displayed by default . . . . . 4

3.1 Example of Blue Ocean visualization . . . . . 12
3.2 TeamCity Build chain feature example . . . . . 13

4.1 Diagram of plugin actions . . . . . 16
4.2 Script for prefilling the Offset action parameters . . . . . 20
4.3 Project structure . . . . . 22
4.4 Form validation . . . . . 28
4.5 Part of plugin POM . . . . . 29

# Introduction

In software development, there are great advantages to automating tasks. When tasks are automated, they are more prone to errors caused by human mistakes. In addition, we automate the repetitive and boring tasks, so the developer can shift his focus to the tasks that require creative solutions. A crucial part of development is testing and fixing errors. When we run our program and something that is not supposed to happen happens, we want to investigate what happened. Most programs leave some kind of log, a text output of what happened during the run of the program. As programs get bigger and more complex, so do these logs. To ease the need for manually reading large logs full of irrelevant information, we can develop tools to pinpoint the information we need. Even though it does not seem to matter at first, it is important to have tools that clearly show what happened and what possibly went wrong.

The target of this work is to create such a tool that helps users of Jenkins, an automation server, to find out what happened or what went wrong based on the logs. This should replace the manual reading of logs by highlighting user-defined parts of the log. Since different logs contain various information, it should be up to the user to define what is useful information for the tool to search for.

The first chapter should be a brief introduction to CI/CD and Jenkins Project. In the second chapter, we take a look at what it takes to develop Jenkins extension. In the third chapter, we compare the functionality of our plugin to already existing alternatives and similar features of other plugins. Finally, in the fourth chapter, we see how the plugin is designed, implemented and what issues arose during development.

The aim of this work is to develop a working Jenkins plugin that makes reading logs easier by highlighting the important parts defined by the user.

# 1 Jenkins

The aim of this thesis is to create a Jenkins plugin that defines marks, finds them, and displays them. To understand what are the conditions for creating such a plugin, this section focuses on basic information about Jenkins and should serve as a basic introduction to Jenkins Project itself.

## 1.1 Project Jenkins

Jenkins [1] is an open-sourced automation server for building, deploying and automating various projects [2]. The project itself was forked from another project, Hudson, which shut down in 2020 and most of the developers moved to Jenkins [3]. Due to a huge community support and extensibility, it has over eighteen-hundred plugins and over three hundred thousand known installations as of 2022 [4].

## 1.2 Continuous Integration/Continuous Delivery

Jenkins serves as a Continuous Integration and Continuous Delivery (CI/CD) tool. The main idea of this concept is to automate repetitive tasks when developing software.

Usually, the practice is that the developer writes a new piece of code for a larger project and pushes it to a project repository. To make sure the code does not break any existing functionality and has the same code style, predefined tests are performed on the new code. In case the tests fail, the developer is notified. After the errors are fixed, the code can be released into a repository and become part of the main project.

Continuous integration tools ensure that the proper tests are run on the code. This prevents the developer from making mistakes when testing and integrating the code manually, as these steps are done for the developer by the server.

The same applies for delivering the code to the project repository. The more complex the project is, the more extra manual steps need to

be taken to integrate and deliver the code, the more the more needed are supportive Continuous Integration/Continuous Delivery tools.

## 1.3 Jenkins structure and user interface

It is necessary to become acquainted with the user interface fundamentals to understand the plugin functionality.

### 1.3.1 Jenkins overview

The point of Jenkins the plugin is targeting is the browser interface of the end user; therefore, the interface is represented by a webpage in a browser.

The main page displays a structured list of Projects called View. Apart from the default overview View, it is possible to create numerous custom Views.

Project represents the main utility of an automation server. It has its own dedicated workspace and can be configured and extended by plugins to perform a series of actions.

These series of actions are performed and result in a Build. Each performed series (Build) has its own unique number within the scope of the Project.

### 1.3.2 Project types

There are multiple types of projects in Jenkins. This section should cover the main ones needed to develop our plugin.

#### Freestyle project

Freestyle project is our target project type. It is used for building the project. Jenkins allows you to configure your project according to your needs, for example, building from a remote repository. Furthermore, you can add multiple build steps and post-build steps.

There is a huge variety of possible steps, usually plugins created to provide specific functionality. Thanks to this variety, you can find suitable plugins and functionalities for any project.

### Pipeline

Pipeline [5] is a newer addition to the Jenkins project than the Freestyle project. It allows us to divide steps (things we want to get done, for example, shell command) into Stages. Steps in the same stage can be run in parallel. The great functionality is that it can be defined as a code, given that it follows the Pipeline syntax. Usually, the file that contains the defined pipeline as a code is called Jenkinsfile.

Our plugin will not be extending this type of project since Pipeline has a different structure from Freestyle project, but some of the plugin features may be inspired by the functionalities of a Pipeline.

## 1.3.3 Console logs

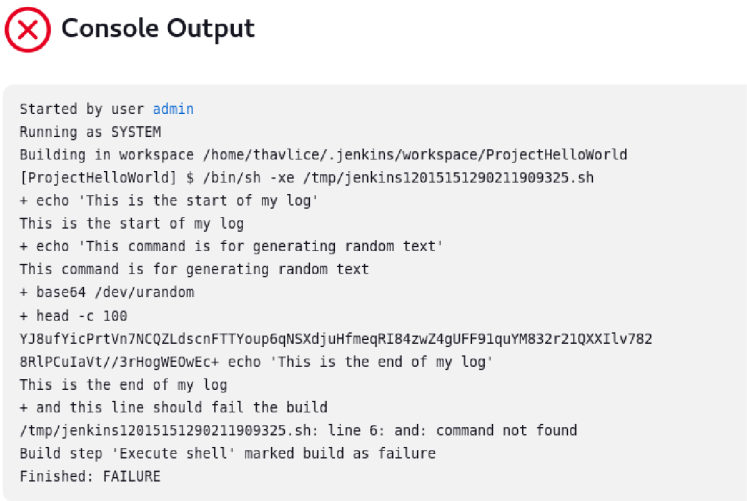


**Figure 1.1:** Example of Jenkins logs displayed by default

Jenkins provides us with functionality of logs for each Build. The log represents the main stream of output information of a Build.

