BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATIC COMPONENT METADATA EXTRACTOR AND CONSOLIDATOR FOR CONTINUOUS INTEGRATION

AUTOMATICKÝ NÁSTROJ K ZÍSKÁVÁNÍ METADAT KOMPONENT PRO ÚLOHY PRŮBĚŽNÉ INTEGRACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. JIŘÍ KULDA**
AUTOR PRÁCE

SUPERVISOR — **Prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Master's Thesis Specification/19997/2016/xkulda00

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems — Academic year 2016/2017

Master's Thesis Specification

For:	**Kulda Jiří, Bc.**
Branch of study:	Intelligent Systems
Title:	**Automatic Component Metadata Extractor and Consolidator for Continuous Integration**
Category:	Software analysis and testing

Instructions for project work:

1. Get familiar with the area of automated continuous integration (CI) testing using Jenkins.
2. Analyze the ways in which different quality engineering (QE) teams in Red Hat are implementing CI testing with a stress on what CI test metadata they store.
3. Propose a unified format for storing CI test metadata.
4. Design and implement an interface for CI metadata querying and collection.
5. Test the developed tool in cooperation with selected Red Hat QE and development teams on a selected component and collect their feedback by a questionnaire.
6. Summarize and discuss the obtained results as well as their possible future improvements.

Basic references:

- Duvall, P., Matyas, S.M., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk, Pearson Education, 2008.
- Vogel, L.: Continuous integration with Jenkins - Tutorial, dostupné online na: http://www.vogella.com/tutorials/Jenkins/article.html.
- Closs, S., Studer, R., Garoufallou, E., Sicilia M-A.: Metadata and Semantics Research, Springer, 2014.

Requirements for the semestral defense:

The first two items plus at least a proposal of the unification from the third point.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**, DITS FIT BUT
Beginning of work:	November 1, 2016
Date of delivery:	May 24, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

Petr Hanáček
Associate Professor and Head of Department

Abstract

This master thesis focuses on the modification of continuous integration practice within the *Platform* team at Red Hat. The result of this thesis is the *Metamorph*, tool which will make it possible to unify the continuous integration tools of sub teams under the *Platform* team. The theoretical part describes the creation of a continuous integration practice and explains its benefits. Subsequently, existing CI tools (in the industry) are presented in detail. The following section demonstrates how continuous integration uses the *Jenkins* tool. This master thesis also contains the particulars of existing internal CI solutions at Red Hat. In the practical part, the design and implementation of tool that was made during the creation of this master thesis are introduced. In conclusion, the results are tested by one team at Red Hat and a possible extension is outlined.

Abstrakt

Tato diplomová práce popisuje úpravu průběžné integrace pro *Platform* tým ve společnosti Red Hat. Výsledkem práce je nástroj *Metamorph*, který umožní sjednocení ostatních nástrojů průběžné integrace pod týmem *Platform*. Teoretická část popisuje vznik, popis a přidané hodnoty průběžné integrace. Následně jsou blíže přiblíženy existující nástroje na trhu. Dále je zde popsáno použití průběžné integrace v nástroji *Jenkins*. V práci jsou také dopodrobna popsány existující řešení průběžné integrace ve společnosti Red Hat. Dále je zde popsán návrh a implementace výše zmíněného nástroje. V závěru jsou výsledky práce otestovány týmem z firmy Red Hat a nastíněny možnosti rozšíření.

Keywords

Continuous Integration, Jenkins, Red Hat, Metamorph, build, Test Data Management, integration, Ansible

Klíčová slova

Průběžná integrace, Jenkins, Red Hat, Metamorph, sestavení, Test Data Management, integrace, Ansible

Reference

KULDA, Jiří. *Automatic Component Metadata Extractor and Consolidator for Continuous Integration*. Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. Ing. Tomáš Vojnar, Ph.D.

Automatic Component Metadata Extractor and Consolidator for Continuous Integration

Declaration

Hereby I declare that this Master's thesis was prepared as an original author's work under the supervision of Mr. Prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by Mr. Gowrishankar Rajaiyan. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Jiří Kulda
May 22, 2017

Acknowledgements

I would like to thank Mr. Gowrishankar Rajaiyan for his technical leading of this Master Thesis. At the same time, I would like to thank Mr. Prof. Ing. Tomáš Vojnar, Ph.D., for his pedagogical leadership. Last but not least, I would like to thank Ing. Jiří Čanderle for pointing me to the correct people.

Contents

Chapter 1

Introduction

The aim of this work is to solve a problem which arose at Red Hat. Nowadays, there are at least twelve different continuous integration tools under the *Platform CI* team in this company with one CI server per every team. The problem is how to unify all the existing CI solutions into one. Resolving this would save Red Hat a lot of money and developers' work, as many CI features are developed separately, such as *covscan* or *rpmdiff*. The solution of this problem could be used by multiple teams. Integrating Red Hat continuous integrations under one project would eliminate duplicated effort so often found in software development.

The purpose of this work is to create a tool which would be the basic building block for solving the above-mentioned problem. The first tool is a metadata extractor. It will eliminate the need for every CI solution to depend on concrete metadata and their storage system. Teams are now focusing on collecting metadata, but they should mainly focus on how to act on the existing ones.

The rest of this thesis is structured as follows. The second chapter provides continuous integration introduction information. It explains what CI is, what benefits it has, why your team should or should not use it, and what the existing CI tools solutions on the market are.

Continuous integration implementations at Red Hat can be found in Chapter 3. CI solutions from different teams are described there, with the focus on their implemented features. Every Jenkins job functionality is explained as well. The documentation of these solutions is, in fact, the first contribution of this thesis. The last section of the chapter introduces the *Freestyle* build job options, which are the key of *Jenkins* build job and a small code example of them in the *YAML* format using the *Jenkins Job Builder*.

Chapter 4 focuses on metadata analysis used by CI teams. At first, it presents a research on existing external solutions for metadata storage. It also contains information about Red Hat teams' internal metadata storage systems. At the end, there is a proposal for solving the CI implementations unification problem.

Chapter 5 describes the *Metamorph* tool and its plugin design, supported with tables and figures. The tooling implementation can be found here. *Ansible* language is described in the implementation as well because of some parts written in this language. The last part of the chapter focuses on testing, which presents feedback on the proposed solution gathered from various Red Hat teams.

The last Chapter contains an overall summary of the created solution for CI unification problem with drafted out future improvements of the implemented tool.

Chapter 2

Continuous Integration

The first time the term *continuous integration* was used is found in the 1994 book by Grady Bosh, called Object-Oriented Analysis and Design with Applications [3]. He explains that developing software with the micro-processes functions can be used as an internal type of continuous integration effort.

A notable step in the evolution of CI occurred within the Extreme Programming methodology invented by Kent Beck and Ron Jeffries in 1997, where continuous integration serves as one of the procedure's pillars. Extreme programming can be described as a pragmatic approach to program development that emphasizes business results first and takes an incremental, get-something-started approach to build the product, using continuous testing and revision. Kent Beck also says that when writing code, you should write tests first so you know which requirements it will have to pass in order to be labeled as successful.

In year 2000 Martin Fowler described continuous integration as a software development practice where members of a team integrate their work frequently, usually once a day per person, resulting in multiple integrations per day. Each integration is verified by an automated build (including testing) to detect integration errors as quickly as possible [6].

2.1 What is Continuous Integration

The most accurate definition of continuous integration was coined by Martin Fowler in [6]. He implies that there is a greater need to integrate software parts and ensure that software components work together early and often. Many teams wait with integration until the end of the whole project, but that can lead into a great number of software quality problems which are costly and usually dramatically delay the project.

As is explained in article [12], continuous integration organizes development into functional user stories. We can imagine them as smaller chunks of work similar to sprints. The author of this article also argues that test-driven development is a part of architecture-based approach, which extends basic agile practices enough to provide both high quality and project flexibility.

Continuous integration also focuses on getting test results as soon as possible. Normally, there is a group of developers and another group of quality engineers who are responsible for software correctness and quality. In smaller projects, when quality engineers detect a new version of software, they run tests for it and create more tests. This time between code change and error detection by QE is acceptable in smaller projects, but this approach does not scale into more complex scenarios. Developers need to have results as soon as possible so they can fix the detected issue in time. Continuous integration can be used to reduce the time between defect detection and its correction, thus improving overall software quality.

Every time „continuous“ integration word is used it can be understood as something which starts once and never stops. This brings up a though that the process is constantly integrating, which is not a CI environment behavior. Continuous integration shall be described rather as „continual integration“ [14].

Almost every continuous system contains the steps below [14]:

- After a change in code, a developer first pushes the code to the source control server. In between, the CI server on the integration build machine is polling project's source code repository changes from the source control server.
- When a commit arrives to the source control server, it is detected by the CI server. Afterwards, the latest version of the project's repository is polled and then the CI server executes a build script which integrates the software.
- The CI system is generating feedback by sending the results to eligible recipients.
- The continuous integration system continues to poll for changes in the source control repository.

These steps can be seen in Figure 2.1.

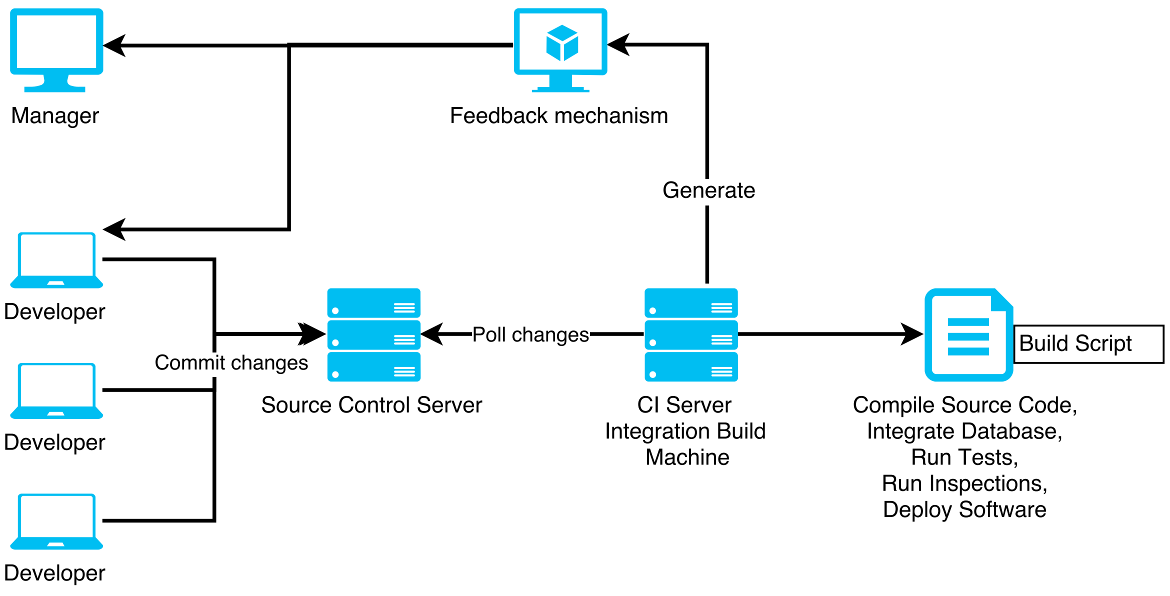


Figure 2.1: A CI system and it's components [14].

2.2 What is the value of Continuous Integration?

The main benefits of continuous integration are an acceleration of development, faster delivery of new versions, and a reduction of errors. All these are done due to the automation of as many development parts as possible.

It is proven [11] that when people have to be repetitive and perform the same task every day, they leads to make more and more mistakes. Moreover, their motivation drops as well. These problems can be mainly solved by introducing the continuous integration system.

Continuous integration prevents the creation of human errors during testing. As we know, the programmers make mistakes and every tool or procedure which prevents them from doing so improves software quality.

At the high level, the value of CI is to [14]:

- **Reduce risks**. Project risks can be reduced by integrating many times a day. As long as there are metrics in place to measure the health of the application, code defects can be detected sooner and fixed quicker. With this early detection, we are also able to fix problems faster, because the developer who made this change still knows what he did and probably how to fix it as soon as possible. The risks that can be reduced are late defect discovery, low-quality software and the lack of project visibility.

- **Reduce repetitive processes**. By reducing repetitive processes we can save time, costs and effort. These repetitive processes occur across all project activities such as building, testing, feedback, and many more. Reducing repetitive manual processes assures that [18]:

  - An ordered process is followed.
  - The process runs the same way every time.
  - The process will run every time a code is committed to the source control.
  - There is the capability to overcome resistance to implement improvements by using automated mechanisms.

- **Generate deployable software**. For clients and users, this is one of the biggest assets of the continuous integration approach. In the previous sections, we mainly speak about improving software quality and reducing risks, but the biggest value is to have deployable software. By using CI, team members are informed in case of any problem and are motivated to fix it as soon as possible. After the fix, we have deployable software again. In case of waiting with integration until the end of the project, it could occur that the software would be unable to integrate and the whole project would be delayed. By fixing bugs in a hurry, new defects could be created and the credibility of the project would go down.

- **Enable better project visibility**. Making effective decisions and implementing new improvements without real or recent data to support them is fairly difficult. Continuous integration systems can provide just-in-time information about the recent build status and quality metrics. The system is able to notice trends in success or failure build, because of frequent integration. A CI system is therefore efficient at providing regular data about software quality, which can be used to support decisions and project management.

- **Establish greater product confidence**. Continuous integration provides clear information, which component build is tested or not. With a CI system, we know that the result is a functionally testable product. One of the advantages of CI system is having fast feedback. That brings a big confidence to developers and other team members in pushing parts of code because in case of any problem they will have the feedback as soon as possible.

2.3 Why Teams avoid Using Continuous Integration?

As was described in Section 2.2, continuous integration systems have many benefits, yet many teams still avoid them. What is the reason? It is often a combination of concerns such as [14]:

- **Increased overhead in maintaining the CI system**. Many teams are discouraged by believing that the continuous integration system maintenance will be too difficult. When teams want to avoid using CI, they need to manage manual processes instead. Therefore they have two options only. Manage the CI system or be controlled by the manual processes. Working on a continuous integration system maintenance is worth trying because of the benefits which it brings. CI is the most suitable solution for multiplatform projects, yet some teams often resist to use it.
- **Too many changes**. A continuous integration system may feel big and robust, which can make teams think that too much implementation would need to be done for them to switch their project to CI. At the beginning when nobody is used to the new system yet, it is better to start slowly adding tests and test builds on daily basis. Once everyone becomes accustomed to the new practice, the build frequency can be raised.
- **Too many failed builds**. This mainly occurs when a developer does not run local tests and builds before committing the code to source control repository. The problem can be caused by anything, but if the project is using a CI system, the developer will be notified about the broken build as soon as possible.
- **Additional hardware/software costs**. A continuous integration system usually requires an extra machine on which it will be launched. It is a nominal expense, but when it is compared to the cost of searching a problem later in the development lifecycle, the acquisition is well worth it.
- **Developers should be performing these activities anyway**. Some developer activities are similar to the ones which are performed under a continuous integration system. Therefore many managers thus mistakenly assume that there is no need for CI since it should be done by developers. That is not true. Developers tries to perform these activities more effectively. On the other hand, CI system ensures that these activities are performed in a clean environment, after a commit and with fast feedback.

2.4 Existing Continuous Integration Solutions

Nowadays there are many continuous integration tools on the market, some of them are widely used. Every project which chooses to use a CI system faces the task of selecting the right continuous integration tool. That is even more true for first timers. The simplest way to create a CI system is to use a script, but this approach is not very recommended and can only be used in special cases.

