D5.1 Continuous Integration and Testing Approach

Project Acronym	SONATA
Project Title	Service Programming and Orchestration for Virtualised Software Networks
Project Number	671517 (co-funded by the European Commission through Horizon 2020)
Instrument	Collaborative Innovation Action
Start Date	01/07/2015
Duration	30 months
Thematic Priority	ICT-14-2014 Advanced 5G Network Infrastructure for the Future Internet

Deliverable	D5.1 Continuous Integration and Testing Approach
Work Package	WP5 Integration, System Testing and Qualification
Due Date	M5
Submission Date	07/12/2015
Version	1.0
Status	Final
Editors	Sharon Mendel-Brin (ALU)
Contributors	Burak Karaboga (Atos), George Xilouris (NCSRD), James Ahtes (Atos), Jose Bonnet (PT), Michael Bredel (NEC), Tiago Batista (Ubiwhere)
Reviewer(s)	Alex Galis (UCL), Shuaib Siddiqui (i2CAT)

Deliverable Type		
R	Document	**X**
DEM	Demonstrator, pilot, prototype	
DEC	Websites, patent filings, videos, etc.	
OTHER		
Dissemination Level		
PU	Public	**X**
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary

WP5 focuses on the integration and qualification of WP3 and WP4 outcomes. The first activity, performed during Months 4 and 5 (M4, M5) of the work plan, is creating the continuous integration and delivery (CI/CD) methodology, followed by defining the automation software, tools, roles and responsibilities to facilitate it. The availability and the maintenance of the infrastructure that will be used to support this methodology is provided by WP6.

The defined development, integration and delivery flow was inspired by common best practices for agile development within large, distributed teams.

The deliverable defines several aspects of the CI/CD methodology, such as branching deployment, relevant tools, style checks, code reviews and various testing levels to support quality assurance.

To support the evaluation of tools a set of criteria was defined, resulting in the following selections:

- Github was selected as a revision control system
- Jenkins as a continuous integration system
- Github as a general issue and bug tracking system.

Additional tools to support the development and integration process are being further evaluated and integrated into the workflow.

Future activities in WP5 will be focused on the integration of WP3 and WP4 outcomes and acceptance testing.

WP6 is responsible on the infrastructure setup, validation and pilots. During the first five months of SONATA the work of WP6 focused on the definition of the three infrastructure variants - integration, qualification and demonstration infrastructures. Moreover, WP6 deploys and maintains the integration infrastructure for the support of the initial development plans of SONATA. The aforementioned infrastructures will be further elaborated in D6.1 (Definition of the Pilots, Infrastructure Setup and maintenance report) due in M12.

As WP5 relies on the infrastructure provided by WP6, this deliverable also contains a description of the dependency and moreover an overview of what WP6 has prepared through M5.

The primary audience is the same distributed developer teams within the project, for which the deliverable is extended and kept in a “live” status on the SONATA wiki, in the form of consolidated workflow guides, tutorials and tool-specific documentation: http://wiki.sonata-nfv.eu/index.php/Developer_Hub.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This document is a practical guide summarising the CI/CD methodology defined for SONATA as part of the work of WP5 during M4 and M5 of the project.

The audience of the deliverable, particularly the latter sections on the adopted tools and workflow, is in fact the development teams of the project distributed between partners. This methodology will be used by WP3 and WP4 while working on software deliverables in order to ensure continuous delivery of high quality, integrated outcomes.

This document includes the specifications of a continuous integration system, continuous delivery system and quality assurance supporting systems. Those systems are defined by WP5, and will be deployed in the next few months by WP6.

Altogether, the methodology defined in this document specifies the overall flow for developing, deploying and testing the software components to be delivered as part of SONATA.

It also includes an overview of the infrastructures provided by WP6 for SONATA's development and testing, namely integration and qualification environments.