While building, the log lines are saved to a file in the build folder with the corresponding build number [1]as a file named “log."

1. /jobs/PROJECTNAME/builds/BUILDNUMBER/

The results represented by this file are then displayed on the Build page sidebar; however, the logs themselves are not markuped, apart from the link of the user who started the build.

This means that changing the way the logs are displayed is not an easy thing to do.

To change their appearance, we would either need to work with a file in byte format or to work with logs in HTML, where all of the logs are represented by one HTML element.

# 2 Jenkins Development and Extensions

## 2.1 Design

### 2.1.1 Open source project

As Jenkins is an open-source project, it is encouraged for everyone to contribute to Jenkins [6]. Most of Jenkins uses an MIT license [7], so the plugin is provided with an MIT license as well.

After a plugin is finished and properly tested, the developers may request [8] a review for the plugin.After the plugin is reviewed and potentially updated according to the review, it can be accepted and hosted as a part of Jenkins in the jenkinsci[1] repository.

### 2.1.2 Extensibility

To implement functionality with a plugin in Jenkins, the plugin needs to be built upon one of the extension points of the Jenkins Project. In this section, we will go through some of the options which were considered while developing this plugin.

#### Extension points

Since the whole project is written in Java, the extension points are provided in a form of Java classes. The whole list of extension points [9] contains classes related to different functionalities of Jenkins. Jenkins also allows creation of new extension points [10].

The chosen class, from the Hudson package containing all following classes [11], for my plugin is Publisher[2], a class that implements BuildSteps[3] that are performed after the build is completed. Publisher has two subclasses: Notifier, as described in the documentation[4]: "Notifier is a kind of Publisher that sends out the outcome of the builds to other systems and humans." and Recorder[5], described: "Recorder is a

1. https://github.com/jenkinsci/jenkins
2. https://javadoc.jenkins.io/hudson/tasks/Publisher.html
3. https://javadoc.jenkins.io/hudson/tasks/BuildStep.html
4. https://javadoc.jenkins.io/hudson/tasks/Notifier.html
5. https://javadoc.jenkins.io/hudson/tasks/Recorder.html

kind of Publisher that collects statistics from the build, and can mark builds as unstable/failure".

Because the plugin functionality relies on processing and visualizing logs which are the result of a build, the chosen class is the latter option, Recorder.

## 2.2 Development

This section should cover the initial steps needed to set up a development environment and to create and deploy a custom Jenkins plugin.

### 2.2.1 Development environment

The most common way to build and manage a Jenkins plugin project is to use Maven. Alternatives such as Gradle are possible to use, but Maven is the recommended tool because it is more supported for developing Jenkins [12].

The opposite applies to choosing an Integrated Development Environment (IDE) as generally more options are available. The best fitting IDEs are IntelliJ IDEA, Netbeans or Eclipse. Again, alternatives are possible to use, but these are the most supported and comfortable to use choices.

#### Java/JDK

Since Java is the main programming language of Jenkins as well as my plugin, we need to provide a Java Development Kit (JDK) to build our plugin. The recommended version by Jenkins is JDK 11 [12].

#### Maven

We are using Maven (version 3.8.1) to build our project. Maven contains a Maven plugin [13] called HPI, which is used for creating a hpi file from the plugin. This Maven plugin also allows us create and test the plugin.

In addition, there are many other Maven plugins which support the development even more, such as spotbugs, a plugin which checks

your code for common code issues and bugs. We can execute spotbugs using command:

```
$ mvn spotbugs:check
```

#### IDE

There are IDE plugins specifically designed to support Jenkins development. It is the Jenkins Development Support plugin for IntelliJ IDEA [14] and Netbeans plugin for Jenkins Development [15]. These plugins make some aspects of Jenkins editing easier, for example editing Jelly files.

#### Jenkins instance

To run a Jenkins plugin, we obviously need some kind of Jenkins instance at which we can run the plugin.

There are multiple ways to run a Jenkins instance depending on the operating system or virtualization way [16], but the most accessible way is to run a WAR file. The WAR file can be downloaded as a single file on the official Jenkins website[6] and then run using a single command:

```
$ java -jar jenkins.war
```

### 2.2.2 Creation of a plugin

To ensure our plugin is correctly set up from the start and has the right POM, there are some helpful archetypes which we can use as a starting point. To create a plugin, we can execute this command [17]:

```
$ mvn -U archetype:generate -Dfilter="io.jenkins
  .archetypes:
```

Maven will give us some possible archetypes it can generate, one of which is a "hello-world-plugin". Choosing this archetype generates basic project structure, POM and an example Build step, which writes "Hello, world!" to the console.

6. https://www.jenkins.io/download/

You can build, deploy and run this Hello world plugin to ensure you have set up everything correctly. There is an alternative to the "hello-world-plugin" archetype, choosing "empty-plugin" will generate just empty structure and POM.