In the sections below, four continuous integration tools are described. Figure 2.2 shows interest in each of the chosen CI tools. As can be seen, the most searched tool is *Jenkins*. Reasons for that are described below. Simultaneously, a discussion about their pros, cons or their features is presented

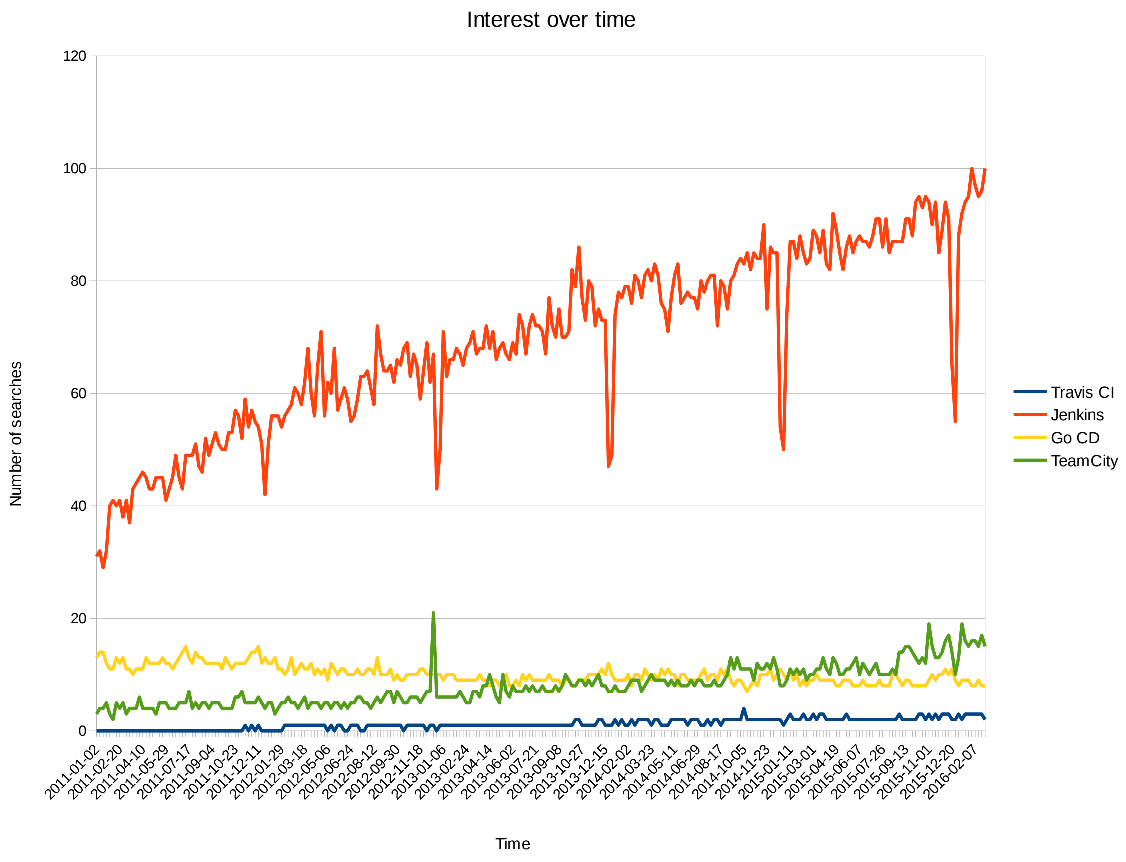


Figure 2.2: Interest in the chosen CI systems on the web [14].

2.4.1 Jenkins

Jenkins is the leading open source continuous integration tool. *Jenkins* is written in the *Java* language and it originated as a part of *Hudson* when *Oracle* bought *Sun Microsystems*. It is a server-based system running in a servlet container, such as *Apache Tomcat*. It supports *SCM* tools including *AccuRev*, *CVS*, *Subversion*, *Git*, *Mercurial*, and many more. It can also execute *Apache Ant* and *Apache Maven* based projects. *Jenkins* is published under the MIT license, so it is free to use and distribute. Its main features are [9]:

- **Easy installation**. A simple execution of `java -jar jenkins.war` is sufficient. No additional install action, no database needed. It is possible to use an installer or a native package as well.
- **Easy configuration**. Mainly by friendly web GUI.
- **Rich plugin ecosystem**. Jenkins integrates virtually every build tool or *SCM* which exists.
- **Extensibility**. It is easy to create new *Jenkins* plugins, because most *Jenkins* parts can be extended or modified. This provides rich possibilities to customize *Jenkins* exactly to fit every team needs.

- **Distributed builds**. No problem with building on various operating systems.

One of *Jenkins'* known disadvantages is its web GUI because its transparency for beginner users. Despite this perceived flaw, *Jenkins* creators present it as one of its features. The web GUI quality was discussed with at least 10 developers and 5 quality engineers at Red Hat and their feedback was collected by a questionnaire. The results can be seen in Figure 2.3. The Figure shows that a majority of respondents replied that it is not friendly, which supports information founded on *Jenkins* forums. Part of the respondents were members of a CI team who works with *Jenkins* on daily basis and they replied that it is friendly. Thus, *Jenkins* web GUI is not good for people who work with Jenkins occasionally.

Jenkins is the main continuous integration solution used in Red Hat and therefore the thesis concentrate on it.

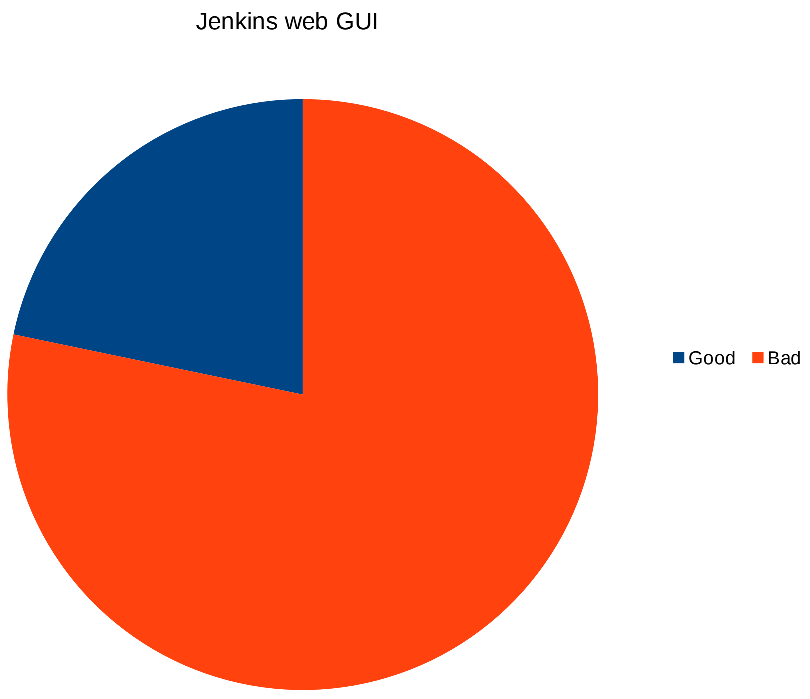


Figure 2.3: How friendly the Jenkins web GUI is.

The most remarkable feature of Jenkins is the number of plugins it offers. There are more than 1000 of them. This advantage really extends *Jenkins*' capabilities and provides extra plugin integrations for teams which are using *Jenkins* [17].

2.4.2 Travis CI

Travis is an open source service free for all open source projects hosted on *Github*[1]. It is configured using `.travis.yml`, which contain testing plan [2]. *Github* have big integration with *Travis CI* which provides automatic notification after a commit or a pull request

[1]https://github.com/

status. Compared to *Jenkins*, the *Travis CI* web GUI is very friendly, clean and easy to navigate.

The *Travis CI*'s main features are:

- Configuration file with code.
- Running tests in parallel.
- Support for Linux and Mac.
- Great API and command line tool.
- Clean virtual machine for every build.

Travis CI supports a wide range of programming languages. Thanks to *Travis* documentation[2], it is very easy to setup a CI system for a chosen programming language. Running more than one job concurrently can be done after acquiring monthly subscription plans.

According to Ben Dougherty's article [5], the biggest Travis CI disadvantage is the low variety of reporting given standard test output formats like *JUnit*.

2.4.3 Go CD

Go was created and then open sourced by the *ThoughtWorks* company. Excluding the commercial support that *ThoughtWorks* offers, *Go* is free of charge. As the previous continuous integration tools, *Go CD* is also supported on multiple platforms. *Go CD* is under the Apache license, so it is free with a paid support.

The *Go CD* main features are:

- **Promote trusted artifacts**. *Go CD* makes it easy to pass once-built binaries between stages. This leads to know exactly what's being deployed.
- **Deploy any version, any time**. It allows to deploy any known good version of application to wherever you like.
- **Eliminate bottlenecks**. By providing trivial parallel execution across pipelines, platform, versions, etc.
- **Plugins**. A big number of plugins already available. One can also write his own need be.
- **Keep configuration tidy**. The possibility to reuse pipeline configurations via *Go CD*'s template system.

CI pipeline is a deployment process break up into stages. Usually contain steps which provides binaries, testing, manual checks and deployment. The benefit of CI pipeline is that they can be done in parallel and provide increasing confidence [7].

The major change compared to *Jenkins* is the pipeline concept. *Jenkins* and *Go CD* pipelines are difficult to compare, because their differing concepts. On the other hand, *Jenkins* pipelines are somewhat simplistic [1]. *Go CD*'s pipelines are designed from scratch to eliminate build process bottlenecks with the parallel execution of tasks.

[2] https://docs.travis-ci.com/user/getting-started/

2.4.4 TeamCity

TeamCity is an adult continuous integration server, developed in the labs of the *JetBrains* company. The *JetBrains* company offers an extended family of integrated development environments for various programming languages such as Java, Python, PHP, C++ and many more. *TeamCity* is Java-based like *Jenkins*. The *TeamCity* server is the primary component, but the main way to administer its users, agents, or projects is via a browser-hosted interface.

The main *TeamCity* features are:

- **Start saving your time from day one**. *TeamCity* has essentials you need to get started in a matter of minutes on various platforms.
- **Extend as you go**. A large number of plugins to use with the possibility to create new ones.
- **All-around customer support**. Extensive customer support base with various channels, forums and comprehensive online documentation.
- **Rely on scalable architecture**. Start free with a small base. In case of power need, simply expand server capacities.
- **Integrate and deploy continuously**. The continuous integration server has encompassed all the features from a mature continuous deployment platform. Nowadays, it is a solution for both.

In comparison to the *Jenkins* web GUI, *TeamCity* is very clear and friendly. It provides important information for team members and for stakeholders. It displays build progress, drill down details, and history information on the projects and configurations. Another advantage is the *TeamCity* tray application, which notifies on events such as the statuses of recent builds. Notifications are stored in trays instead of emails [13].

TeamCity looks like a perfect CI solution, but it has some cons as well. If a team needs only to checkout, build a project, and afterwards see the status of a build, *TeamCity* is too complex, and they will have to pay for it. Moreover, if the team has 50+ members, they will not be able to acquire it — it will be too expensive. Many existing continuous integration tools are open-sourced; not *TeamCity.* If *JetBrains* (the company that makes TeamCity) decides to stop a team support, the team is on its own. No big forums and no tweakers [10].

2.5 Jenkins Build Job Setup

Continuous integration using *Jenkins* is managed by its jobs. This section introduces general information and configuration of these jobs with the provided example.

For every continuous integration server, build jobs are the construction cells. A build job's responsibility is to compile, test, deploy, or do anything else which is needed. The variety of build job forms is very wide.

Build jobs for software projects are usually sorted into logical sequences. For example, the first build job will run unit testing. If it passes, the next job is run. The next build job can execute long running integration tests, run collection of code quality metrics, or generate technical documentation. Once all the build jobs in a sequence pass, it is possible to bundle the project and deploy it to a test server.

Creating a new build job in *Jenkins* is very simple. One just selects "*New Item*" in the menu. Jenkins supports several build jobs which appear after selecting *New Item*. They are:

- **Freestyle software project**. A general build job, provides huge flexibility.
- **Maven project**. A build job specially adapted to Maven projects.
- **Monitor an external job**. A build job which monitors a non-interactive process.
- **Multiconfiguration job**. A build job able to run in different configurations.

The most usable and flexible build job is the *Freestyle software project*, which can fit to almost every project [16].

Every *Freestyle software project* build job has several options [16]. These options allow to set up every build job to match almost all needs of a project. The main important options are:

- **Parameters**[3]. This option which provides the ability to specify build job parameters. You can choose between many types, such as *bool* for boolean parameter, *choice* type, which is a single selection parameter, *validating-string* to validate given string parameter, and many more. Some parameter types need additional *Jenkins* plugins, therefore it is needed to check whether the *Jenkins* server supports them.
- **SCM**[4]. Allows s to specify the source code locations of various projects. It is also possible to pass an empty value to the *scm* value. This enables other jobs to override this value. *SCM* specifies repositories for git, mercurial, Visualworks Smalltalk Store repository, and many more. Same as the *Parameters* option, some of the *SCM* options need additional plugins to be installed.
- **Triggers**[5]. After configuring the above two options, it is important to set up the *Triggers* option to tell Jenkins when to kick off a build. There are three main ways how to start a build job. It can be by starting a build job after another finished job, launch a build job in periodical intervals, or poll *SCM* for changes [16].

  For all scheduling tasks, *Jenkins* uses a cron-style syntax, consisting of five fields separated by white space (MINUTE HOUR DOM MONTH DOW).
- **Builders**[6]. *Builders* option is the basic building block which lets *Jenkins* know how to build a project. In a *Freestyle* build, it is possible to have many build steps or none if its need to. You can execute an Ant, Maven and CMake targets, but there is a *shell* option for more specific builds. Some well known build tools can be integrated, or additional plugins can be installed [16].
- **Publishers**[7]. The last option in *Freestyle* build job is *Publishers*. This option close ups build job behaviour and its purpose is to report build results, archive some of generated artefacts or to notify people about the build results.

[3]http://docs.openstack.org/infra/jenkins-job-builder/parameters.html
[4]http://docs.openstack.org/infra/jenkins-job-builder/scm.html
[5]http://docs.openstack.org/infra/jenkins-job-builder/triggers.html
[6]http://docs.openstack.org/infra/jenkins-job-builder/builders.html
[7]http://docs.openstack.org/infra/jenkins-job-builder/publishers.html

Reporting test results should not be about failed tests only, but also about how many of them were executed, how long they have run and so on.

Jenkins provides a support for email notifications and the only thing to be done to use it is to manually select receivers email addresses.

A *Jenkins* job configuration can be created through GUI or via the *YAML* files. The *Freestyle software project* job in the *YAML* format can be seen in Example 2.1. This example shows an *eDeploy* unit tests configuration. It contains options explained before. This job first clones *edeploy* repository hosted on *github*, provided by *SCM* option. The *triggers* options provides information that *Jenkins* will execute this job hourly. The job behaviour is controlled by *shell* commands in *builders* option. The final option is *publishers*, which collates *junit* test results, *pylint* static analysis violations or sends a notification email to relevant recipients. Thanks to the *YAML* format, *Jenkins* jobs can be easily created and shared between other CI teams.

```
- job:
    name: eDeploy-UnitTests-YAML
    description: 'Do not edit this job through the web!'
    project-type: freestyle
    block-downstream: false
    scm:
      - git:
          skip-tag: false
          url: git@github.com:enovance/edeploy.git
    triggers:
      - pollscm: '@hourly'
    builders:
      - shell: |
          git clean -dxf
          sloccount --duplicates --wide --details . > sloccount.sc
          find . -name test\*.py|xargs nosetests --with-xunit || :
          rm -f pylint.log
          for f in `find . -name \*.py|egrep -v '^./tests/'`; do
            pylint --output-format=parseable $f >> pylint.log
          done || :
          python /usr/local/python2.7/dist-packages/clonedigger.py:
    publishers:
      - warnings:
          workspace-file-scanners:
             - file-pattern: pyflakes.log
               scanner: PyFlakes
      - junit:
          results: nosetests.xml
      - sloccount:
          pattern: sloccount.sc
      - violations:
          cpd:
            pattern: output.xml
          pylint:
            pattern: pylint.log
      - email:
          recipients: devops@mycompany.com
```

Listing 2.1: eDeploy Unit tests configuration in the YAML format.