1.2 Organisation

The document is structured as follows:

- Chapter 2 provides an overview of best practices and tools for CI/CD; a study that has been done within WP5 in the first months of the project.
- Chapter 3 then uses the aforementioned study to adopt a workflow and appropriate tools for SONATA's specific development goals and CI/CD needs. This section will also be adapted to the wiki as a living document for the project's development teams.
- Chapter 4 describes the three infrastructure variants of WP6, including Integration, Qualification and Demonstration. The priority is the Integration infrastructure, which has been defined and scheduled to deploy in December 2015.
- Chapter 5 concludes with next steps to follow in WP5 and WP6.

1.3 List of Acronyms

Table 1: List of Acronyms

Acronym	Definition
CD	Continuous Delivery
CI	Continuous Integration
CLI	Command Line Interface
CM	Configuration Management
CPU	Central Processing Unit
IDE	Integrated Development Environments
Mx	Month # of the project work plan (e.g. M2)
NFV	Network Function Virtualisation
NFVI	Network Function Virtualisation Infrastructure
OS	Operating System
PoP	Point of Presence
QA	Quality Assurance
RAM	Random Access Memory
VCS	Version Control System
VPN	Virtual Private Network
WPx	Work Package # (e.g. WP5)

2 Best Practices in Continuous Integration and Delivery (CI/CD)

This section describes the requirements and building blocks of SONATA's continuous integration, continuous delivery and quality assurance.

2.1 Introduction to Continuous Integration and Delivery (CI/CD)

Continuous Integration (CI) and **Continuous Delivery** (CD) supplement each other. The former is the practice, in software engineering, that requires developers to integrate all working copies into a shared repository several times a day [1]. The latter, CD, is an approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time. It aims at, building, testing, and releasing software, faster and more frequently [2].

A continuous integration system should include the following characteristics [3]:

- Maintain a single source repository
- Automate the build
- Make the build self-testing
- Everyone commits to the mainline every day
- Every commit should build the mainline on an integration machine
- Fix broken builds immediately
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what is happening
- Automate deployment

The above list has inspired the development and delivery flow specification for SONATA, detailed in the next sections.

To achieve the required quality, it is necessary to test different aspects throughout the software development process:

- **Unit tests** [4] focus on the smallest part of software design, like functions or object behavior. They identify issues such as incorrect or inconsistent type casting; incorrect initialisation of variables; incorrect or unexpected behavior of code; etc.
- **Integration tests** [5] combine several software components, such as libraries and modules, and test them as a whole. They ensure that the complete system (or at least parts of it) works as expected. Evidently, this is not necessarily the case, even if each of the components has passed all of their unit tests.
- **System tests** [6] focus on the user's or customer's perspective, checking whether the overall system meets the given specification.

Within SONATA there will be several sub-projects, such as the Service Platform and SDK. These sub-projects may differ in types of programming languages or development tools like Integrated Development Environments (IDE). To this end, the workflow should not put any constraints on the development environment or tools. However, all projects should agree on one (Distributed) Version Control System.

2.2 Version Control

2.2.1 Overview

Version control is a system that performs the following tasks [7], recording:

1. changes in a file, or set of files,
2. who made them
3. when they were made
4. why they were made

Stored on a central or distributed repository, a Version Control System (VCS) allows the following functionality [8]:

1. to revert a file, set of files, or even the entire project to a specific state
2. compare versions
3. track changes that are causing problems
4. if some issue related to file loss appears, the VCS can recover it

VCS are designed for collaborating teams where many developers can change the project and some level of revision is required.

A source code oriented VCS is quite simple at a conceptual level. It should simply allow a developer to save the history of changes performed to a set of source files.

The modern VCS systems that are in use by the open source community all belong to the second or third generation. The first generation found on local systems is not suited for collaborative work. The second generation enabled team collaboration on a project, and centralised the authoritative source of code as well as the project history. The third generation broke with the traditional client-server model and started allowing offline development on a distributed model, where each developer has access to the entire project history, and can perform local changes and easily choose which changes to publish. Second generation systems were the precursors of some of the branching methodologies still used today, where each branch or tag was simply a copy of the whole project on a different directory. While most tools started including support for what was called a standard repository layout, it became apparent that the branching capacity should be at the core of the VCS. Other limitations like the requirement to be online to commit code led to another set of architecture requirements. The current trend was led by systems like BitKeeper [9], which offered branch management capabilities allied to the possibility of working offline.