### 2.2.3 Deploying a plugin

Let us say we already have a functionality programmed in Java and we want to use it or check if it works properly. First of all, we need to build our plugin. We can do so by executing this command[18]:

```
$ mvn install
```

It is going to build the plugin and generate the results of a build in the "target" directory, including a hpi file named after the artifact Id, so in our case virtual-pipeline.hpi.

After this is performed, we can deploy our plugin. There are two ways to do it. First one is to upload the hpi file to a running Jenkins instance, under:

```
Manage Jenkins > Plugin Manager
> Advanced options > Deploy plugin
```

This is going to integrate the plugin into the instance and can be used right away, except in some cases where you need to wait for the restart of Jenkins instance.

Second way is run a new jenkins instance using maven command [18]:

```
$ mvn hpi:run
```

This will run and deploy our plugin to a new Jenkins instance running explicitly on port 8080. This parameter can be changed by running and adding the following flag:

```
$ mvn hpi:run -Dport=5432
```

The user interface of the Jenkins instance can then be accessed [18] at the URL in your browser at http://localhost:5432/jenkins/.

### 2.2.4 Testing a plugin

Essential part of every development is testing. We can use tools provided in IDE, syntax errors are shown while programming or shown while building. Other than that, Maven spotbugs plugin or tests running during building can provide more debugging information.

Finally, testing can be performed in a running Jenkins instance, while checking the output console for errors while using the plugin features.

### 2.2.5 Jelly

Jenkins uses dedicated script language called Jelly to define its web user interface (UI).

Jelly itself is a XML-like language [19] that defines the behavior of a webpage and can use some methods of classes bound to this script.

There are some basic types of Jelly scripts, based on the names and their use:

- config.jelly
  - setting configuration of a class
- index.jelly
  - general looks of the page
  - can be specified to edit parts of the page, like sidebar
  - if bound in the root folder, serves as a description of the plugin
- summary.jelly
  - displays a defined summary of the build on the Build page
- floating box.jelly
  - displays a defined summary of the last build on the Project page, usually on the right side

To create a Jenkins-specific UI, we need to use some Jenkins specific libraries [20]:

- /lib/layout
  - contains main elements of each Jenkins page displayed, such as main-panel or side-panel
- /lib/form
  - contains tools based around user input for creating forms, such as combo boxes or select elements
- /lib/hudson
  - contains more Jenkins specific elements related to the functionality of Jenkins, such as references to builds or queue

For a better understanding on how to use each element correctly, you can refer to a Jenkins Design Library Plugin [7], which explains most of the elements with code examples.

7. link: `https://github.com/jenkinsci/design-library-plugin`

# 3 Comparison to similar features

The idea for the main feature of the plugin was inspired by another automation server feature, TeamCity from JetBrains.

There are some plugins in Jenkins that create similar features or implement our desired features just partly.

The name of my plugin is also inspired by the Pipeline approach of these features.

## 3.1 Blue Ocean

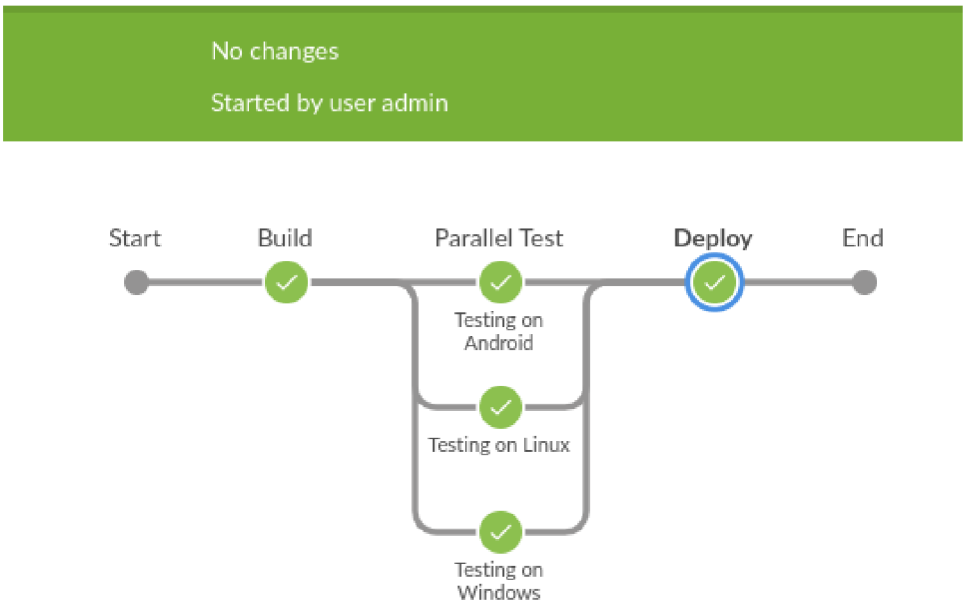


**Figure 3.1:** Example of Blue Ocean visualization

Blue Ocean is a plugin which presents a new look for Jenkins. Its main features are user-friendly editing and visualizing Jenkins Pipelines. These pipelines can be divided into steps, for example Test step, Build step and Deploy step.

Therefore it is easy to distinguish a failed step, which is one of the main functionalities of my plugin. However since Blue Ocean performs these functionalities on Pipelines and not Freestyle Project, it is not a suitable plugin to fulfill our needs.

In addition, Blue Ocean is no longer developed [21]. Its functionalities are provided and developed in Pipeline:Stageview [22] or Pipeline Graph View [23].