Chapter 3

Continuous Integration Implementations at Red Hat Company

This chapter will explain how three chosen teams at Red Hat are using *Jenkins* for their continuous integration. The workflows will be first presented through the CI pipeline designs the teams are using, followed by a deep analysis of each job. The identifications and descriptions of these processes are an important part of this thesis. The teams' names are kept confidential as per Red Hat request, but it does not affect the content of this work. New metadata format is proposed at the end of this Chapter.

3.1 Team 1

The continuous integration design for this team is in Figure 3.1. As can be seen, it is divided into three main behaviour parts: *test on build*, *covscan*, and *RPMDiff*. These parts are described in detail in the subsections below. That leaves us with just one job to describe here.

The *Component setup* job is used as a simple component registration. A new component adding to CI, must be done by manual configuration of the *Component setup* job which expects the component and package collection name. After that it is needed to choose which behaviours (*Test on build*, *Covscan* or *RPMDiff*) should be activated for this component: whether it will be the *Test on build* part only or *covscan* or some combination of all the three behaviours. After this job is built, the *Listeners* and *Dispatchers* jobs for the chosen behaviours are uploaded to *Jenkins* server. When this is all done, the component is successfully registered and can be tested by the continuous integration framework designed by Team 1.

3.1.1 Test on build CI

Test on build behaviour is the most common part of the whole continuous integration. The purpose of this part is to provision a new machine with a clean operating system, setup a machine environment, run tests, send the results of the tests, and tear down the provisioned machine.

This behaviour part contains five *Jenkins* jobs:

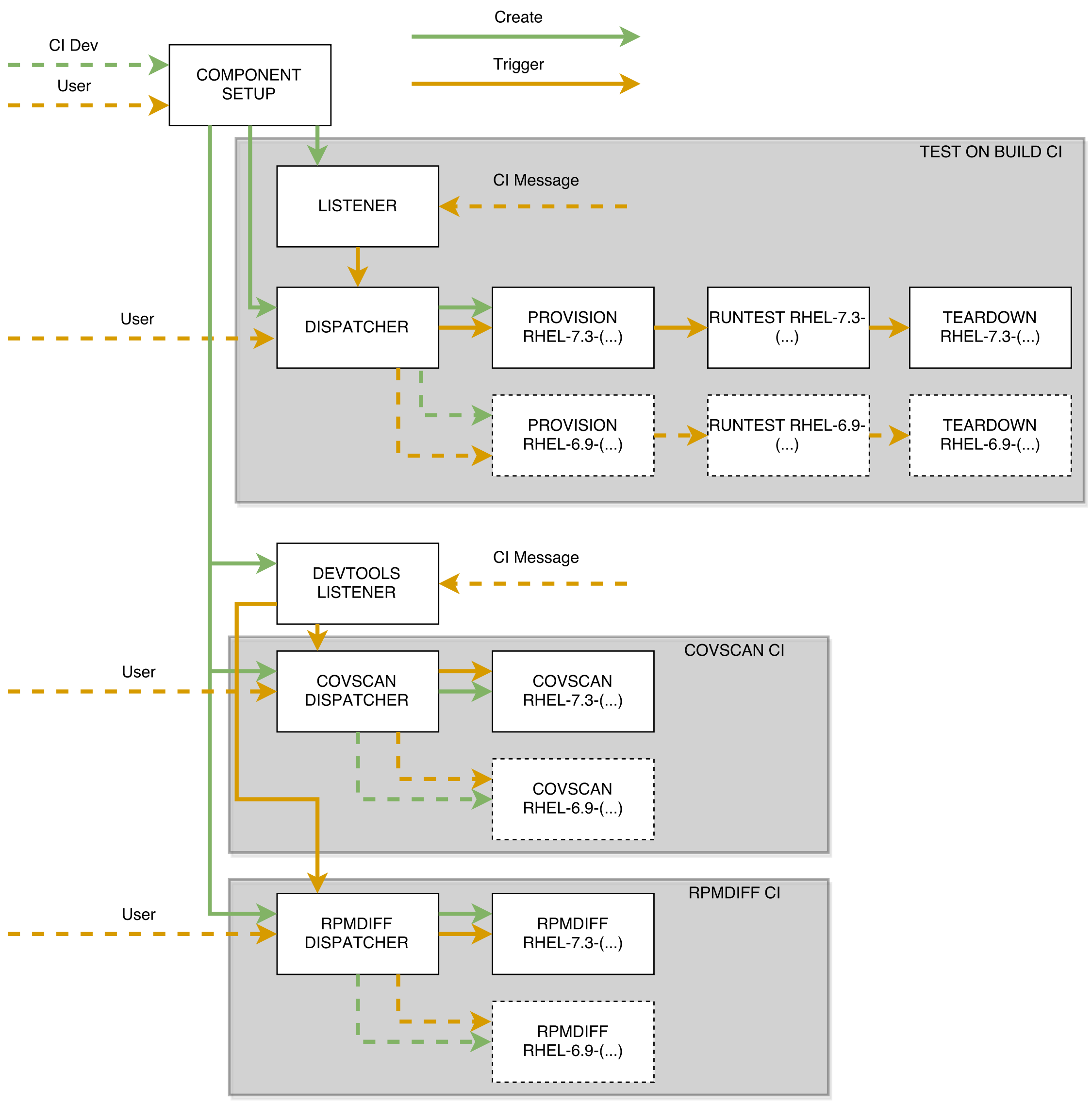


Figure 3.1: Team 1 continuous integration design using the Jenkins tool.

- **Listener**. This job gets automatically triggered whenever a component is successfully built in a building system. Its only job is to extract the build task ID from the CI message. The *Listener*'s action is to trigger a component *dispatcher* job for the given build task.

- **Dispatcher**. The *Dispatcher*'s job is automatically triggered by the *Listener* job, or it can be run manually through the *Jenkins* web GUI; build task IDs, however, need to be inserted manually. The basic purpose of this job is to create or update the *Provision*, *Runtest* and *Teardown* jobs on the *Jenkins* server for a given build target and task ID. After that, the *Dispatcher* job will trigger the *Provision* job. The newest job configuration is polled from team 1's CI repository, where it is stored in the YAML format. Therefore, by using *Jenkins Job Builder*, it is easy to update them.

- **Provision**. *Provison* is the first job in the *Provision*, *Runtest* and *Teardown* pipeline. Its main purpose is to provision a new machine for a given build target. After that, this job downloads the metadata for the *Runtest* job, such as the test case, notification metadata, machine information, etc. The *Provision* job is triggered by the *Dispatcher* job and it should not be manually triggered. It should always be automatically triggered by the *Dispatcher* job. If the *Provision* job succeeds, it triggers the *Runtest* job. If it fails, it triggers the *Teardown* job.

- **Runtest**. The main purpose of this job is to run tests on the provisioned machine and send results to relevant recipients. However, at the beginning, this job sets up the machine environment. That includes installation of a given component, setting up repositories and setting up the testing tool. After a successful machine environment setup, tests are executed. When this step is done, test results are sent to relevant notifiers, who had been gathered in the *Provision* job. At the end, *Runtest* job will trigger the *Teardown* job.

- **Teardown**. The last remaining step is to clean all provisioned resources. This is done by the *Teardown* job. After that, the entire *Test on build* pipeline is done.

3.1.2 Covscan CI

Team 1 CI solution contains a *Covscan* feature for the component. The *Covscan* stands for coverity scan static analysis, whose the purpose is to run static analysis over every line of the code. *Covscan* CI jobs are:

- **Devtools Listener**. This listener job is common with the *RPMDiff* CI part. The behavior is similar to the *Listener* job in *Test on build* part. It is automatically triggered after a successful build in the building system, but since this listener is common for more than one job, it includes a decision mechanism which chooses which dispatcher should be triggered.

- **Covscan Dispatcher**. It expects a build task ID as an input and is automatically triggered by the *Devtools Listener*. *Covscan dispatcher*'s purpose is, again, similar to *Test on builds Dispatcher* job, which is only the *Covscan* job actualization. After that, the *Covscan* plugin is triggered.

- **Covscan**. The *Covscan* job is the closest to the *Runtest* job in the *Test on build* part. The only exception is that is does not set up any machine environment, because it does not have any. It only runs the configured `covscan` tool, which performs the coverity scan. When the coverity scan is complete, *Covscan* results are sent to the relevant notifiers.

3.1.3 RPMDiff CI

The purpose of this feature is to search for differences between two RPM packages. The design of this feature is very similar to the *Covscan CI*, but it uses different jobs such as:

- **Devtools Listener**. This listener job is common with the *Covscan CI* part and has been described above.

- **RPMDiff Dispatcher**. It is really similar to other described dispatchers (*Test on build* or *Covscan*), with the difference that it actualizes and triggers the *RPMDiff* plugin only. It can be run manually or automatically.
- **RPMDiff**. This job runs the *RPMDiff* tool only, which does the comparison. After its behavior is complete, it reports the results. At the end, *RPMDiff* results are inserted into a notification template and sent to relevant notifiers.

3.2 Team 2

The continuous integration solution of Team 2 was based on Team 1's solution as can be seen in Figure 3.2. The *Covscan CI* and the *RPMDiff* are not supported for this team. *Test on build CI* remains, with the little change.

The main difference is that *Dispatcher* job triggers a *Tier1* job instead of the *Provision* job. This represents a future preparation for higher tier jobs. It means that the *Tier1* job is similar to the *Dispatcher* job in *Test on build CI* of Team 1. Another difference is that the *Tier1* job triggers multiple *Provision* jobs for a single component, which is a specific behavior for Team 2.

The remaining jobs are not described, because their functionality is the same as in team 1's CI.

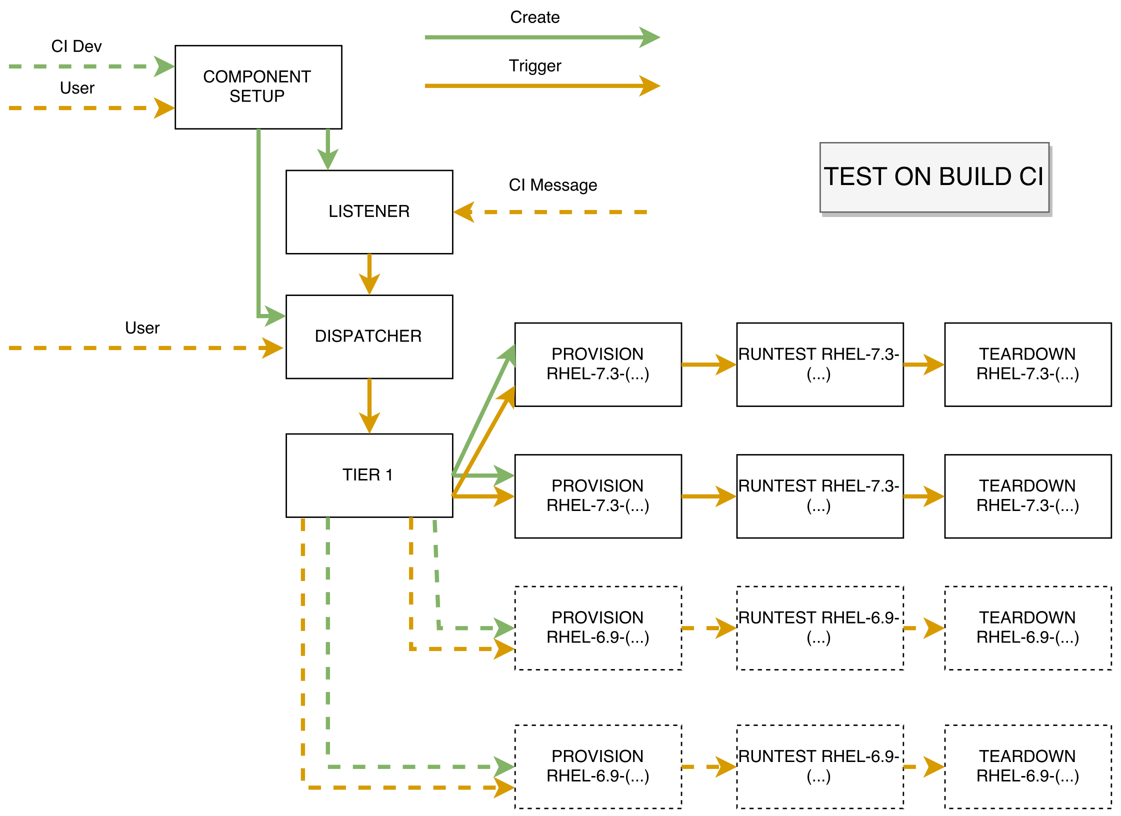


Figure 3.2: Team 2 continuous integration design using the Jenkins tool.

3.3 Team 3

Team number 3 is fully integrated into the *Platform CI* team, and their solution is called *MVP*[1]. Even though this solution is merged into shared *Platform CI* team solution it is used by Team 3 only. The design of *MVP* is quite different and can be seen in Figure 3.3. It contains two jobs:

- **Trigger**. This job behaves similarly to the *Listener* job used in Team 1 Example 3.1. It is automatically triggered by a CI message from a successful component build in the building system. As a benefit, the *Trigger* job can also be triggered manually. When the *Trigger* receives the CI message, it extracts the build task ID. After this extraction, the build task is validated. If the validation is successful, it triggers a *Component-MVP* job.

- **Component-MVP**. This is the main job of the whole CI functionality. The *Component* in *Component-MVP* job's name is replaced by the name of the actual component, for example httpd, curl, cups, kernel, etc. It is triggered by the *Trigger* job. The behavior of this job is fairly simple. It only runs a configured *beaker* command, nothing else. Subsequently, *Beaker* creates an appropriate machine, sets it up and runs tests on it. In other words, everything is handled by *Beaker* tooling. At the end, the *Trigger* job gathers the test results and sends them to the *xUnit* reporter or Jenkins Dashboard.

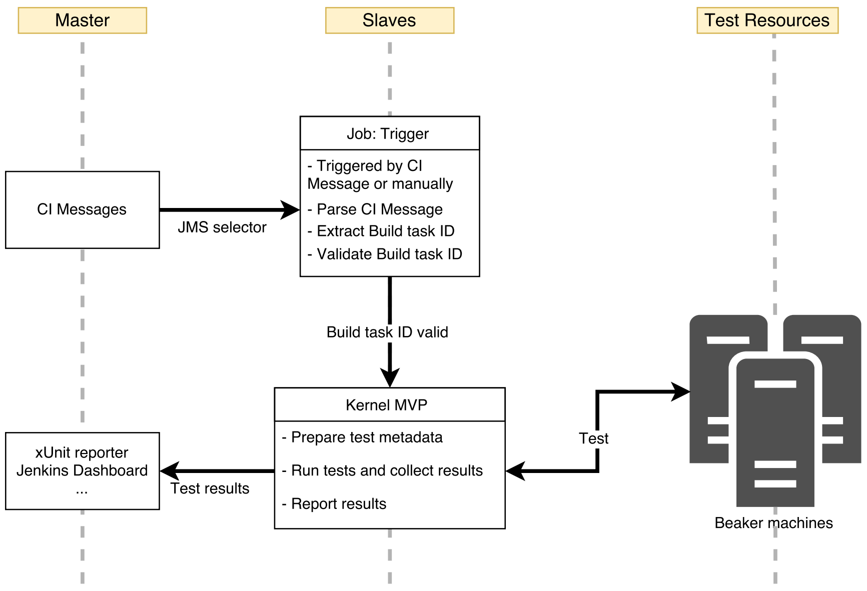


Figure 3.3: Team 3 continuous integration design using the Jenkins tool.