Currently the most used VCS of each generation are Subversion [10] and Git [11]. Others exist such as CVS [12], Darcs [13], Bazaar [14] and Mercurial [15]. However, their advantages are few and in some cases, community support for these tools has begun to fade.

Hence, the question for a new project is reduced mostly to a centralised versus a distributed system, a second or third generation system, where the choice of implementations is steered mostly by the user community.

2.2.2 Relevant Version Control Tools

2.2.2.1 Subversion

Subversion [10] is the most modern of the second generation VCS tools. Still in use on many companies, it has been tested to exhaustion and has few bugs. It has, however, the disadvantages that characterise every second generation VCS: it is centralised in nature, and its branching ability is described by many developers as a difficult to manage. Some larger projects like OpenOffice still use Subversion. Public Subversion hosting is possible on Sourceforge.

2.2.2.2 Git

Git [11] is currently the preferred VCS of open source communities. It was developed from day one with the intent of being the VCS system used by the Linux kernel development. Git has very good branch and merge support, to a point where most projects that are using it rely on the feature for their standard workflow. The development of the tool is still quite active, and the community of users keeps growing. There are quite a few large projects using Git for their day to day development work, among them the Linux kernel, Samba, and Gnome. The biggest hosts of Git projects for the open source community are GitHub and Bitbucket.

2.2.3 Branching Methodology

2.2.3.1 Branch

During the development process, branching the source code is a very useful practice. It allows teams of developers to cooperate on the same code base without conflicts, as well as the support of several releases concurrently. Without branching, an unfinished feature would easily end up on a public release, and applying bug fixes to previous releases would be very difficult to manage.

The motives that lead to the creation of a branch can be varied. The most common branches are created for release maintenance or feature development, however it is not uncommon to see personal branches for a developer or special branches to test compatibility with upstream libraries or other dependencies. [16]

Types of branches include:

- **Release branches** are created when a new version of the software is released. Each one contains the history of the development up to the moment of the release. When a bug that affects the released software is found, its fix can be applied both to the main project and to the affected releases, allowing the maintainer to perform corrective releases from multiple supported branches.

- **Feature branches** are created to give a single developer or a team of developers the ability to create a new complex feature in multiple commits without affecting the release. By using a branch, any temporary breakage or incomplete functionality is kept from impacting the work of the remaining developers, or from released code.

- **Task branches** are the most granular of all the branching patterns. Using this methodology, the development of each task is performed on a separate branch. This

allows the developer to iterate the solution several times before contributing it to the main branches.

2.2.3.2 Merge

The contribution of code from one branch to the other is a merge, where all the changes applied to a branch are applied on top of the history of another branch. This task can be daunting times, as the merge between two branches that have diverged can sometimes lead to complex conflicts that must be solved.

2.2.3.3 Fork

Modern versioning systems have introduced a new capacity called forking. Forking a project works in a very similar way to creating a branch, it has all the advantages that the creation of a branch on a repository has, plus the added advantage that the contributing developer or team does not need write access to the central repository. A mix of branching and forking is common in many projects, where release branches and feature branches are used, but all code contributions come from forked repositories. This model facilitates the contribution of external developers that no longer need to send patches to a mailing list for review [17].

2.2.3.4 Code Quality and Branching

The branching (and forking) process allows the VCS to automatically invoke testing processes before accepting code contributions. This increases the quality of the contributed code, as it allows the early discovery and resolution of problems that might be overlooked because a step was forgotten during a manual process.