## 3.2 TeamCity feature

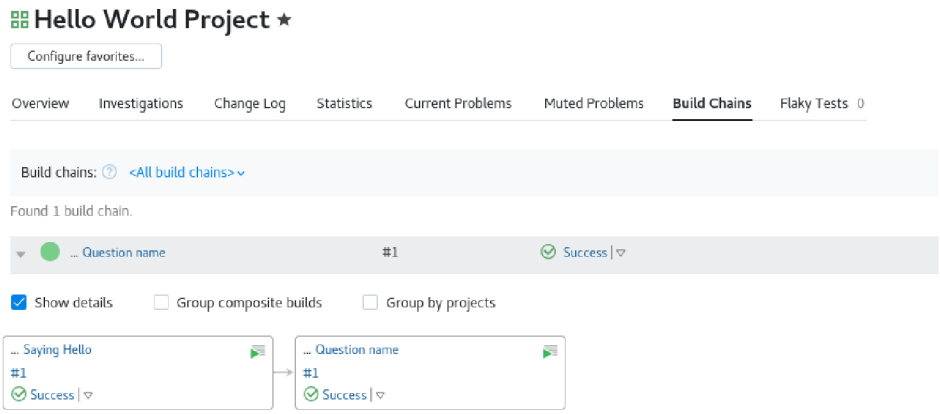


**Figure 3.2:** TeamCity Build chain feature example

TeamCity [24] is a CI/CD automation platform from JetBrains. It is a whole another automation server, meaning it might differ in approach to functionalities compared to Jenkins plugins or Jenkins itself.

### 3.2.1 Build chain

The feature we are focusing on is Build Chains. The idea is very similar to Jenkins Pipeline strategy, dividing the build process into steps and creating dependencies between steps. When a build fails as a whole, we can easily pinpoint the part that failed and for what reason.

We cannot obviously use this feature because we are looking for a solution in Jenkins.

However, it is meaningful to look at the approach to our problem from the perspective of another automation server.

## 3.3 Other Plugins

### 3.3.1 Log Parser Plugin

Log Parser Plugin [25] takes a user-defined set of rules (what is an error or warning) and applies them to the log. Then it highlights the results of the rules in the log and provides a numerical summary of the findings.

This plugin shares some of the wanted features; however, it focuses on making the full log more readable, but it is lacking the important "find where the build failed" functionality and some other features.

If the use case is to only visualize the standard Jenkins log better, this plugin is a good alternative to Virtual Pipeline.

### 3.3.2 Log File Filter Plugin

Log File Filter [26] is another plugin that has somewhat intersecting functionality with Virtual Pipeline. The main purpose of the plugin is to censor sensitive information in the Jenkins build log, for example, passwords used to execute commands in the build.

To match and replace the sensitive lines, it uses regex patterns in a way compared to Virtual Pipeline. However, the functionality ends there, so this plugin was not enough to fulfill our functional requirements.

# 4 Plugin design and implementation

The general need for this plugin arose from having to manually search the logs for the error or failed test which caused the build to crash. The following section contains information about design and development decisions and solutions that were made to satisfy this need.

## 4.1 Design and model

### 4.1.1 Use cases

The use case for this plugin is to highlight the important part of a build. Usually, there is a sequence of steps happening in a build, for example, syntax check, compiling, testing, and deploying. The idea is to have these steps marked somehow. Since there are many ways to mark these steps across multiple platforms, the marks need to be customizable.

Regular expressions (regex) were chosen as the optimal way of defining what a mark is, because of their high expressiveness, general knowledge amongst target users, and supported development tools.

### 4.1.2 Functional requirements

The target of the plugin is to implement these key functionalities:

- allow the user to define custom log marks
- find given marks in the main log of a build and visualize them on the Build page and on the Project page
- make marks linkable to the main log
- generate a picture with a visualization of the marks
- compare the marks to previous build to see the difference

Furthermore, the marks should have following options:

- option Simple: this option is meant for simple one line marks defined by a single regex, matching one line

- option Advanced: this option is meant for multiline marks with varying content, defined by two regexes, one for the beginning mark and one for the end mark, matching the lines between them

## 4.2 Plugin steps

This section should provide an abstract overview on the main processes and functionalities of the plugin and should serve as a basic user guide.