In this solution, there is no *Component setup* job for component registration. The registration is done by `jenkins-jobs.sh` script. As a component configuration the `sample_job.yaml` file is used. Created configuration file can be tested by the `jenkins-jobs.sh` test. If the test does not find any problems, then component registration is fulfilled by `jenkins-jobs.sh update` command execution, which will upload component configuration to the *Jenkins* server.

[1]https://github.com/RHQE/platform-ci/tree/master/MVP

3.4 Platform CI

The last solution described is the open source *Platform CI*[2] project. This project was created with a desire to create a unified solution for all *Platform* teams in Red Hat. By the time it reached one year of its existence, the *MVP* and *Staging branch CI* had been developed. As we know from previous sections, the *MVP* is too specific, so only one team can use it. That is why this section is focused on the *Staging branch CI*.

Purpose of *staging branch* is to test future git branches from which new versions of *Red Hat Enterprise Linux* will be build. Design of *staging branch* can be seen in Figure 3.4. It consists of these jobs:

- **Component setup**. A similar registration job for components as is in Team 1 and 2's solutions.
- **Commit dispatcher**. The *Dispatcher* job which is automatically triggered by a CI message. After that, it updates the component's Build branch job.
- **Build branch**. This job is triggered by the *Commit dispatcher* job. Its purpose is to tell the building system to create a scratch build of this branch. If the build is finished successfully, the other CI teams can react to a CI message which is created after the building process.

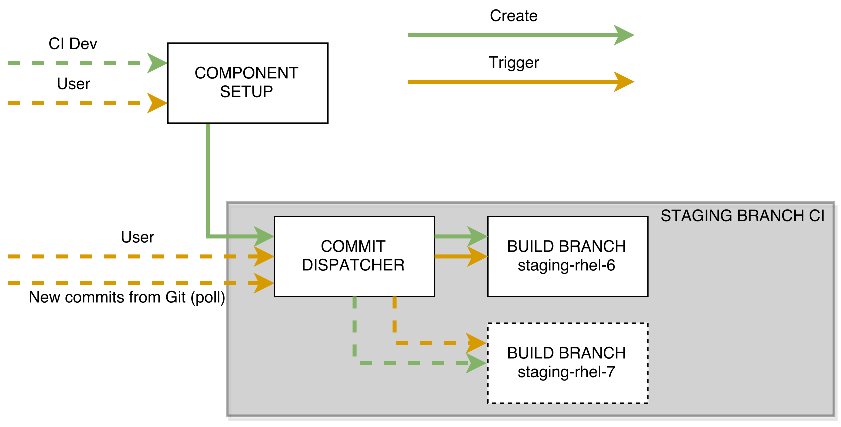

Create
Trigger
CI Dev
User
COMPONENT SETUP
User
New commits from Git (poll)
STAGING BRANCH CI
COMMIT DISPATCHER
BUILD BRANCH staging-rhel-6
BUILD BRANCH staging-rhel-7

Figure 3.4: Platform Staging branch CI design using Jenkins tool.

Staging branch test is done by trying to create a component build from the selected branch. If the component build is successful, a CI message is posted on messagebus, which is listened to by other CI solutions. Therefore the CI message created by the component build will start Team 1, Team 2 and Team 3 CI behaviour.

3.5 Continuous Integration Implementation Summary

Staging branch test is done by trying to create a component build from the selected branch. If the component build is successful, a CI message is posted on messagebus, which is listened to by other CI solutions. Therefore the CI message created by the component build will start Team 1, Team 2 and Team 3 CI behaviour.

[2]https://github.com/RHQE/platform-ci/

Chapter 4

Test Metadata Analysis

The *Platform* team is taking care of *Red Hat Enterprise Linux* development and testing. This team contains of several smaller teams, three of whom took part in the previous Chapter. The unification on continuous integration jobs would be difficult because of specific behaviours described in Chapter 4. On the other hand, teams could be unified on the level of metadata they are using. The problem is that every team is using multiple storage systems for metadata. This Chapter brings an analysis of the storage systems used in Red Hat's teams. It also suggests possibilities of new storage systems and metadata formats. For better understanding, the metadata types are described in this Chapter.

4.1 Metadata overview

In this text, the term *metadata* is used several times. However, it has various meanings that are described below:

- **Test job metadata**. It consists of all metadata needed for CI job execution. For example *job name*, *build number* or *job url*.
- **Test provision metadata**. To run a test, we need to provision correct testing environment, install and setup software under the test, and install and setup the testing framework. All these steps could be done in multiple ways and need specific metadata describing how to do it for the given test. Examples include the *number of resources*, *resource type* or *resource credentials*.
- **Test run metadata**. When the testing environment is provisioned, it is needed to know how the test shall be executed. Will it be *test harness*, *test framework*, *test dependencies* or *test configurations*.
- **Test result metadata**. The point of running a test is to get its results and the resulting test artefacts. For instance, pass or fail information together with structured or unstructured logs. It is simply aggregated data of all run jobs with information where and how the results should be achieved.
- **Test report metadata**. Where to publish the results and what type of results need to be published. For example *quality engineer owner*, *developer owner* or *results url*.

4.2 Existing External Solutions

The first step was to find an ideal storage system that would tend to the needs of all teams. According to article [15], the best solution should be a type of *Test Data Management*. Other storage choices are *text files*, *spreadsheets*, *Relational database management system*, *XML* or *Application configuration files*, according to the mentioned article.*Xqual*[1] website provides plenty of information comparing existing *Test Data Management* tools and each one's pros and cons. The *Xqual* page compares existing *Test data management* systems, discussing whether they provide properties such as *Integrated bug-tracking system*, *Manual testing*, *Automated testing drivers*, *Unit testing*, and many more. The best free *Test data management* system, according to the article, is *XStudio* from *XQual* company. Still, this solution provides only management properties without any automated test execution support. A tool that combines management and automated test support is *Expecco* from *Exept Software AG*, but it is commercial. The analysis shows that there is no existing solution which would solve all Red Hat teams's problems.

4.3 Analysis of used metadata storage systems

The second part of this master thesis is focused on the usage of metadata storage systems in the described Red Hat teams. Tables 4.2 and 4.1 map metadata types to storage systems. Lists of metadata types are shown in the left column in both tables. Storage systems are shown in the first row in both tables.

		git type4	Jenkins	pytest	jjb yaml	git type5	github	git type6	PDC	Beaker
Execution record	**Team1**									
	Team2								1	1
	Team3									
Test ownership	**Team1**									
	Team2				1		1			
	Team3									
Dependency (Test)	**Team1**									
	Team2									
	Team3			1				1		
PEPA Rules	**Team1**	2								
	Team2									
Deployment source	**Team1**			1						
	Team2									
Environment Requirements	**Team1**					1				
	Team2									

Table 4.1: Metadata x Storage for each team.

[1]http://www.xqual.com/qa/tools.html

Table 4.2 shows that *TCMS* is the most used tool for storing metadata. The least used storage systems are *wiki*, *git type3*, *github*, *Jenkins*, *pytest*, *jjb yaml*, *PDC* and *Beaker* according to Tables 4.2 and 4.1. Interesting fact is that the tables contain seven different types of *git* (git type 1 up to type 6 plus github) storage. If all *gits* were unified, then *git* storage would be the most used storage system.

		TCMS	git type1	UNKNOWN	Polarion	Makefile	git type2	wiki	git type 3
Execution record	Team1	1							
	Team2			1					
	Team3			1				1	
Timeout (Test)	Team1	2					2		
	Team2				2				
Test ownership	Team1	1					1		
	Team2								
	Team3		1	1					
Dependency (Test)	Team1						1		
	Team2				1				1
	Team3								
Component relation	Team1	1							
	Team2		1						
	Team3		1						
Relevancy	Team1	2							
	Team2		2						
PEPA Rules	Team1	2							
	Team2		2						
HW Requirements	Team1	1							
	Team2		1						
	Team3					1			
Execution order (Test)	Team1	1							
	Team2		1						
	Team3					2			
Test existence record	Team1	1							
	Team2	1							
CI Tier	Team1	1							
	Team2		1						
	Team3					1			
Static parameters (Test)	Team1	1							
	Team2		1						
EWA rules	Team1								
	Team2		2						
Test status	Team1	1							
Deployment source	Team1								
	Team2		1						
Environment Requirements	Team1								
	Team2		1						
Destructivity	Team1	1							
	Team2		1						
Team1 Internal Tier	Team1	1							
	Team2		1						

Table 4.2: Metadata x Storage for each team.

4.4 Unified Test Metadata Format

According to previous sections, we know that *git* system is the most used metadata storing solution. Every continuous integration team stores their test metadata differently. Therefore a new CI test metadata format needed to be created. The main requirement was to have a metadata format which would be easy to create, process and easy to read. To achieve this requirements, the *yaml* format was chosen. *Yaml* is a human-readable data serialization language which is commonly used for configuration files. This is a good presumption, but it needs to be provided with a human readable format. The proposed unified format for storing CI test metadata can be seen in Listing 4.1.

```
Owner: Jiri Kulda jkulda@redhat.com
Version: 1.0
Type: Sanity
TestTime: 480m
Requires:
  - openscap
  - openscap-utils
  - libxml2-devel
  - rpm-devel
  - libgcrypt-devel
  - pcre-devel
  - python-devel
  - libxslt-devel
  - libacl-devel
  - libcap-devel
  - libnl-devel
Environment:
  SRPM: http://some/server/with/package.srpm
Description:
  Simple test description
Relevancy: |
  distro != rhel-5: False
  distro < rhel-5.9 && arch != x86_64: False
  component == java-1.8.0-openjdk: path/to/java/jdk
Dependencies:
  - /path/to/metadata
    name: important data
```

Listing 4.1: Unified format for storing CI test metadata.

Above CI test metadata format contains these values:

- *Owner.* Name and email address of test owner. Can be used as a *test report metadata.*
- *Version.* Test version. Used for *test result metadata.*
- *Type.* Test type such as *Sanity*, *Smoke*, *Black box* tests, etc. Mainly used for *test result metadata.*
- *TestTime.* Test running limit. If this limit is reached, the test should be terminated with *fail* status and CI system should send relevant information to test owner and component owner. Part of *test run metadata.*
- *Requires.* Contain list of packages which need to be installed before test execution. Important part of *test provision metadata.*

- *Environment.* Needed test environment rpm packages. Important part of *test provision metadata.*
- *Description.* Few lines length test description. Can be used in *test report metadata.*
- *Relevancy.* Information whether this test is relevant for specific operating system distribution. The test would not be executed if the provisioned distribution would not satisfy *Relevancy* section. Very important for *test run metadata.*
- *Dependencies.* Contains list of other test metadata files which are needed for test execution. Part of *test run* or *test provision metadata.*

4.5 Proposed Unification Tool

The proposed unified format for CI test metadata provides a possibility to unify all *git* storage systems across all teams. However, according to test metadata analysis in Section 4.3, CI teams use a large palette of storage systems. Another problem is that every CI team is using a different approach to retrieve their test metadata (because of different storage). It is thus difficult to create a unified CI framework solution which would be suitable for multiple QE teams.

To solve this problem, an automated tool needs to be created that would feed the needed test metadata into *Jenkins* and/or other tools as appropriate. After this tool creation, it is possible to design and implement new job plugins without the metadata storage dependency. This tool is described in the next chapter in more detail.

Chapter 5

Metamorph

Metamorph is a tool which will be transforming the acquired metadata from storage systems to a machine readable format. The name *Metamorph* was created from the word metamorphosis, which represents the transformation from an egg to a butterfly. *Metamorph* tooling will provide such a transformation to get metadata from storage systems for the purpose of new CI tool solution. *Metamorph* will be responsible for converting existing raw metadata to a format that can be parsed by other CI tools. It can also be defined as a one time metadata carrier in the CI pipeline.

This chapter will provide deep information about the *Metamorph* tool design. This design will be presented using high and low level diagrams. Implementation of this tool will be described in the second part of the chapter. The last section contain the *Metamorph* testing information.

5.1 Metamorph Design

5.1.1 Problem Definition

Every team stores their test metadata in a different storage system (seen Analysis 4.3). The problem is that every CI team is using different approaches to retrieve their test metadata because of different storage systems or data formats. Therefore it is difficult to create a unified continuous integration tool which can be suitable for multiple QE teams.

To solve this problem, we shall create a tool (*Metamorph*) which will extract metadata from any existing metadata storage system as appropriate and then append the metadata to a file which is then passed on to other parts of the pipeline for further actions based on the metadata in it. With this in place, it is possible to design and implement new plugins without a CI test dependency focused on the type of the storage system. The *Metamorph's* core responsibility is to append respective metadata to `metamorph.json` (default if not configured) in a standard format throughout the pipeline where necessary. Continous integration pipeline was described in Subsection 2.4.3.

5.1.2 Metamorph Requirements

This section contains most of *Metamorph* tool requirements. These requirements were obtained through discussions with QE teams representatives for continuous integration. General requirements of *Metamorph* are:

- Accept raw metadata in the *YAML* format.

- Query *TCMS* for metadata.
- Create and append metadata to `metamorph.json`.
- Be available as a Python library.
- Ansible modules for created plugins so that one can have desired tasks in his playbook.
- Inform if a component build should be tier tagged or not.
- Be able to support custom workflows e.g. „for package XY, I want to run only tests on s390x as my package exists only on s390x“.

Metamorph quality standard requirements are:

- to have Unit tests,
- to have CI setup,
- to have up-to-date documentation,
- to follow PEP8 Python standards.

5.1.3 Metamorph Requirements for Jenkins Jobs

This subsection describes the *Metamorph* requirements for common *Jenkins* jobs used in continuous integration.

Listener Job

The purpose of this job is to subscribe to CI messagebus and launch a *Jenkins* job when a matching CI message is received. The contents of the message is stored in an environment variable named `CI_MESSAGE`. *Listener* job requirements are:

- The *listener* should provide a way to extract message contents from the environment variable and dump it into `metamorph.json`.
- Offer a possibility to specify the output file as desired and default to `metamorph.json` if not configured.

Dispatcher Job

The purpose of the dispatcher job is to decide which jobs will be executed based on the CI message. Requirements for *Dispatcher* job are:

- Load `metamorph.json` using *Ansible* playbook.
- Extract and return individual message fields from `metamorph.json` for any further user actions.
- Extract and return multiple message fields from `metamorph.json` for further user actions.
- Fetch raw metadata from the above provided storage source and data type.

- Create a mapping file in the *Metamorph* for *linch-pin*[1] provisioning tool.
- Retrieve the image name from *Product Definition Center*[2] for the given component.

Provision Job

The purpose of this job is to provision a new machine for a given build target. The requirement is to be able to provision resources using *linch-pin* with `metamorph.json` as an input.

5.1.4 Metamorph Benefits

This subsection contains a list of benefits which *Metamorph* will bring. General benefits are:

- Common tooling between teams in Red Hat CI to perform actions based on metadata.
- Increased collaboration between teams.
- Metadata footprints at any given point in the continuous integration pipeline.
- Common tooling for Red Hat distributions (if they have a common metadata schema).

Values for other Red Hat teams are:

- Quality engineering. Teams can benefit by reusing the *Metamorph* plugins instead of developing and maintaining their own to get metadata. If they use *Metamorph* in their infrastructure, it will be easy to restart a job in their continuous integration pipeline in case of an infrastructure failure. In general, any other services can be offered/deployed easily.
- Developers. Developers can use metadata provided by *Metamorph* and feed it into their local continuous integration, thereby enabling them to run specific tests and not worry about any of the provisioning or test run metadata.
- Team1. *Metamorph* can be the sole interface for Team1 tooling.
- Team2. All the metadata could be stored in Test Run templates, which may help in triggering.
- Platform CI. Specific sections of metadata could be published in the message bus in a consistent way for all on-board teams for any further actions.