2.3 Quality Assurance

The software development methodology for creating the SONATA platform is inspired by the agile principals [18], which are:

1. Customer satisfaction by early and continuous delivery of useful software
2. Welcome changing requirements, even in late development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (a difficulty at times for a collaborative project)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity, the art of maximising the amount of work not done, is essential
11. Self-organising teams
12. Regular adaptation to changing circumstances

Software quality and frequent, usable deliverables are at the heart of this approach. This section covers the steps to be followed by SONATA's development team members to deliver high quality software.

2.3.1 Characteristics of Required Tests

2.3.1.1 Automation

In SONATA, development will follow an approach based on agile methodology [19], hence automated tests are a requirement. This is due to the "speed" that changes are introduced in the system that uses an agile approach, all of which must be validated. Manual tests would simply make that speed impossible to achieve, not to mention the costs of manually repeating the tests. Nevertheless, there are tests that cannot be automated, or may be with higher costs then making them manually. An example of these is usability tests.

2.3.1.2 Complexity

Since tests are in fact code that tests other code, they must be kept very simple. There is hardly a benefit to test the tests themselves, when time and costs are considered.

2.3.1.3 Unit Tests

Unit tests are usually automated and executed by developers, as opposed to the ones executed by a Quality Assurance (QA) team. [20]

Well-designed unit tests typically follow a four phase sequence:

1. **Setup** - prepare/initiate the object(s) to test into a known state.
2. **Exercise** - the (set of) action(s) to test.
3. **Verify** - compare the results of the (set of) action(s) with what was expected.
4. **Tear-down** - clean the changes made both in the setup and the exercise phases, so that other tests can be executed without impact in the order of the test execution.

Unit tests (and any other kinds of tests) designed by the above four phases are easier to maintain, follow and debug.

Other important characteristics of unit test include:

- **Irrelevant order of execution** - A good unit test suite has tests that can be executed in whichever order. This is important because the execution of the test may leave traces that would not otherwise be noticed.

- **Quick to execute** - Since unit tests are supposed to be executed very often, it is mandatory that they execute quickly. There are a number of strategies that can be used to achieve high speed of execution:

    - **Mock** - a simulated (fake) object with which the unit to be tested interacts to verify its **behaviour** [21], that is, when one cannot, from observing the state, infer if the test would be ok or not (e.g. in a ''caching'' system, when a hit or miss cannot be inferred from the data present in the cache).

    - **Stub** - a simulated (fake) object with which the unit to be tested interacts to verify its **state** [21]. Stubs are also called **spies** when they can additionally verify behaviour.

    - **Parallelisation** - execute the whole suite of tests in parallel, taking advantage of the irrelevancy of the order of execution of tests.

    - **Minimisation of the number of tests to execute** - Often IDEs and Frameworks can keep a matrix of which tests are affected by which code, thus allowing only

those tests to be executed, instead of the whole suite. This minimisation leads to shorter test execution times.

Mocks and stubs are part of what are more generically called **Doubles** (or **Fakes**) as the real object would. This is particularly useful when the object to be 'doubled' is not available at the time of testing (e.g. an API) or has high costs of usage (e.g. time to access a database, a remote service, etc.). While literature has not been clear to define and distinguish these, the main point is that whenever the 'unit' one that wants to test depends on other units, there are advantages in 'mocking'/'stubbing' those dependencies and making unit tests fast(er).

- **Run after every save** - If unit tests run quickly, they can run whenever a change in the code is saved. It is fairly easy to configure and use such a mechanism with whichever code editor or IDE being used.

2.3.1.4 Integration Tests

These tests are written to prove that a certain (change in a) 'unit' (see unit tests above) integrates correctly with the other 'units' of the system. Often all existing Integration Tests are run, and not just the ones referring to the specific change, in order to prove that the change has not affected the other interactions with its eventual side effects. In this case, they are better referred to as Regression Tests.

Order of execution - It is often more difficult to make Integration Tests independent of the order of test execution (in comparison to unit tests). But nevertheless, it is a good practice to try, since lots of bugs may hide behind assumptions made under that order of execution.