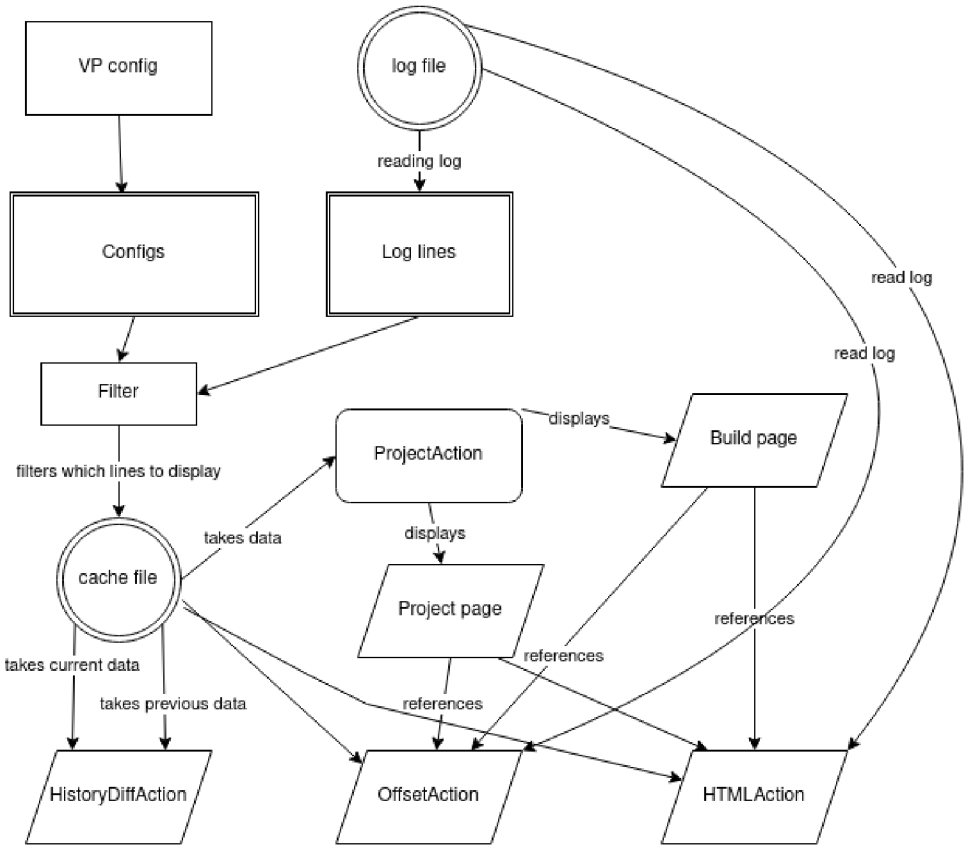


**Figure 4.1:** Diagram of plugin actions

### 4.2.1 Configuration

The first step of using the plugin is to define "marks" which should be printed out to the main log. At this point, marks are just standard text. They can be printed as a part of other build steps or added manually as steps that print into the main log.

Either way, it is up to the user to define the marks in the project configuration section.

Two options for defining a mark are available depending on the complexity of a given mark. The first one, Simple, is searching for a mark on one line. Regex input does not have to be complex, if your mark is the word "START", the configured regex can be just that.

The second one, Advanced, has the same structure, but it has an end regex, for example, the word "END". It comes with more options, an upper limit for finding the end mark, and index choosing which line to display in later summaries(as the whole Advanced mark is displayed as one line only).

Both complexities have an option not to display the mark, if you wish not to pollute your result log. The marks in the result log will not be shown, but remark that this change does not affect the original log created by Jenkins.

It is possible to create any number of marks, but each line can be matched by just one.

### 4.2.2 During building

By building on a configured project, internal processes of the plugin are triggered.

After all the steps are completed in a build, the following steps are performed:

1. Reading the main log of the build and loading the project plugin configuration.

2. Searching and filtering the marks and saving them into a cache file in the appropriate builds folder of Jenkins.

3. Reading the cache file and displaying the results on Project page and Build page.

4. Displaying the results as a part of HTML Results page, responding to request on Offset page, and comparing history on the History Difference page.

### 4.2.3 After building

The results of the build can be viewed on the corresponding Build page. In addition, the results of the latest build are shown on the Project page. The result lines are linked to their place in the HTML log, so a specific line will bring you to the specific line in the HTML log, along the lines not matched by marks. Furthermore, the full marks, even with the contents of an Advanced mark, are colored accordingly.

This should provide you with a good overview of what happened in the build, if something went wrong and where it went wrong.

### 4.2.4 Multiple builds

After performing multiple builds with a defined configuration, the differences in relation to the last build can be viewed on the History Diff Action page. If marks in the summary do not look according to expectation, you can look here for details, since the possible issues, such as not finding the end mark, are indicated there.

Furthermore, if you are dealing with large log files which might take some time to load in HTML, you can access the logs from the Offset Action page, as reading the log file this way jumps right to the part of the given offset, which significantly improves performance since you do not have to read the whole log file.

## 4.3 Implementation dilemmas and problems

While implementing, some problems arose and had to be dealt with.

Some were purely programming challenges of a given task, some were given by the provided Jenkins interfaces and limitations, and others were purely a matter of user experience. This section talks about some of the problems I encountered during development.

### 4.3.1 How to display logs in Jenkins interface

As mentioned previously [1.3.3], there is not much support for displaying logs. Therefore the main functionality must be created basically from scratch. However, Jenkins supports creating additional pages under the directory tree of each function and also supports accessing the log file, either via methods defined in Jenkins library (getLogs, getReader) or by generally accessing the "log" file.

As the use case is to have somehow modified logs visible, thus we created a webpage using a combination of Jenkins markup, HTML, CSS, and JavaScript. This will allow us to fully control the output according to the user's needs, for example, changing the visuals (highlighting, fonts, etc) or possibly adding or removing lines.

Alternatively, to display parts of larger log files, another page is dedicated to inputting the offset range and sending the request, resulting in a plaintext response. Differentiating between these two ways means the HTML page with full logs is not rendered until accessed, preventing substantial loading times in the case of large log files.

### 4.3.2 Adressing lines on another page

One of the key functionalities that can improve developer time spent is linking between the mark displayed in summary and the exact position in the log.

For the HTML logs page, we can take advantage of link anchors. By wrapping every line by anchor with the name corresponding to the number of the line, we can address the anchor element on another page. This allows the user to jump from any summary directly to the exact place of the given mark in the HTML logs page.

This approach is not possible for the Offset page because we need to transfer numerical arguments across pages, not the position of the element on the page. Fortunately, this can be done by putting these arguments to the URL.

This means that we can add these arguments to the offset anchor link, which then leads to the same Offset page, but the arguments in the URL are different. These arguments can then be extracted using a simple JavaScript and prefill them in the corresponding inputs for maximal user-friendliness.

```
<script>
    const urlParameters = new URLSearchParams(window.location.search);
    const fromParameter = urlParameters.get('from');
    const toParameter = urlParameters.get('to');
    console.log(toParameter)
    const elemFrom = document.getElementsByName('from')[0];
    const elemTo = document.getElementsByName('to')[0];
    elemFrom.value = fromParameter;
    if (!elemFrom.value.empty()){
        elemTo.value = Number(fromParameter) + 2000;
    }

</script>
```

**Figure 4.2:** Script for prefilling the Offset action parameters

### 4.3.3 Displaying image on Build page and Project Page

In the early stages of development, I was working with the possibility of displaying the summary in the form of a PNG image or JavaScript image.