5.1.5 Use Cases

Metamorph use cases are shown in Figure 5.1. The general use case is to run *Metamorph* at the beginning of a continuous integration job. It will aggregate metadata from various storage systems. Afterwards, it will append the metadata to existing `metamorph.json` or it will create a new `metamorph.json` file. The new `metamorph.json` file will be shared between jobs in the CI pipeline. Appending new metadata to existing ones is valuable because afterwards these metadata will be shared as well. Sharing metadata is a major benefit because multiple CI jobs can use the same metadata and so there is no need to gather them again and waste valuable time.

[1] http://sexysexypenguins.com/posts/introducing-linch-pin/
[2] https://pdc.fedoraproject.org/

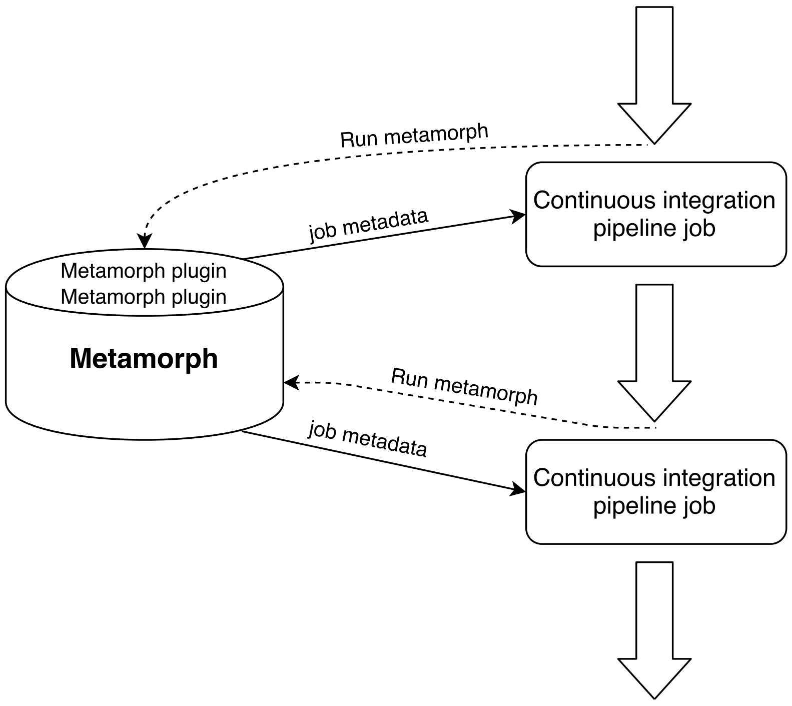


Figure 5.1: Metamorph tooling usage.

Storage Systems Automatic Analysis Use Case

Metamorph expects a component name and type of storage (git, resultsDB, PDC, ...). With this knowledge, the tool will automatically know how to gather all needed test metadata from the given storage. Therefore, *Metamorph* would need to store as many as possible storage systems internal metadata mappings. However, some teams are using the same storage systems, so there would not be a big need to create a specific solution for each Red Hat CI team.

Given Repository Analysis Use Case

Metamorph will expect a path to a test repository and type. It could be git, resultsDB, PDC or anything else, and, from this repository, the test metadata file will be created. The difference between the *given repository analysis* and the *storage system automatic analysis* is that the *storage system automatic analysis* is gathering metadata related with a specific job name. On the other hand, *given repository analysis* is searching for any possible metadata in the provided repository. The algorithm will mainly search for configuration files and parse data from them.

5.1.6 Metamorph Output Format

The *Metamorph* metadata output will be used by different CI jobs and therefore it needs to be formatted in some data-interchange language. The *JSON*[3] format was chosen for the *Metamorph* project because first of all it is really easy for machines to parse and generate.

[3]http://www.json.org/

Other benefits are human readability and language independent format which is close to the C programming family.

5.1.7 Implementation Design

The *Metamorph* tool should be easy to run and be system independent. Because of its focus on particular storage systems, *Metamorph* would need to be developed in a form of plugins where one plugin will match one storage system.

How *Metamorph* should interact with storage systems can be seen in Figure 5.2. The top half of the figure describes *Metamorph* interaction with two storage systems. A continuous integration job will execute (request arrow) the *Metamorph* tool to get metadata from *PDC* and *resultsDB* storage systems. *Metamorph* then launches each metadata storage plugin to handle the metadata extraction. The tooling plugin extracts metadata from storage systems and appends the gathered metadata with an existing `metamorph.json` file in the *JSON* format. This file is returned (the return JSON arrow) to the continuous integration job. The bottom half of the figure is similar to the top half. The difference is that, in the bottom half, a *Metamorph* plugin searches for *report* and *results* metadata, but the top half can be used for provision, run or job metadata. In this part, *results* metadata are taken more as simple data than metadata because it will mainly contain data about the result itself and not the result test metadata. An advantage of this logic is that all plugins are

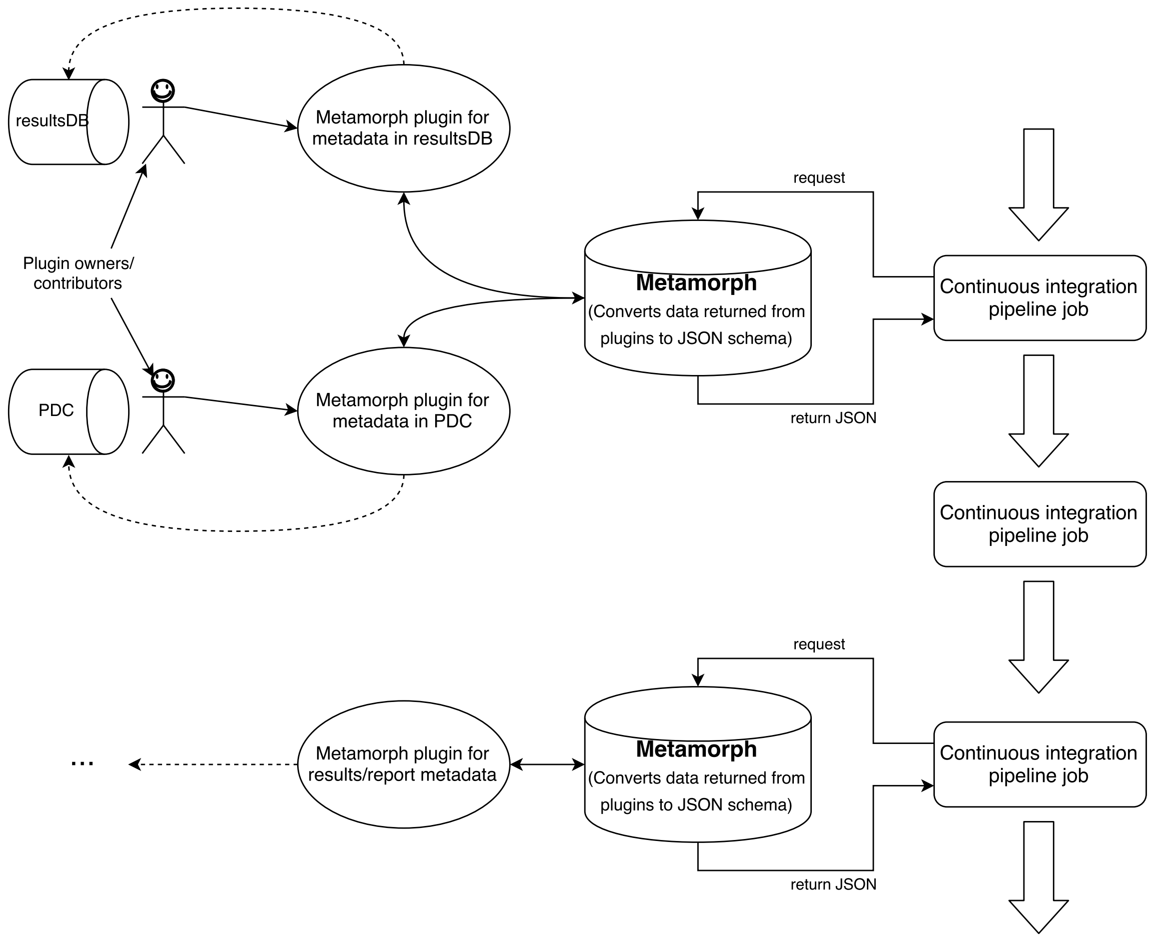

resultsDB
Metamorph plugin for metadata in resultsDB
Plugin owners/ contributors
PDC
Metamorph plugin for metadata in PDC
request
Metamorph
(Converts data returned from plugins to JSON schema)
return JSON
Continuous integration pipeline job
Continuous integration pipeline job
request
...
Metamorph plugin for results/report metadata
Metamorph
(Converts data returned from plugins to JSON schema)
Continuous integration pipeline job
return JSON

Figure 5.2: Metamorph tooling implementation usage.

independent of each other, so it is possible to run them in parallel and save valuable time.

5.1.8 Interactions With Storage And Metadata

This subsection describes how *Metamorph* will interact with different storage systems in both use cases.

Storage System Automatic Analysis Use Case

Metamorph will expect name (typically build name), type of storage (git, resultsDB, PDC, ...) and data format (Makefie, YAML ...) or only name with extraction metadata type (provision metadata, job run metadata, ...). With this knowledge the tool will automatically know how to gather all needed test metadata. There are multiple ways how this information can be passed to the *Metamorph*.

- Have this information sent through the message bus as part of the initial *CI_MESSAGE*.
- Have the component register with *Metamorph*. Configuration would need to be carried in *Metamorph* repository. For e.g., in `metamorph/mappings` file.
- Have the job/task owner provide it as a command-line arguments in the form of name (typically build name), storage (for e.g., TCMS, git, etc) and data format (for e.g., YAML, etc) and run modules provided by the respective team.
- Have only name provided. In case that no storage type is selected, Metamorph will run all plugins as default. This approach would get big amount of metadata but it would also take some time. Therefore it is not recommended.
- Same as above, but with only name and extraction metadata type (provision metadata, job run metadata, ...).

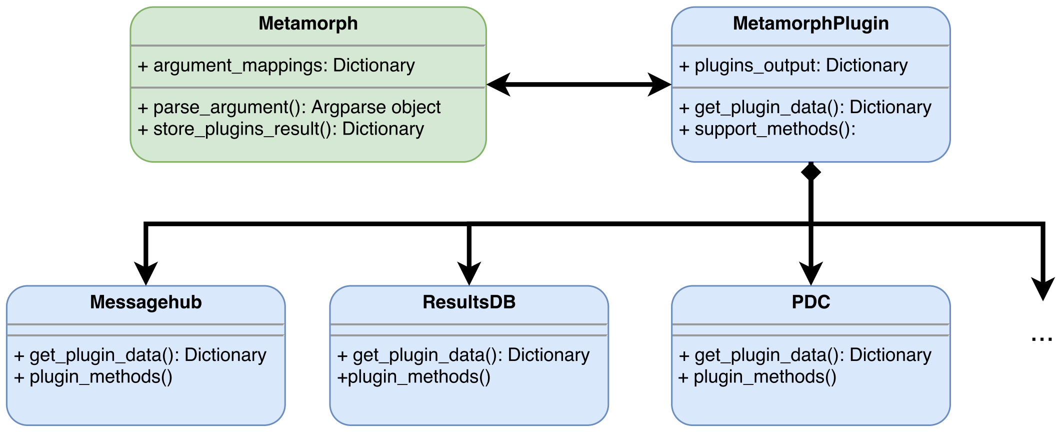
Given Repository Analysis Use Case

In this approach, *Metamorph* will expect a path to repository/storage and files which contain important metadata. The tooling will need to have some internal file recognition to recognize file types. Some ways how to pass all information to *Metamorph* can be seen below.

- Have the job/task owner provide it as command-line arguments containing only path to repository/storage. The tooling will recognize type of repository/storage and according to internal mapping it will know which files contain metadata and how to extract them. For *git* repository, it can be *Makefile* or a specific *YAML* file which will contain them.
- Similar as above, but with a provided list of important files. Internal mapping would need to be still needed, but not as big.
- Same as above with metadata extraction information provided.

5.1.9 Low Level Implementation Design

The low level implementation design of *Metamorph* which can be seen in Figure 5.3 is divided into two main classes: *Metamorph* (green color) and *MetamorphPlugin* (blue color).



Metamorph
+ argument_mappings: Dictionary
+ parse_argument(): Argparse object
+ store_plugins_result(): Dictionary
MetamorphPlugin
+ plugins_output: Dictionary
+ get_plugin_data(): Dictionary
+ support_methods():
Messagehub
+ get_plugin_data(): Dictionary
+ plugin_methods()
ResultsDB
+ get_plugin_data(): Dictionary
+plugin_methods()
PDC
+ get_plugin_data(): Dictionary
+ plugin_methods()
...

Figure 5.3: Low level implementation design.

MetamorphPlugin Class

The purpose of this class is to provide a unified interface for other plugins, which will inherit from this class. Inherited classes will contain logic to extract metadata from specific metadata storage types, e.g. *PDC*, *resultsDB* or *Messagehub* plugin. The *MetamorphPlugin* class contain:

- **plugins_output**. It is a static class variable which contain data from selected plugins. If some plugin finishes gathering metadata from specific storage, it appends data into this variable. When all plugins finish their jobs, `metamorph.json` will be created from this variable.
- **get_plugin_data()**. This method will be automatically executed for every selected plugin. The main logic of every plugin will be implemented in this method. After this method execution, the extracted metadata should be present in *plugins_output* variable.
- **supportive_methods()**. This method is here to explain that *MetamorphPlugin* will also contain methods which can be used by inherited plugins. The intention is to use shared common methods across plugins and not to implement new ones again. Simply, use what is already created. For example, it can be a method for querying *api* interface, which would be useful for *PDC* and *resultsDB* plugins.

Metamorph Class

The *Metamorph* class will contain a decision logic based on input arguments. The main purpose is to run selected plugins by given input and collate their data through *plugins_output* variable. The *Metamorph* class contain:

- **argument_mappings**. This argument mapping will be mainly used for *Metamorph* execution by provided metadata type (provision, job run, ...). Plugins will be executed by these mappings. There are two ways how to do this mapping.
  - Every storage plugin inherited from the *MetamorphPlugin* will contain a variable which will indicate the type of metadata provided in it. This approach can be

executed without any internal mappings. Simply run all the tooling plugins and if their type matches the needed metadata type, then execute plugin behavior.

  – The second way is to have an internal mapping which can be seen in the Listing 5.1. The listing shows that to get *provision* metadata, plugins *morph_dispatcher* and *morph_pdc* will be executed. The same logic will be used for remaining metadata types. The disadvantage of this approach is that, after every plugin creation, *argument_mapping* must be changed to support the new plugin.

- **parse_arguments()**. Specified input options in *argparse* format.
- **store_plugins_result()**. Method for storing plugins result. This method can create and also append data to existing `metamorph.json` files.

```
{
  "provision-metadata": ["morph_dispatcher", "morph_pdc"]
  "run-metadata": ["morph_tcms", "morph_pdc"]
  "report-metadata": ...
  ...
}
```

Listing 5.1: Metamorph argument mapping example.

5.1.10 Source Directory Structure and Output Metadata Format

The whole project design is focused on implementation in the *Python* language. With this focus, a specific directory structure was created as one can see in Listing 5.2. The provided example contains a structure with double `metamorph` directories. This design is needed for the *Metamorph* installation. All files in the `metamorph/metamorph/` and `metamorph/bin/` folders are installed into `site-packages/metamorph` folder in running system. Afterwards, *Metamorph* plugins can be imported or executed.

```
metamorph/README.md  # Metamorph documentation
metamorph/metamorph/lib/  # common libraries
metamorph/bin/metamorph  # Metamorph executable file
metamorph/metamorph/etc/  # configuration files
metamorph/docs/  # documentation using sphinx
metamorph/tests/  # nosetests
metamorph/outputs/  # default location for outputs
metamorph/ex_schemas/ # Example schemas
metamorph/metamorph/library/  # Ansible modules
metamorph/metamorph/plugins/  # Metamorph plugins
metamorph/metamorph/plugins/morph_pdc.py
metamorph/metamorph/plugins/morph_tcms.py
metamorph/metamorph/plugins/morph_polarion.py
metamorph/metamorph/plugins/morph_yaml.py
metamorph/metamorph/plugins/morph_makefile.py
metamorph/metamorph/mappings/  # mappings for component metadata
metamorph/metamorph/mappings/yaml_mappings.{json|yaml}
metamorph/metamorph/mapping/tcms_mappings.{json|yaml}
metamorph/metamorph.py  # Contains Metamorph class
metamorph/metamorph_plugin.py  # Contains MetamorphPlugin class
metamorph/setup.py  # Python setup script
```

Listing 5.2: Metamorph source directory structure.