Run when - The sooner the change is tested against its dependencies the better; unit tests usually eliminate these dependencies by mocking or stubbing them.

It is thus recommended that as soon as the change is 'stable', these tests are written and start being executed in the final development cycles before merging the code to the main branch (see Section 2.2.3). At this time, due to the distributed nature of development, usually only the Integration Tests related to the change introduced by the developer are run (although the whole suite of Integration Tests is available in the code base).

Speed of execution - Since these tests imply using all dependencies (again, no mocking or stubbing), they're expected to execute slower than unit tests. Nevertheless, every strategy that can make them faster, like parallelising the execution of tests, should be used.

Scope - Integration Tests are often too broad (i.e. testing too much at once), which makes them slower to execute, harder to make independent of the order of execution, harder to maintain, and in general waste precious time.

The best Integration Tests are focused on the change they want to prove integrates well with the remaining system under test.

2.3.1.5 System Tests

These tests are written to prove that the system under test does what it is supposed to do and can be accepted by the 'customer'. These are end-to-end tests, that look at the system under test as a black-box: providing inputs and comparing the outputs with what is expected.

System Tests are executed by the 'customer'. In SONATA, this can refer to the service developers within the project's pilots.

Run when - Depending on the organisation(s) involved, these tests can be executed by independent teams, often the more traditional QA team. They can even be manual, when the 'customer' is not 'a developer', when the tests are to be executed rarely (e.g. once per new 'public' version) or when tests involve aspects that are hard to automate (see Section 2.3.1.1).

Scope - The scope of System Tests and Acceptance Tests are the broadest possible.

2.3.2 Style Checks

In order to further increase the code quality, to simplify an onboarding of new developers, and to help programmers (and reviewers) to read and understand source code, SONATA adopts best practice coding conventions. To this end, SONATA implements a programming style, i.e. a set of rules and guidelines that should be followed throughout the software development.

To enforce the usage of the correct coding style, SONATA WP5 implements an automatic style check for every pull request to the SONATA repository. To this end, SONATA could leverage tools, which will help programmers to write code that adheres to a coding standard.

2.3.2.1 Relevant Tools

2.3.2.1.1 Checkstyle

Checkstyle [22] is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare developers of the task. It can find class design problems and method design problems. It also has the ability to check code layout and formatting issues.

2.3.2.1.2 SonarQube

SonarQube [23] is an open platform to manage code quality. It covers the 7 axes of code quality, including style checks. It covers more than 20 programming languages through plugins, including Java and Python. It has a powerful extension mechanism that enables adding rules engines and computing advanced metrics.

2.3.3 Tests Coverage

Code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. The assumption is that a program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.

SONATA will automatically calculate the code coverage for all its software modules.

2.3.3.1 Relevant Tools

2.3.3.1.1 Cobertura

Cobertura [24] is a free Java tool that calculates the percentage of code accessed by tests. It can be used to identify which parts of the Java program are lacking test coverage. It is based on jcoverage.

2.3.3.1.2 SonarQube

As mentioned in Section 2.3.2.1.2, SonarQube is an open platform to manage code quality. It covers the 7 axes of code quality, including Tests Coverage.

2.4 Issue and Bug Tracking

A bug tracking system or defect tracking system is a software application that keeps track of reported software bugs in software development projects. Such a system is usually a necessary component of a good software development infrastructure, and its use is a requirement for a modern, collaborative development team. [25] An issue tracking system is similar to a "bug tracker" and some bug trackers are capable of being used as an issue tracking system, and vice versa. [26]

The bug/issue tracking systems serve as a central repository for monitoring the progress of bug/issue reports, requesting additional information from reporters, and discussing potential solutions for fixing the bug or discussing about the progress of the issue. [27] These systems basically record facts about the issues and/or bugs such as its severity, which part of the software the bug or the issue belongs, details about the erroneous behaviour or work to be done, status, reporter, assignee, etc. The provided information is used by the team members to identify the cause of the defect, and narrow down plausible files that need fixing or improving. [27] The information can also be used to set milestones.