In theory, doing so would allow me to use more tools to create better visuals. However, working with images has proven to be significantly more difficult. Converting text to image and then retrieving the information back is not practical, as Jelly provides many more options of displaying text, including hyperlinks.

Creating useful hyperlinks in images would require JavaScript, which would add more complexity to the page, that is why the Jelly script was chosen as the better option.

Furthermore, to display the image, we would need to use <img> tag from HTML with reference to the static image, contrary to text from cache file, which can even be modified by Java methods according to needs. However, the summary image should be generated dynamically, for each Build. To reference the image, we would need to save it to the directory tree of a Jenkins running instance, which can be achieved by placing it as a static image in the "webapp/img" folder.

But this approach will not work for us, as said previously, our image needs to be generated dynamically.

Another option is to generate the image in the workspace of a build. This option is partly fitting, however since workspaces can be

overwritten or wiped, the functionality of looking at history would be rather hard to do.

After encountering these problems, I chose to pick strings in HTML as the way to represent the summary. However the generated image is still part of the plugin. As an optional feature, the image can be generated as an artifact for each build, which means it is a product of each build and can be kept for each build.

## 4.4 Implementation

### 4.4.1 Plugin structure

The project structure tree logically divides files that serve similar purposes together. The core of the implementation is located in the "main" folder [4.3].

The "java" folder contains the folder with name consisting of groupId and artifactId, in our case "io.jenkins.plugins.VirtualPipeline", which contains the classes listed in the Classes section.

The "resources" folder holds jelly scripts. These scripts are named according to their purpose, for example config.jelly is used for configuration of a class. In addition, they are also located in folders named exactly as the Java classes they belong to. This is important, as having a different name of a script path would result in the script not being assigned to the class, thus not working properly.

The "webapp" contains static content that is exported to the running Jenkins instance directory, used for icons for example.

Outside of the main structure is "pom.xml", a project dependencies and configuration file, aswell as "Jenkinsfile", which is configuration for the Jenkins pipeline. Lastly, the project also contains license and readme files.

```
virtual-pipeline
├── src
│   └── main
│       ├── java
│       │   └── io.jenkins.plugins.VirtualPipeline
│       │       ├── VirtualPipelineProjectAction.java
│       │       └── other Java classes
│       ├── resources
│       │   ├── io.jenkins.plugins.VirtualPipeline
│       │   │   └── VirtualPipelineProjectAction
│       │   │       ├── summary.jelly
│       │   │       └── other jelly scripts
│       │   └── index.jelly
│       └── webapp
├── pom.xml
├── LICENSE.md
├── README.md
└── Jenkinsfile
```

**Figure 4.3:** Project structure

### 4.4.2 Classes

The following section summarizes the main purpose and function of each class.

#### VirtualPipelinePublisher

This is the "root" class of the plugin, extending the Recorder class as described earlier [2.1.2]. It has input configurations as a part of the constructor with the "@DataBoundConstructor" to bind the data

from the form using Stapler[1]. The class also includes its Descriptor, DescriptorImpl, which is necessary to provide info about the class as a part of the Extension. The root method of the plugin is the overridden method "perform", which includes all of the code we wish to run as a part of the plugin (not including supportive methods, such as naming the plugin in Descriptor).

### VirtualPipelineProjectAction

Class is used mainly for displaying the results on the Build page (bound to summary.jelly), Project page (bound to floatingBox.jelly), and displaying other actions on the sidebar of a project (bound to index.jelly). The side purpose of this action is to perform the comparison to the previous build and displaying the build. The reason why this function is located here is that both display summaries are defined here, happening after reading from cache.

### VirtualPipelineInput

Abstract class defined to be a encapsulation of the two types of input, Simple and Advanced.

### VirtualPipelineInputDescriptor

Empty Descriptor class as all important content is defined in the children classes.

### VirtualPipelineInputSimple

Represents Simple mark that can be defined in the project configuration. The purpose is catching one-line mark defined by attribute "regex". Optionally the mark matched by the regex could be deleted in the result by setting the "deleteMark" to true. The class also includes its DescriptorImpl which includes the form validation for this class, with the form looks defined in config.jelly.

1. library for mapping classes to pages in URL tree structure: `https://github.com/jenkinsci/stapler`

### VirtualPipelineInputAdvanced

Represents Advanced mark that can be defined in the projects configuration. Allows to match multiple line mark with regex matching the start line (startMark) and regex matching the end line (endMark).

In case of error, we do not want to corrupt the whole result by an unmatched endMark. That is the purpose of maxContentLength is to have an upper cap of lines to match. When matching, reaching this lines cap will act as if the endMark was found, with the exception of being marked as LIMIT_REACHED_LINE LineType. Attribute numberOfLineToDisplay assigns a specific line which should be displayed in summaries due to advanced mark having multiple lines. The mark can be configured as an index staring at zero representing the startMark.

Other future possibility would be to configure this line by creating another regex attribute.

Similarly to the VirtualPipelineInputSimple, the mark can be deleted from the resulting logs setting the deleteMark attribute to true. Class also includes its DescriptorImpl with the FormValidationfor the form defined in config.jelly.