For the output, the *JSON* format was chosen in Section 5.1.6. An Example of a simple **`metamorph.json`** can be seen in Listing 5.3 in the *JSON* format. The first field is named **`metamorph`**. Every metadata file created by *Metamorph* tool will begin with this name. This field contain data from existing plugins, for example *resultsDB* or *PDC*. The first plugin value will always be **`results`** followed by extracted metadata.

```
{
"metamorph":
  {
    "pdc":
    {
      "results": {..}
    },
    "resultsdb":
    {
      "results": {..}
    }
    ...
  }
}
```

Listing 5.3: metamorph.json output structure.

5.1.11 ResultsDB Plugin Design

All components that are successfully tested by CI system need to be tagged with specific **`release-tierX`** tag. Tagging a build is easy if the use case is common, however, for complex use cases there is no easy solution that would enable users to tag a component build. In this purpose *Metamorph* can help to collate all the results and optionally return boolean for the given NVR (name-version-release format) and test tier. Most of the required data is sent to *resultsDB* as part of *CI_metrics* data collection.

ResultsDB Plugin Use Cases

ResultsDB plugin use cases of this plugin is to provide possibility to tag component builds in complex use case. *ResultsDB* plugin use cases are:

- **Common use case**. Component is tested only in one continuous integration job. At the end of this job, it is simple to check if all tests were successful or not.

- **Complex use case**. There exist multiple continuous integration jobs for one component. At the end of one job, it is impossible to say whether the component build should be tagged or not. It is difficult because at end of CI job, only one job test results are present. The results data need to be stored in a database which can be queried by *Metamorph* based on certain parameters provided by the component metadata.

ResultsDB Plugin Overview

Teams that are using *Jenkins* as their CI system would have to install *Jenkins Multijob Plugin*[4] and then in their JJB configuration add all of their respective jobs to this tier

[4]https://wiki.jenkins-ci.org/display/JENKINS/Multijob+Plugin

multijob. For consistency, job name can be kept as `release-tierX`, where X is 0,1,2,3 or provide list of respective tier jobs. The tier multijob can be configured to be triggered off messagebus or any of its upstream job. *Metamorph* would be executed as a post-build step, which would get all the job names added to `release-tierX` and query *resultsDB*. The data returned from *resultsDB* will be then collated to `metamorph.json` file with tier tag field containing boolean information.

The described process can be seen in Figure 5.4. A whole multijob pipeline starts by a component build, which is represented as `CI_message` information from *Component builder*. After component testing, test results are sent to *resultsDB* storage system. *Metamorph* as a build step is waiting until needed test results are present in the storage. Test data from all jobs are collated in *Metamorph* tool and provided via `metamorph.json`. CI system decides if to tag a component build upon provided results. If all tests are successful, then component build is tagged with `release-tierX` tag.

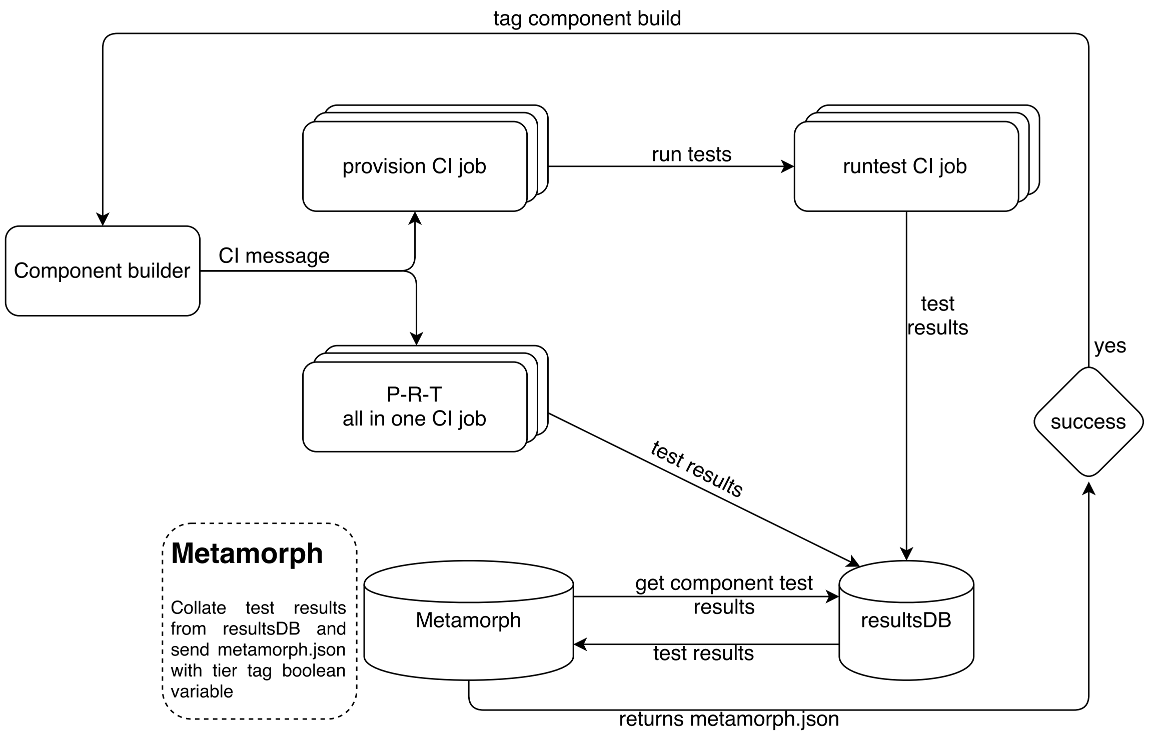


Figure 5.4: ResultsDB plugin design.

ResultsDB Requirements

To make *resultsDB* plugin work, CI jobs must start sending the following data to *resultsDB*:

- job name,
- job build number,
- job build status,
- job build URL,
- component NVR that is being tested,

- test tier number.

Metamorph tool requirements are:

- Need a way to get all runtest job names for the given component and tier.
- Need a way to get all P-R-T (all-in-one) job names for the given component and tier.
- Query *resultsDB* for the Jenkins job build status for the given NVR and test tier.
- Collate the results to return boolean.

ResultsDB Output

An example of a *JSON* output from the *resultsDB* plugin can be seen in Listing 5.4. The main pieces of information in this format are:

- **ci_tier**. The testing tier number.
- **nvr**. Tested component name in `name-version-release` format.
- **tier_tag**. Aggregated boolean value of queried job names. It indicates whether a component build should be tagged or not.
- **job_name**. List of job names results for given component and test tier.

```
{
  "resultsdb": {
    "results": {
      "tier": {
        "ci_tier": int,
        "nvr": string(256),
        "job_name": [
          {
            "jobname": [
              {
                "build_url": anyURI,
                "build_number": int,
                "build_status": string(64)
              }
            ]
          }
        ],
        "tier_tag": boolean
      }
    }
  }
}
```

Listing 5.4: Resultsdb metadata output.

5.1.12 Provision Plugin Design

Every testing environment needs to be successfully provisioned by provision tooling. Some teams have hard coded provision topology files that contain needed metadata for provisioning tool. *Linch-pin* is new provision tooling which enables to provision systems in different environments such as *LibVirt*, *Duffy*, *GCE*, and more. This tooling can be very useful for CI teams. Therefore a new topology file for *linch-pin* is needed. *Metamorph* can help with this task by collating metadata from different storage systems and creating correct topology files for *linch-pin* provisioner. *Provision* plugin is the most important plugin.

Provision Plugin Use Case

The main goal of this plugin is to automatically provide *linch-pin* topology file.

Metamorph will collate data from various storage systems to satisfy needed fields in topology file. The topology file will be sent to *linch-pin* to provision the wanted system. *Metamorph provision* plugin will also provide credentials file for successful provisioning.

Provision Plugin Overview

Teams would need to create configuration files per system in their *git* repositories. A unified configuration format would need to be created which CI teams could easily follow. *Metamorph* tool would expect *git* repository and path to system configuration file in provided repository. *Provision* plugin would then collate data from configuration file and from other storage systems. Topology and credentials file will be the result of the collation process. These two files are necessary for *linch-pin* provisioner.

Figure 5.5 shows above described behavior. The figure also describes testing environment creation. After successful system creation, accessing credentials are sent to following plugin.

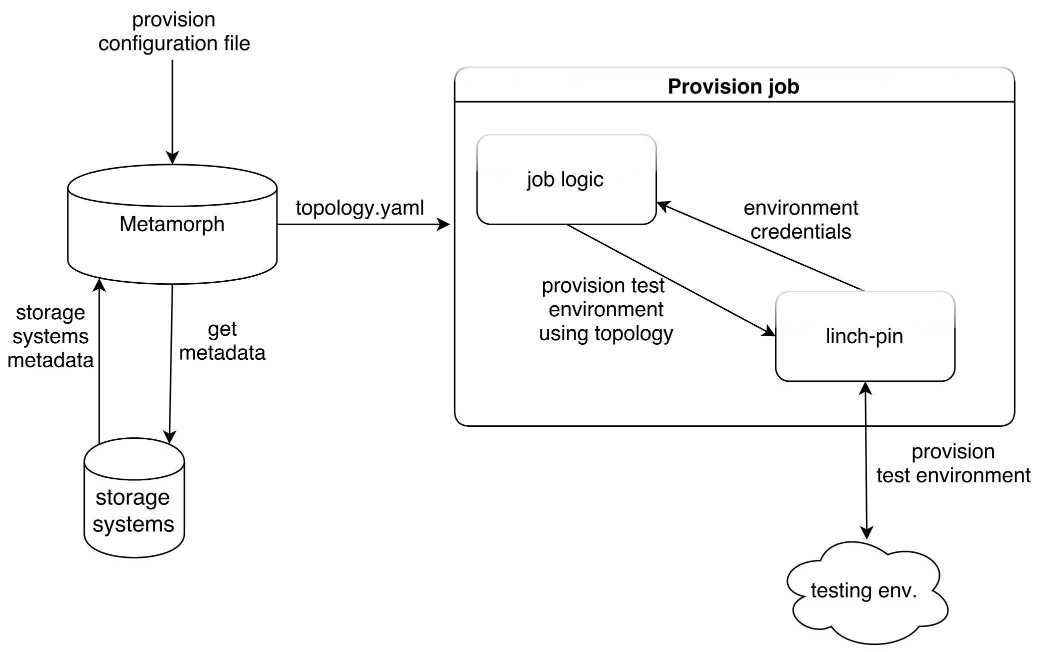


Figure 5.5: Provision plugin design.

Provision Plugin Requirements

For proper working, *provision* plugin needs these data from *git* repository:

- system configuration
- additional metadata which are not present in configurations
- path to credentials file

Metamorph tool requirements are:

- Need a way to clone given repository and create topology yaml according to configuration file.
- Same as above but with added metadata file path and metadata location.
- Query other storage systems to get additional information.
- Collate the results to topology and credentials yaml.

Provision Plugin Output

Topology yaml generated by *provision* plugin is seen in Listing 5.5. The generated output for *linch-pin* provisioner is divided into three sections.

First section contains general information such as *topology name*, *site* and *resource groups*. *Site* field contains information of testing environment configuration file name.

Second section is slightly more important than the previous one. Contains information about name of environment for testing which is stored in `resource_group_name`. Another important field is `assoc_creds` which holds credentials for yaml file name.

The last part contains future system specifications. This is the most important part. Almost all data are collected from system configuration file, but for example `image` must be provided by an external storage system. The `count` value is gathered from additional metadata file.

```yaml
topology_name: "component_name_topo"
site: "site_name"
resource_groups:
 -
  resource_group_name: "resource_name"
  res_group_type: "resource_type"
  res_defs:
   -
    res_name: "component_name_inst"
    flavor: "m1.small"
    res_type: "os_server"
    image: "image_name"
    count: 1
    keypair: "keypair name"
    networks:
     - "list_of_netforks"
  assoc_creds: "credentials.yaml"
```

Listing 5.5: Provision metadata output.

Linch-pin provisioner needs `topology.yaml` and `credentials.yaml` which can be seen in Listing 5.6. This is example credentials output for openstack system. Without this file, *linch-pin* would not have sufficient rights to provision testing environment.

```
endpoint: http://example.com:5000/v2.0/
project: example
username: example
password: example
```

Listing 5.6: Openstack credentials output.

5.1.13 PDC Plugin Design

All plugins need some kind of metadata. As a side note, many continuous integration teams are collecting metadata on their own. Most of the needed metadata is stored in *Product Definition Center*[5]. It is a web application with *Rest API* interface. This tooling is automatically filled with data from existing processes, which enables to develop better tooling without metadata dependency. *Metamorph* can be used as user friendly interface to get needed metadata from the *PDC* tool.

PDC Plugin Use Case

PDC plugin's goal is to provide wide metadata stored in *Product Definition Center*. To do that, *Metamorph* comes with three use cases:

- **Wide metadata use case**. *Product Definition Center* API will be queried to get as big as possible metadata output. This can be used by teams to explore what they can use from *PDC* and how these data will be provided.
- **Single data use case**. *Metamorph PDC* plugin will be queried to get one specific type *Product Definition Center* data. For example, it can be recipients information or components releases.
- **Metadata type use case**. *PDC* plugin can be queried with metadata type such as test job, test provision or test run metadata argument. User can use it by their needs without any deep knowledge of *Product Definition Center* data types. This use case can save a lot of time for plugins that need multiple *Product Definition Center* data types, but not all of them.

PDC Plugin Overview

To support this plugin, every continuous integration job should adjust its behavior to expect `metamorph.json` file with metadata information. Afterwards *PDC* plugin can be used as the first or the last build step in their job behavior. Metamorph plugin will need to create argument mapping to support all use cases. Simultaneously, *Product Definition Center* data types would need to be divided into metadata type groups that could be easily queried afterwards.

Figure 5.6 shows how *PDC* plugin can be used. The figure contains three different continuous integration jobs. *CI job* stands for a general continuous integration job which needs single specific metadata from *PDC* plugin. On the other hand, *Provision* and *Runtest* jobs are executing *Metamorph* tooling to get type specific metadata such as *provision* or *test run* metadata.