Currently there are plenty of good bug and issue tracking tools in the market to choose from and thus a good analysis of the project at hand is required seek the most appropriate tool. Some tools provide better integration with other systems, offer a rich functionality to be able to record very detailed bugs and issues, and flexible and suitable for complex workflows. The level of complexity can be sometimes excessive, unnecessary for small to medium scale projects/teams and consequently reduce productivity. On the other hand, if the tool is too simple for the needs of the project then it will most probably also be a slowing factor.

2.4.1 Relevant Tools

2.4.1.1 GitHub Issues

GitHub's tracker is called Issues [28], and has its own area in every GitHub repository. The tracker can be considered as one of the most light-weight trackers on the market. Issues provides a flexible UI even though it is offering a limited amount of features and is very open-source friendly. A unique "selling point" is of course its integration with the core GitHub offering, ideal for communities using the repository.

2.4.1.2 Redmine

Redmine is a free and open source, web-based project management and issue tracking tool. It allows users to manage multiple projects and associated subprojects. It features per project wikis and forums, time tracking, and flexible role based access control. It includes a calendar and Gantt charts to aid visual representation of projects and their deadlines. Redmine integrates with various version control systems and includes a repository browser and diff viewer. [29]

2.4.1.3 Mantis

Mantis Bug Tracker is a free and open source, web-based bug tracking system. The most common use of MantisBT is to track software defects. However, MantisBT is often configured by users to serve as a more generic issue tracking system and project management tool. MantisBT also has a plug-in system which allows extension of the tool through both officially maintained and third party plug-ins. As of November 2013, there are over 50 plug-ins available on the MantisBT-plugins organisation on GitHub. [30]

2.4.1.4 JIRA

JIRA is a commercial issue tracking product that can be licensed for running on-premises or available as a hosted application developed by Atlassian. Pricing depends on the maximum number of users. It provides bug tracking, issue tracking and project management functions. Compared to Bugzilla, JIRA is more internally customisable. Most of the bug trackers have a single, fixed, state machine to represent the lifecycle of a 'bug'. JIRA allows this state machine to be changed by its users, and also for different classes of issue to be tracked, each with their own state machine. This makes JIRA an "issue tracker" rather than solely a bug tracker. [31]

2.4.1.5 Launchpad

Launchpad is an open-source, software collaboration platform that provides: bug tracking, code hosting and code reviews. Launchpad provides cross-project and cross-platform bug linking which synchronises the status of the linked bugs and reflect them on the main bug. Launchpad also provides a RESTful API to modify data in Launchpad, allowing the user to report, access, and manage bugs, and also provides the ability to authenticate users' apps to Launchpad. [32]

2.5 Continuous Integration Software

Continuous Integration software facilitates the CI process. It can also support the qualification if used to run artefacts defined and deployed for software projects into both development and integration environments.

2.5.1 Relevant Tools

2.5.1.1 Jenkins

Jenkins is an open source MIT licensed continuous integration software. It is a server-based system running in a servlet container such as Apache Tomcat [33]. Jenkins is widely supported by community and a wide variety of plugins have also being developed for Jenkins.

It is free and open source, and relatively simple to install and use. Its success and adoption mean the availability of extensive documentation and support from an active community. Also relevant is Jenkins' support of Docker plug-ins. [34]

Jenkins requires local infrastructure for deployment, as opposed to a hosted service model like GitHub.

2.5.1.2 Travis CI

Travis provides CI as-a-Service [35] which is widely used by open source projects. It features close integration with GitHub. For example, Travis registers every push to GitHub and automatically builds the branch by default. Support for internal, local setups is limited, and relies heavily on GitHub communities.

It supports a variety of languages, including C, C++, Clojure, Erlang, Go, Groovy, Haskell, Java, JavaScript, Perl, PHP, Python, Ruby and Scala.

Travis uses a freemium model that provides its service free for public repositories on GitHub. However, for private repositories, there is an added pricing model.