### VirtualPipelineLineOutput

Class represents the the format of cache saved in the cache files. The objects of this class are directly serialized and stored in the cache file. When needed afterwards, read and transformed back to an Java Object. Thus the attributes of this class should include all the necessary information about matched lines because when comparing and displaying the results, this cache is the only source of information we get about the matched lines. Thus the class attributes:

**regex** the regex under which the line was matched

**line** String value of line itself with applied deletion of the mark if deleteMark was true

**index** index number of the line with relation to the full log

**deleteMark** true if mark was deleted, false otherwise

**type** the role of the line, more described in LineType

**display** true if line was chosen to be displayed, false otherwise

**lineStartOffset** file offset corresponding to the start of line of the "log" file in build folder, necessary for VirtualPipelineOffsetAction

#### VirtualPipelineOutputHistoryMarked

Class extension of VirtualPipelineInput for representing the status of line when comparing to previous builds. The only addition is the historyType attribute described further in HistoryType.

#### VirtualPipelineFilter

Class for manipulating and filtering lines. The most important method "filter" is taking lines (from log, with added offset information - more in LineOffset) and given configurations (those created by user) and applies them together.

The result is a List of matched lines with the proper information in VirtualPipelineLineOutput. Furthermore, the result can be directly saved to a file in JSON format via the supportive method.

If deleteMark option is selected, deletion of the mark is performed by method removeRegexMark, simply by replacing the matched regex with empty string.

#### VirtualPipelineHTMLAction

Class represents one of the Actions displayed in the sidebar and implementing SimpleBuildStep.LastBuildAction. The actions build up a new page with logs of the latest build, including non-marked logs.

However, the logs are enhanced by color representation of marks found. In addition to color, each line is referencable via HTML anchor, thus making it easy to jump to exact wanted line from the summary from Build page or Project page.

Furthermore, class contains multiple method for working with lines from cache. Significant part of the action effect is in the corresponding index.jelly file, as it defines the looks of the resulting HTML page rendered.

### VirtualPipelineOffsetAction

This class implements the Action displayed in the sidebar, also implementing SimpleBuildStep.LastBuildAction. The action takes parameters "from" and "to" representing offset in the log file as an input of form. Submitting the form sends a Stapler GET request to the backend, where, if successful, lines corresponding to the offset range are sent back as plaintext.

### VirtualPipelineHistoryDiffAction

This class implements a page that displays the difference between matched marks from last build and the previous build. The previous build can be chosen as the immediate previous build or the latest stable build, depending on the configuration chosen by the user.

In the earlier versions of the plugin, there was an initiative to implement own algorithm for displaying differences between builds.

However, using the already existing advanced library's Java Diff Utils [2] functions was deemed better.

### VirtualPipelinePictureMaker

Class with the sole purpose of fulfilling the functional requirements to generate image of found marks in png format. Class methods are customizable enough for our purposes.

### LineType

Enum class with possible values describing the line's relation to marks:

**ONE_LINE** for lines matched by Simple input

**START_MARK** for lines matched by start mark of Advanced input

**CONTENT_LINE** for lines that are between start and end mark of advanced mark

**END_MARK** for lines matched by end mark of Advanced input

2. `https://github.com/java-diff-utils/java-diff-utils`

**LIMIT_REACHED_LINE** for lines that reached the maxContentLength of Advanced input

**DEFAULT** for lines that were not matched

#### LineWithOffset

Class for storing lines from files with the given line start offset. Since reading the lines and keeping the count of offset may be unclear, working with this class is smoother.

#### HistoryType

This class implements a page that displays the difference between matched marks from last build and the previous build. The previous build can be chosen as the immediate previous build or the latest stable build, depending on the configuration chosen by the user.

**SAME** both lines are same, no change

**DIFFERENT_PREVIOUS** line differs and is from previous build

**DIFFERENT_CURRENT** line differs and is from current build

**DEFAULT** used when comparing is not performed

In the later versions of the plugin, we use a library to compare differences between current and later build, so this format is deprecated.

### 4.4.3 Cache JSON format

The JSON cache file is converted from a type List< VirtualPipelineLineOutput >. Each item represents one line along with the metadata describing how it was matched.

This makes comparing to any previous build possible just by reading the cache file of the previous build, not by searching all the full logs of a build each time we want to compare.

### 4.4.4 Form input

The user can define the marks in the Configure section. To ensure avoid typos and incomplete configurations, some form validations were implemented. Jenkins has support for form validation, meaning it is quite developer friendly to implement these restrictions.

For regex input, form validation checks whether the regex string is empty and whether the regex string can be compiled. Similarly, we check for number values and if the user input makes sense, for example, no negative numbers are inputted.

```
@Extension
public static final class DescriptorImpl extends VirtualPipelineInputDescriptor {

    no usages   Tomas Havlicek *
    public FormValidation doCheckRegex(@QueryParameter String regex) {
        if (regex.isEmpty()) {
            return FormValidation.error("Regex is empty");
        }
        try {
            Pattern.compile(regex);
        } catch (PatternSyntaxException exception) {
            return FormValidation.error("Regex is invalid");
        }
        return FormValidation.ok();
    }
```

**Figure 4.4:** Form validation

### 4.4.5 Jelly scripts

To use display the contents of our plugin, we use Jelly scripts [2.2.5]. Each script uses methods from the bound class, mainly doing the heavy lifting of opening and filtering log file in the background. The resulting lines are handed to a Jelly script, which can format them in any way needed. The script can also access all of the class properties of the given lines, which means we can display different types of lines in different colors or style (using CSS in “style” tag).

### 4.4.6 Used libraries

The plugin [4.5] uses the parent POM[3] of a plugin, which allows it to function as a plugin and also the necessary dependencies from Jenkins, which allows us to work with classes defined in Jenkins, such as AbstractBuild or AbstractProject.