[5] https://pdc.fedoraproject.org/

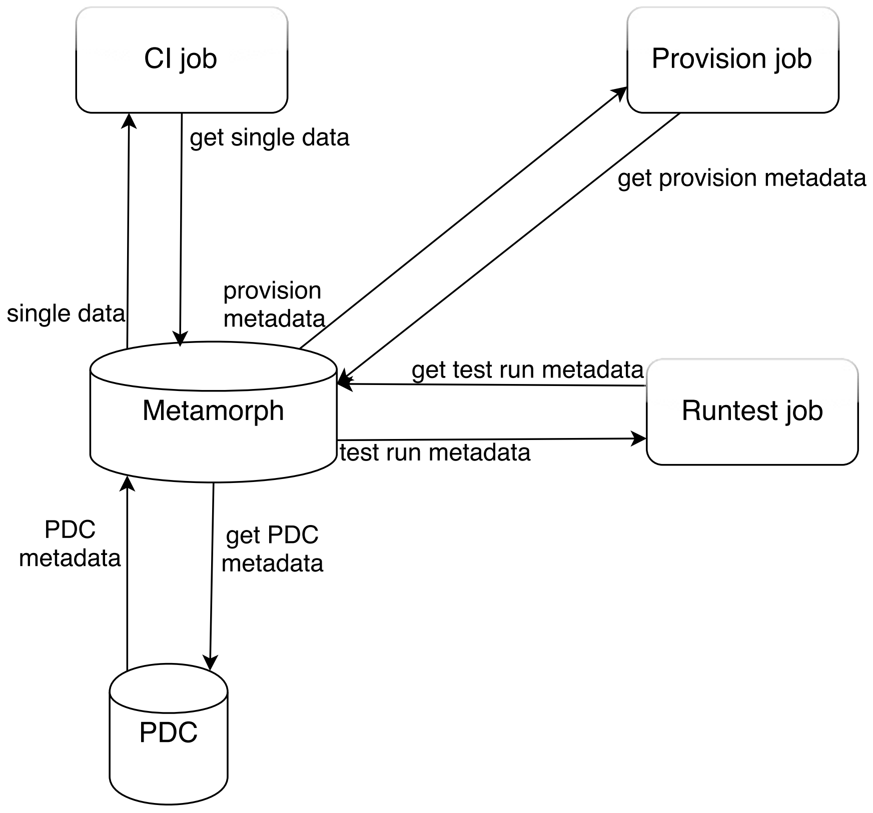


Figure 5.6: PDC plugin design.

PDC Plugin Output

PDC plugin output is in *JSON* format. The output can be seen in Listing 5.7. The example shows all metadata types provided by *Product Definition Center*. As provision type, metadata can be considered `rpms` and `rpm-mapping` where `rpm-mapping` provides information about type of system which needs to be tested (Workstation, Server, ...). `release-component-contacts` can be used as *report* metadata. This type of metadata is often managed by the CI team by themselves and they need to maintain this information up to date.

```
"pdc": {
  "results": {
    "release-component-contacts": [...],
    "release-components": [...],
    "release-component-relationships": [...],
    "global-component-contacts": [...],
    "rpms": [...],
    "global-components": [...],
    "bugzilla-components": [...],
    "rpm-mapping": {...}}}
```

Listing 5.7: PDC metadata output.

5.2 Implementation

This section contains implementation details about the *Metamorph* tool. The first subsection describes the *Ansible* automation tool. The *Ansible* tool is described here because every plugin has also a duplicate version for *Ansible*. The next subsections deeply describe the implementation of *Metamorph* plugins.

5.2.1 Ansible

Ansible is an unique automation tool which supports deployment and orchestration. This tool is mainly used as a remote administrator through the *SSH*. *Ansible* is very easy to run on various platforms because of its system requirements. It can run on Linux servers with the *python2.4* support only. It does not require any other expensive tooling to be installed. The output of this tool is the JSON format, which is the same output format used in the *Metamorph* tool. *Ansible* logic is divided into *Ansible playbooks* and *Ansible modules*, where playbooks are written in the *YAML* format and modules are programming language independent.

Ansible history is quite short. The tool was developed by *Michael DeHaan*, who is the author of *Cobbler*[6] provisioning server and a co-author of *Func* framework for remote administration [4]. Before the *Ansible*, *Linux* admins had to run different tools such as Puppet, Chef, Fabric or Capistrano to automate their configurations and software deployment. The big load of *Linux* admins tool was *Michael DeHaan's* key motive to create a new tooling which would support their needs. An interesting fact is that *Michael DeHaan* was a *Red Hat* employee at the time he was working on *Cobbler* or *Func*, but he left to create the *Ansible* tool. When *Ansible* started to be famous, *Red Hat* bought the *Ansible* and *Michael DeHaan* became a *Red Hat* employee again.

Even though the starting point of the *Ansible* was to make sysadmins' lives easier, the *Ansible* can be used by developers, release engineers, IT managers, or just anyone who wants to manage their environments. This tool can be used for small system setups or for maintaining enterprise environments with hundreds of instances. All *Ansible* communications are using *SSH*. The *SSH* is one of the most used open source components. Therefore security risks are at minimum.

„Ansible" name comes from the *Rocannon's World* book written by *Ursula K. Le Guin* and it stands for a „communication device which allows for a light speed transportation".

Ansible Playbooks

Ansible playbooks can be easily described as configuration files. Playbooks are written in the *YAML* format, which is trying not to be a programming language but more of a configuration or a process. Configuration files were designed to be human readable and intuitive. Playbook's behavior is done by mapping a group of hosts to predefined roles. Roles are represented as tasks in the *Ansible* playbooks. Tasks are simple *Ansible* modules' executions.

A simple *Ansible* playbook can be seen in Listing 5.8 below. The purpose of this playbook is to have a working *Apache* server on *webserver* system. This is done by execution of three *Ansible* modules - *yum*, *template* and *service* - which installs, configures and starts *Apache* server. At the end, *Apache* is restarted by handlers.

[6] http://cobbler.github.io/

```yaml
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running
    service:
      name: httpd
      state: started
  handlers:
   - name: restart apache
     service:
       name: httpd
       state: restarted
```

Listing 5.8: Ansible playbook example.

Ansible playbooks mainly contain these fields:

- **hosts**. Is a list of one or more groups of environment systems divided by columns. *SSH* will connect to a system which is provided in the host field with given variables. Variables examples can be seen in Listing 5.8 in `vars` section. Another important field in `hosts` section is `remote_user`. This field contains the name of the user who will be logged in the remote system.

- **tasks**. Every `hosts` contains a list of tasks. Tasks are executed one after another in all systems matched in hosts field. A next task is executed after the successful finish of the previous one. The main benefit is that on every host, the same amount of tasks will be executed with the same directives. *Ansible* playbooks are running `tasks` from up to down and if one task for a single host fails, this host is taken out from playbook execution. Tasks are generally *Ansible* modules, but it is also possible to run them from *shell* command. On the other hand, *shell* command execution is not very recommended if it is not a simple command like *chmod* or something similar. Every task should also have a `name` field, which would contain short descriptions of task behavior. Tasks variables are listed below task names, such as `name` or `state` in Listing 5.8 in service task section. Tasks also contain a `remote_user` field to execute a special task under a different user who is specified in the `hosts` section.

- **handlers**. *Ansible* modules should have idempotent behavior, but it can occur that they do not have it. For this reason, playbooks have a *Handlers* section, which is a simple event system used mostly as a response to a change. If a module is not idempotent, then `notify` field is needed to specify the particular task. It can be seen in Listing 5.8 in the *template* task. *Handlers* are tasks that are executed after finished `tasks`. They are executed only once, regardless of the number of handlers notifications.

Generally, playbook names are *play.yaml*, and they are executed with `ansible-playbook play.yaml` command [8].

Ansible Modules

Ansible modules are performing actions in the *Ansible* language. Modules can be executed in the command line by `ansible` commands or in playbooks. *Ansible* has a broad base of modules to choose from. Therefore there is no need to create new modules. A new playbook creation from existing ones should solve many problems that could occur. On the other hand, if someone wants to create an Ansible module, it is possible and easy. An *Ansible* module is basically a normal programming language code (Python, Java, etc) with an extra import.

What are the challenges for *Ansible* beginners? New *Ansible* users have common problems with passing arguments to modules. There are two ways to solve this. The first solution was described in the previous subsection - by playbooks. The other way to do it is to pass them as a command line arguments in the following format: `'key=value'`.

```python
#!/usr/bin/python
from ansible.module_utils.basic import AnsibleModule

def can_reach(module, host, port, timeout):
  nc_path = module.get_bin_path('nc', required=True)
  args = [nc_path, "-z", "-w", str(timeout),
  host, str(port)]
  (rc, stdout, stderr) = module.run_command(args)
  return rc == 0

def main():
  module = AnsibleModule(
    argument_spec=dict(
           host=dict(required=True),
           port=dict(required=True, type='int'),
           timeout=dict(required=False, type='int', default=3)),
    supports_check_mode=True)

  host = module.params['host']
  port = module.params['port']
  timeout = module.params['timeout']
  if can_reach(module, host, port, timeout):
    module.exit_json(changed=False)
  else:
    msg = "Could not reach {0}:{1}".format(host, port)
    module.fail_json(msg=msg)

main()
```

Listing 5.9: Python example of Ansible module

In this document, the most important question is how to implement *Ansible* modules in the *Python* programming language. *Metamorph* plugins are developed in *Python*, but every plugin also has a module that provides the same behavior while supporting *Ansible*. An example of such *Python* module can be seen in Listing 5.9. Some interesting parts in the code example are:

- **line 2**. Python import of `AnsibleModule` class. This class needs to be imported in every *Ansible* module written in Python.
- **line 5**. This line shows how easy it is to get an external program path by the method from *AnsibleModule* class.
- **line 8**. An execution of a program that was gathered in line 5 with additional arguments. This line could be normally done by *subprocess.Pyopen* class in pure Python.
- **lines 12 - 17**. These lines contain a module creation. Input arguments are configured by dictionary. Even though it is a new argument configuration, it is quite similar to *argparse* format, which is mostly used to parse input arguments in Python. Hence it is easy to setup input arguments for *Ansible* modules. Line 17 says that this module supports check mode for *handlers* in playbooks.
- **line 19**. Accessing input parameters. In pure Python, input parameters would be passed as *argparse* object variables, but `AnsibleModule` supports them in dictionary type.
- **line 23**. This line shows python „return“ example in *Ansible*. In this case, it is only informing that nothing changes, but it can also send output data.
- **line 26**. An example of failed output with custom message.

Developing *Ansible* modules can be really easy and powerful, because nothing more would be needed to know for a new Python *Ansible* module creation. An *Ansible* module is a key part in *Ansible* language [8].

5.2.2 ResultsDB Plugin Implementation

The design and purpose of this plugin was deeply explained in previous sections. This section is focusing on implementation details of this plugin. *ResultsDB* plugin implementation is in `morph_resultsdb.py` file. Similar code can be found in `resultsdb.py` *Ansible* module.

This plugin requires three arguments: `--resultsdb-api-url`, `--test-tier`, and the name of component in NVR format. The last argument can be passed to plugin in three ways. The simplest way is to use `--nvr`. The second option is to send exported data from CI message to *resultsDB* plugin through `--ci-message` argument. The last way to pass a component name is to use an environmental variable and pass it through `--env-variable` argument. Continuous integration job names or different metadata output can be passed to *resultsDB* plugin as well. An argument configuration is formatted using *argparse*, and it is placed in `parse_args` function.

Class `ResultsDBApi` contains plugin behavior. The key method in this class is `get_test_tier_status_metadata`. This method executes other methods to get metadata from *resultsDB* and to collate the data to provide information about whether the component build should be tagged or not. The method is divided into two halfs. The first half controls metadata extraction and aggregation when CI job names are provided. The second half manages extraction and aggregation as well for cases when CI job names are not provided. The result of this method is a dictionary, where keys are job names and their values are lists of *resultsDB* extracted results for this jobs' names. Other methods in this class are:

- `get_resulstdb_data`. This method was created to manage metadata extraction from *resultsDB*. It was created to support both plugin use cases with or without provided job names. The main core of the method is in *while* loop. Every loop launches `query_resultsdb` method to get *resultsDB* metadata and appends them into output list. If a problem occurs during the metadata extraction method, behavior will be stopped for one minute and then it will attempt to download metadata again. This behavior can repeat several times, but for maximum of two hours.

- `query_resultsdb`. From its name it can be presumed that this method was created for querying *resultsDB*. This method is using *requests* library for querying and it automatically checks query status. If an error occurs, `query_resultsdb` method tries to query *resultsDB* after one minute. If the error does not disappear after three attempts, an exception is raised.

- `format_result`. Another important method in `resultsDBApi` class was created to aggregate results from `get_test_tier_status_metadata`. It checks every job result. If one contains the *FAILED* variable, then `tier_tag` variable is set to *false*. Simultaneously, it formats plugin output. Afterwards, the result of this method is exported to `metadata.json` file.

5.2.3 Provision Plugin Implementation

Provision logic is implemented in `morph_provision.py` as pure Python. This method does not have *Ansible* modules, because it is blocked by extra storage which should provide the name of topology image.

Provision plugin expects two parameters by default. First parameter `--git-repo` requires *git* repository path for further cloning. The `--osp-config` parameter expects path to system configuration file in the provided *git* repository. Information provided by these parameters is needed for *linch-pin* topology creation. On the other hand, some of the information are not present in configuration file, therefore `--metadata-file` and `--metadata-loc` were created for this purpose. `--metadata-file` expects filename which contains additional topology metadata. The biggest struggle here was to find a way to provide metadata location in the provided file. Listing 5.10 solves this problem by new format support which can be seen in *help* part.

```
metadata.add_argument(
    "--metadata-loc",
    action='append',
    type=lambda kv: kv.split("=", 1),
    help='Usage --metadata-loc metadata=path,to,metadata')
```

Listing 5.10: File metadata location parsing

`Provision` class provides the *Provision* plugin logic described in previous sections. The method managing provision metadata extraction is named `get_provision_metadata`. At the beginning, this method clones *git* repository and creates a general topology file. Afterwards, it sets provisioner credentials. Topology file adjusted by specified metadata is managed by this method as well. At the end, this method returns *linch-pin* topology file. `Provision` class contains more methods, such as:

- `clone_git_repository`. The purpose of this method is to clone a given repository. The method behavior starts by repository name extraction from its path. For correct *git* cloning, the *GitPython* library was chosen. This method does not return anything, but after its completion the cloned repository can be found in current working directory.
- `get_metadata_from_location`. Metadata extraction and provision topology adjustment is driven by this method. The first half of this method performs metadata extraction, which is done by `get_metadata_from_location` method. The second half updates provision topology file by the extracted metadata.
- `setup_topology_by_osp_config`. General *linch-pin* topology creation is managed in this method. Topology file is adjusted by extracted metadata from system configuration file.

5.2.4 PDC Plugin Implementation

The purpose of this plugin is to get as much metadata information as possible from the *Product Definition Center*. The implementation of *PDC* plugin is held in `morph_pdc.py` python file and `pdc.py` *Ansible* module.

PDC plugin requires only two information components in NVR format provided by `--component-nvr` parameter, and *Product Definition Center* API url provided by `--pdc-api-url` parameter. In case of any problems with API url certificate, the `--ca-cert` parameter can be used to specify the correct certificate for url verification.

Class `PDCApi` provides this plugin behavior. *Product Definition Center* metadata extraction is managed by `get_pdc_metadata_by_component_name` method. At first, component name, version and release are extracted from the *NVR* format. *Product Definition Center* api options are then updated with the extracted information. The options information are held in an internal dictionary, which can be seen in Listing 5.11. It is really easy to add or support another *PDC* option because of this concept. *Product Definition Center* metadata extraction is driven by *foreach* and *while* loop. The foreach loops over `pdc_name_mapping` dictionary from Listing 5.11 and sets up *PDC* api url with provided options. Adjusted url is then processed in the *while* loop, which is inserted in the *foreach* loop. The second loop manages *Product Definition Center* metadata extraction by api url query and then its appending into output `pdc_metadata` dictionary.

```
pdc_name_mapping = {
    "bugzilla-components": {"name": '{}'},
    "global-components": {"name": '{}'},
    "release-component-contacts": {"component": '^{}$'},
    "release-component-relationship": {"from_component_name": '{}'},
    "release-components": {"name": '{}'},
    "rpms": {"name": '^{}$', "version": '{}', "release": '{}'},
    "global-component-contacts": {"component": '^{}$'}
}
```

Listing 5.11: Product Definition Center options mapping.