In addition, we are using the Jackson library [4]from FasterXML for serializing Objects to JSON and back, which is used when creating the cache file.

As the best way to compare differences between builds, we use a Java Diff Utils library[5]. It has multiple settings for displaying the differences so that it can be adjusted according to user feedback.

```
<modelVersion>4.0.0</modelVersion>
<parent>
    <groupId>org.jenkins-ci.plugins</groupId>
    <artifactId>plugin</artifactId>
    <version>4.52</version>
    <relativePath />
</parent>
<groupId>io.jenkins.plugins</groupId>
<artifactId>virtual-pipeline</artifactId>
<version>${revision}${changelist}</version>
<packaging>hpi</packaging>
<name>Virtual Pipeline</name>
<url>https://github.com/jenkinsci/${project.artifactId}-plugin</url>
<licenses>
    <license>
        <name>MIT License</name>
        <url>https://opensource.org/licenses/MIT</url>
    </license>
</licenses>
```

**Figure 4.5:** Part of plugin POM

3. https://github.com/jenkinsci/plugin-pom
4. https://github.com/FasterXML/jackson
5. https://github.com/java-diff-utils/java-diff-utils

# 5 Conclusion

The goal of this thesis was to create a working plugin that takes user-defined marks, finds these marks in the main log, and provides a friendly user interface to access and jump places in the log specified by the marks. It has the ability to generate png images, compare with previous builds and most importantly to display a link on the project and build, which allows the user to jump directly to a place in the log. Alternatively, it has a working offset log feature, which allows the user to read even big logs.

The plugin satisfies the functional requirements, however since it is a user-experience oriented plugin, there is room for improvement in the way how the users would like to use and customize the plugin. For example, as there is a lot visual settings involved, some of it may be customized in the step settings instead of having one general visuals. The same applies for specific elements, such as differences between builds, which can be displayed in multiple ways according to the user use case.

As of now, the necessary steps for the plugin to be accepted and hosted on the jenksci repository are being undertaken. The users using the plugin have smoother access to the Jenkins log and are provided with better workflow compared to the manual work. The plugin is publicly available on a Github repository[1].

1. `https://github.com/thavlice/jenkins-report-virtual-pipeline`

# Bibliography


1. *What is Jenkins?* [online]. Jenkins Infrastructure team [visited on 2023-04-24]. Available from: https://www.jenkins.io/doc/.
2. SMART, John Ferguson. *Jenkins: The definitive guide: Continuous integration for the masses*. 1st ed. Cambridge (Mass): O'Reilly Media, 2011. ISBN 9781449305352.
3. MONTE, Christian Del. *Hudson is Retiring. The End of a Jenkins' Story.* [online]. The Startup, 2020-06-26 [visited on 2023-04-24]. Available from: https://medium.com/swlh/hudson-is-retiring-the-end-of-a-jenkins-story-9cee866135ab.
4. *About Jenkins* [online]. Jenkins Infrastructure team [visited on 2023-04-24]. Available from: https://www.jenkins.io/press/.
5. *What is Jenkins Pipeline?* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/book/pipeline/#overview.
6. *Contribute code* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/participate/code/.
7. *License* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/developer/publishing/preparation/#license.
8. *Open Hosting request* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/developer/publishing/requesting-hosting/.
9. *Extension Points defined in Jenkins Core* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/developer/extensions/jenkins-core/.
10. *Jenkins : Defining a new extension point* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://wiki.jenkins-ci.org/display/JENKINS/Defining+a+new+extension+point.

11. *Hudson* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://javadoc.jenkins.io/hudson/tasks/package-summary.html.

12. *Preparing for Plugin Development* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/developer/tutorial/prepare/.

13. *Maven HPI Plugin* [online]. Github, Inc. [visited on 2023-04-25]. Available from: https://github.com/jenkinsci/maven-hpi-plugin.

14. *Jenkins Development Support* [online]. JetBrains s.r.o. [visited on 2023-04-25]. Available from: https://plugins.jetbrains.com/plugin/1885-jenkins-development-support/.

15. *netbeans-stapler-plugin* [online]. Github, Inc. [visited on 2023-04-25]. Available from: https://github.com/stapler/netbeans-stapler-plugin.

16. *Installing Jenkins* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/book/installing/.

17. *Create a Plugin* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/developer/tutorial/create/.

18. *Build and Run the Plugin* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/developer/tutorial/run/.

19. *Jelly : Executable XML* [online]. The Apache Software Foundation, 2002/2010 [visited on 2023-04-25]. Available from: https://commons.apache.org/proper/commons-jelly/.

20. *Jenkins Taglib Documentation* [online]. Jenkins Infrastructure team, 2004/2023 [visited on 2023-04-25]. Available from: https://reports.jenkins.io/core-taglib/jelly-taglib-ref.html.

21. *Blue Ocean status* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://www.jenkins.io/doc/book/blueocean/.

22. *Pipeline: Stage View* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://plugins.jenkins.io/pipeline-stage-view/.

23. *Pipeline Graph View* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://plugins.jenkins.io/pipeline-graph-view/.

24. *TeamCity* [online]. JetBrains s.r.o., 2000/2023 [visited on 2023-04-25]. Available from: https://www.jetbrains.com/teamcity/.

25. *Log Parser* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://plugins.jenkins.io/log-parser/.

26. *Jenkins Log File Filter Plugin* [online]. Jenkins Infrastructure team [visited on 2023-04-25]. Available from: https://plugins.jenkins.io/log-file-filter/.