The `PDCApi` class contains other supportive methods from which the most interesting are:

- `get_rpm_mappings`. This method manages metadata extraction from `rpm-mapping` *Product Definition Center* option. This option can not be inserted in Listing 5.11 dictionary, because it needs parameters that are present in these options. Therefore these parameters need to be extracted from the queried metadata. The result of this method is an `rpm_mappings` dictionary containing the extracted metadata.

- `get_release_ids`. The purpose of this method is to extract Linux distribution release IDs for given component. The `get_release_ids` extracts component releases from *release-components Product Definition Center* metadata. Extracted releases are then tested with *rpms* metadata and correct release IDs are then returned. This method is very important for `get_rpm_mappings` method.

5.2.5 Additional Metamorph Plugins Implementation

Messagehub Plugin

The *Messagehub* plugin's purpose is to listen to information bus which contains messages about component builds. These messages have metadata important for other plugins. Plugin implementations are held in `morph_messagehub.py` and `messagehub.py` files.

This plugin can be executed in two different ways. The first case is to sniff for messages on information bus. Parameters `--user`, `--password` and `--host` are required for this case. If these parameters are present, `messagebus_run` method is executed to manage message sniffing. Messages are extracted from information bus by configured object from `CIListener` class, which inherits from `stom.ConnectionListener` class. After message extraction, the metadata are then stored in `metamorph.json` file. The second case needs `--env-variable` parameter that expects the name of the environmental variable containing CI message. If this case is chosen, then method `env_run` is executed to manage message extraction from the given environmental variable. The result is stored in `metamorph.json` file.

Message Data Extractor Plugin

The purpose of this plugin is to extract important metadata from CI message. The extracted metadata should satisfy other *Metamorph* tool plugins. That means other plugins can be executed automatically without any internal mapping or other metadata. The implementation of *Message Data Extractor* plugin are in `morph_message_data_extractor.py` and `message_data_extractor.py` files.

Plugin behavior is occupied in `MessageDataExtractor` class, where method `get_ci_message_data manages` CI message metadata extraction. At the beginning, it is checked whether given CI message type is supported. If yes, it is probable that it will contain the needed metadata. The important metadata which will be extracted are component name, component build version, release, target and owner. These metadata are then stored in `metamorph.json` file.

5.3 Metamorph Testing

When developing the *Metamorph* tool, we used the *Test Driven Development* process. This process ensures that almost every method in each *Metamorph* plugin class has its own dedicated test. Therefore if someone changes its behavior, these tests should detect it. The *Metamorph* tool *git* repository is hosted on *Github*[7]. The benefit of *Github* hosting is that it has a very good support with *Travis CI* which was explained in Subsection 2.4.2. This continuous integration tool helps to develop the *Metamorph* tool in the best possible quality.

Metamorph tests can be found in the `tests/metamorph_tests.py` file. This file contains more than thirty tests for all *Metamorph* plugins. *Unittest*[8] was chosen as the testing framework because of its test automation support, aggregation support for various tests, and its independence.

5.3.1 Metamorph Testing by Red Hat Teams

The created *Metamorph* plugins were tested by one Red Hat CI team and several members of the Red Hat development teams. The tool was tested in one meeting where all *Metamorph* plugins were explained. Feedback from the people who tested the tool was collected by a questionnaire with the following questions for each plugin: what are the plugin advantages, disadvantages or how it can be improved. The obtained results are summarized below and the questionnaire can be found in appendices.

ResultsDB Plugin

This plugin was evaluated as a really good part of *Metamorph* and metadata provided by this tool are really valuable for the Red Hat CI team. On the other hand, a bug was found in the *ResultsDB* plugin idea. The problem is that, in the moment of querying, the *resultsDB* plugin does not know whether *resultsDB* contains all testing results or just a few. Therefore some orchestrator would need to be created to provide this information. Anyway, the propped approach still brings a better solution than the Red Hat CI team have, so they will use it, but the bug needs to be fixed in a new *Metamorph* version.

- **pros**. Ability to get test tier results for multiple jobs.
- **cons**. None found except the mentioned bug.
- **improvements**. Possibility to provide advanced *resultsDB* querying and inform users that results are stored in `metamorph.json` file.

PDC Plugin

This plugin was also well received by the testing team. They see a broad usage for this tool to get *test run metadata* or *test report metadata*. On the other hand, old information about component owner was found in *Product Definition Cente*r metadata. Because of this knowledge, an issue was created for *Product Definition Center* to actualize their data and to provide information which metadata are up to date.

- **pros**. Testing team sees *PDC* plugin pros in improving lots of things in CI use case and in gathering valuable information from *Product Definition Center*.
- **cons**. Plugin cons were not found.

[7] https://github.com/RHQE/metamorph
[8] https://docs.python.org/3/library/unittest.html

- **improvements**. Provide better possibility to query *Product Definition Center* and inform that information are stored in `metamorph.json` file.

Provision Plugin

This plugin brings a possibility to use *linch-pin* provisioner and its features. On the other hand, it is out of scope of the testing team because of various changes which would need to be implemented to support this plugin. However, they would like to use it in the future when this plugin will be completed (missing image support).

- **pros**. Easier way to use *linch-pin*.
- **cons**. Testing team cannot use it right now.
- **improvements**. Needs to improve the *PDC* plugin to provide feedback on what it is doing.

Messagehub Plugin

This plugin brought a big discussion about its purpose. The Red Hat CI team is using the *Jenkins* plugin to do a similar job, and at the beginning, they did not see any point of it. After a discussion, they agreed that it could be useful for them, because the *Jenkins* plugin has blackouts and it is difficult to prevent them. On the other hand, if they use the *Messagehub* plugin, they can control the code and then avoid blackouts.

- **pros**. The benefits are filtering and transformation of metadata.
- **cons**. Does not allow to work in daemon mode.
- **improvements**. Improve plugin input.

Metamorph Tool Overall

The testing team rated this tool as a benefit, because they really need something like it. On the other hand, they suggested that the created plugins should be improved and have added support for new metadata storage systems.

- **pros**. This type of tool is needed.
- **cons**. Implemented *Ansible* modules are hard to debug and there is no uniform way how to execute *Metamorph* from the command line.
- **improvements**. *Metamorph* tool could use existing internal tool design to solve future implementation problems.

Chapter 6

Conclusion

The aim of this Master Thesis was to analyze existing continuous integration solutions in Red Hat Czech, then create and test a solution that could support the unification process.

As a part of working on the above goal we studied and presented the technologies used in Red Hat company. We then conducted a research on continuous integration solutions and metadata analysis which the Red Hat teams are using. The results of this research are documented in Chapter 4. They were used as basis for designing the *Metamorph* tool. The designed tool was implemented and tested.

Tables 4.2 and 4.1 show where continuous integration teams store their metadata. Because of this research, a new format for storing metadata was created, as can be seen in Example 4.1. This research brings yet another conclusion, which is a tool that will be able to get metadata from various storage systems when needed. This tool was named *Metamorph*. From the beginning of this tool creation, it was thoroughly discussed with Red Hat managers and principal quality engineers. According to them, *Metamorph* is the key for *Platform CI* teams unification. The implementation process was lengthy, because every *Metamorph* plugin has to have a design document, and the implementation can start only after its approval. Another element which slowed down the whole process was that some of the storage systems were not documented (an extensive analysis of the *Product Definition Center* was needed to know which metadata are relevant for the CI teams) or did not support some options for querying (pull requests were needed to create which enables to query *resultsDB* by job names and ci tier). Because of these reasons, *Metamorph* tool contains only five plugins. On the other hand, every plugin has an *Ansible* module.

On top of that, *Metamorph* tool design was created to understand how *Metamorph* plugins should cooperate and be executed. Unfortunately, this design document is still in development process, but one principal quality engineer said it was exactly what we need for the future of this tool.

The *Test Driven Development* process was chosen as the development process for the *Metamorph* tool. Therefore almost all methods were tested and it brought better quality for implementation. The tool functionality was tested by *Travis CI* after every commit and on internal Red Hat *Jenkins*. Every plugin had to be acknowledged by code review, which brought more readable and easy to use code and there is *Metamorph* documentation is *sphinx* format which describes plugins execution and usage. Furthermore, the *Metamorph* plugins were tested by one CI team and one development team and their feedback was collected by a questionnaire. The results say that some plugins have to improved to support their needs, but the respondents all agreed that *Metamorph* tool is something really needed.

Metamorph tool's advantage is that plugin results can be easily shared between continuous integration jobs. Also, because of created *Ansible* modules, it is possible to run plugins on different hosts, which brings big flexibility to CI use cases.

For the future, the following improvements of *Metamorph* are planned such as easy execution of the created plugins from one place. This step should allow one to spread *Metamorph* into other CI teams. The improved tooling design is already in review process and it will be the next implementation step.

Another key step is to improve *Metamorph* plugins according to the collected feedback and try to modify them to fit the described needs. This approach can motivate engineers to create new plugins by themselves. This would be a key moment for *Metamorph* as an open source tool, because that would result in a community developing new plugins.

In the future, *Metamorph* should support more storage systems, such as *TCMS* or *CDN*, to provide more metadata. On the other hand, better communication with other teams will be needed as well to modify the tool exactly to their needs.

Bibliography

[1] ABS: *Continuous Delivery Pipelines: GoCD vs Jenkins*. June 2014. [Online; visited 28.12.2016].
Retrieved from: https://highops.com/insights/continuous-delivery-pipelines-gocd-vs-jenkins

[2] Basu, S.: *Travis-CI: What, Why, How*. September 2013. [Online; visited 28.12.2016].
Retrieved from:
https://code.tutsplus.com/tutorials/travis-ci-what-why-how--net-34771

[3] Booch, G.: *Objected-Oriented analysis and design*. ADDISION-WESLEY. 1994. ISBN 0-8053-5340-2.

[4] Cloud, C.: *An Interview with Ansible Author Michael DeHaan*. April 2012. [Online; visited 8.5.2017].
Retrieved from: http://www.coloandcloud.com/editorial/an-interview-with-ansible-author-michael-dehaan/

[5] Dougherty, B.: *Bamboo vs. Travis CI vs. Circle CI vs. Codeship*. November 2016. [Online; visited 28.12.2016].
Retrieved from: https://www.itcentralstation.com/product_reviews/travis-ci-review-32073-by-ben-dougherty

[6] Fowler, M.: *Continuous Integration*. September 2000. [Online; visited 26.12.2016].
Retrieved from:
http://www.martinfowler.com/articles/originalContinuousIntegration.html

[7] Fowler, M.: *Deployment Pipeline*. May 2013. [Online; visited 15.05.2017].
Retrieved from: https://martinfowler.com/bliki/DeploymentPipeline.html

[8] Hochstein, L.: *Ansible Up and Running*. Oreilly. 2015. ISBN ISBN 978-1-491-91532-5.

[9] Kawaguchi, K.: *Meet Jenkins*. November 2013. [Online; visited 28.12.2016].
Retrieved from: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins

[10] Neokrates: *Choosing continuous integration (CI) tool. Comparison*. April 2010. [Online; visited 28.12.2016].
Retrieved from: http://www.thinkplexx.com/learn/article/build-chain/ci

[11] Nikola Banovic, F. C. J. M., Tofi Buzali; Dey, A. K.: *Modeling and Understanding Human Routine Behavior*. ACM. 2016. ISBN ISBN 978-1-4503-3362-7.

[12] Novoseltseva, E.: *Top benefits of continuous integration*. December 2015. [Online; visited 26.12.2016].
Retrieved from:
https://apiumtech.com/blog/top-benefits-of-continuous-integration-2/

[13] Ritchie, S.: *TeamCity vs Jenkins: Which is the Better Continuous Integration (CI) Server*. November 2012. [Online; visited 28.12.2016].
Retrieved from: https://www.excella.com/insights/teamcity-vs-jenkins-better-continuous-integration-server

[14] with S. Matyas, P. M. D.; Glover, A.: *Continuous Integration: Improving software quality and reducing risks*. Pearson Education. 2007. ISBN 978-81-317-2291-6.

[15] Singh, I. P.: *What is the best place to store test data for your automated tests?* March 2010. [Online; visited 5.1.2017].
Retrieved from: http://inderpsingh.blogspot.cz/2010/03/what-is-best-place-to-store-test-data.html

[16] Smart, J. F.: *Jenkins: The definite guide*. O'Reilly. 2011. ISBN 978-1-449-30535-2.

[17] Tikhanski, D.: *Jenkins vs. Other Open Source Continuous Integration Servers*. January 2016. [Online; visited 28.12.2016].
Retrieved from: https://www.blazemeter.com/blog/jenkins-vs-other-open-source-continuous-integration-servers

[18] Weiss, M.: *The benefits of continuous integration*. April 2013. [Online; visited 26.12.2016].
Retrieved from:
https://blog.codeship.com/benefits-of-continuous-integration/

Appendices

Appendix A

The Contents of The Included Media

The CD directory structure is:

- *Metamorph*. The *Metamorph* tool source code with documentation.
- *Thesis*. The LaTeX and *PDF* Master Thesis sources.
- *license.txt*. Project license file.

Appendix B

Manual

This chapter describes how to install and execute the *Metamorph* tool. The tool is implemented in the *Python* and *Ansible* language.

- **Requirements**. The *Metamorph* tool source codes can be found in enclosed CD or *Github*[1] repository. The tool requirements are listed in `metamorph\requirements.txt`. The *Metamorph* is implemented in the pure *Python*, therefore support of the *python 2.7* or higher is required.

- **installation**. Installation of the *Metamorph* is really easy. It is installed by `python setup.py install` command, where `setup.py` file can be found in *Metamorph* repository.

- **Execution**. Execution of *Metamorph* plugins is described in `index.rst` file in `metamorph\docs` repository. The file contain guidlines for easy *Metamorph* plugin execution.

[1] https://github.com/RHQE/metamorph

Appendix C

Questionnaire Results

C.1 ResultsDB Plugin

C.1.1 resultsDB pros

- good start, but api need to be defined
- For our automation tool we will be using ResultsDB plugin to query tier1 test status where multiple jobs provide results.

C.1.2 resultsDB cons

- none
- none found

C.1.3 What can be improved in resultsDB plugin

- We would need advanced querying of resultsDB, which does not provide such an API by itself.
- When ran from the command line, it might be nice to have indication that some output is being written to metamorph.json

C.2 PDC Plugin

C.2.1 PDC pros

- Using PDC will improve a lot of things in CI, I hope this plugin will be usable for CI use case
- The PDC plugin of metamorph provides a convenient way how to query Product Definition Center and gather valuable information for tested package used in testing.

C.2.2 PDC cons

- none
- good start, but api need to be defined

C.2.3 What can be improved in PDC plugin

- api to get value / list of some variables, not only json
- It would be advised to have more option for querying.

C.3 Provision Plugin

C.3.1 Provision pros

- easier to use linchpin
- The provision plugin makes it possible to directly generate LinchPin Provisioner configuration files according to queried data.

C.3.2 Provision cons

- none
- no use for our CI now

C.3.3 What can be improved in Provision plugin

- When ran from the command line, it might be nice to include some feedback as to what is going on...as it is, I have no idea what was performed (without reading the source code).
- Support for all systems provided by linch-pin

C.4 Messagehub Plugin

C.4.1 Messagehub pros

- filtering, transformation of metadata
- Filter how many messages it will download

C.4.2 Messagehub cons

- strange input, I would prefer message, not listening on message bus itself
- Missing daemon use case

C.4.3 What can be improved in Messagehub plugin

- type of input
- Add daemon use case. Afterwards this plugin can be used in our CI.

C.5 Metamorph Tool

C.5.1 Metamorph tool pros

- We need to have such tool because of too many different sources/format of metadata
- Provided metadata in unified format.

C.5.2 Metamorph tool cons

- ansible modules - hard to debug; json for sharing data between modules?
- There is currently not easy/uniform way how to run the metamorph tool from command line.

C.5.3 What can be improved in Metamorph tool

- rewrite it to use internal tool design for better debugging, logging and integration with CI
- The metamorph could reuse the existing internal tool design frame