2.6 Configuration Management

Configuration Management (CM) is a practice designed to eliminate error and inconsistency by managing system changes made during development, production, and subsequent modification. Recently, Configuration Management tools' popularity has increased due to the proliferation of DevOps in the IT industry. Provision environments, deployment of applications and infrastructure maintenance are critical and complex tasks that traditionally were done manually. In the virtualisation and NFV era the above tools play a serious role as facilitator of the automation within the DevOps cycle [36]. The main benefits of implementing CM are [37]:

- Greater system reliability and faster detection of bottlenecks in performance
- A clear roadmap for disaster recovery
- More accurate budgeting and planning, empowered by a better understanding of Configuration Item components
- The ability to identify causal relationships and maintain version-specific data
- More consistent regulatory compliance and license management

SONATA will employ CM tools in order to support the agile development and deployment of SONATA components or at a later stage to integrate this methodology for the support of DevOps within the SONATA Service Platform and SDK ecosystem. In this view, CM will be employed to maintain integration and qualification environments with concrete configurations and to automatically deploy the software 'packages' generated from WP3/4 components

This section provides an introduction to CM and a quick overview of open source CM tools, in an attempt to assess the current status, features and differences. As the CM tools impose difficulties in mastering them, with some of them requiring a steep learning curve, SONATA will follow an incremental approach. The project will first take into account the current experience of the consortium members involved in this task, and then experiment with the selected candidates.

2.6.1 Introduction to CM Systems

Most state-of-the-art CM systems are built around four fundamental disciplines [37]:

1. **Configuration Identification**: a CM system will create a map and configuration items within the infrastructure, including dependencies between items and their larger IT environment. These items are stored (usually) in a CM database, however the approach varies in cloud-based or dispersed infrastructures.

2. **Configuration Control**: This discipline relies on configured baselines used to monitor an environment shift or fluctuation from a desired state for a particular component. In a case where the change is intentional, the CM system may be able to run an impact analysis of how the environment as a whole has been affected.

3. **Configuration Status Accounting**: This discipline handles accounting for different versions of a software or hardware item that may be in operation or development, as well as any performed or proposed changes to those items. Foremost aspect of the status accounting is keeping accurate documentation of all changes, deviations, and versions to the configuration.

4. **Configuration Verification and Audit**: The discipline enables for the assessment hardware and software components to determine compliance with performance standards and previously determined baselines.

The aforementioned disciplines, allow a set of features supported by the current solutions. A non-exhaustive list of these features is provided in the list below:

- Discovery automation
- Dependency mapping
- Configuration management databases (CMDBs)
- Change management
- Impact analysis
- Asset/inventory management
- Baselines
- Incident/Fault Reporting

2.6.2 Relevant Tools

The five most prominent open source CM systems currently available are listed below. The key point to remember is that no single tool can perform CM effectively on its own. Some of the tools may be complementary and work well together, while others excel in specific areas and/or use cases. Any chosen solution should form part of a comprehensive DevOps toolchain built with specific tools in alignment. [38]

2.6.2.1 CFEngine

CFEngine has been automating mission-critical production environments for more than two decades. It runs on C (as opposed to Puppet's use of Ruby) with architecture that allows easy scaling; it runs fast and has very few dependencies. Its community edition is licensed under the GNU General Public License GPL.

One of the main complaints regarding CFEngine is that the learning curve is very steep. [38] [39]

2.6.2.2 Puppet

Puppet open source is a declarative, model-based configuration management solution that is based on the Puppet language. It is licensed under the Apache 2.0. Its learning curve is less imposing due to Puppet being primarily model driven, understanding JSON data structures in Puppet manifests is preferable to a system admin is more comfortable at the command line. [38] [40]

2.6.2.3 Chef

Chef, like Puppet, is also written in Ruby, and its CLI also uses a Ruby-based DSL. Its open source edition is licensed under Apache 2.0. Chef operates as a master-client model, with a separate workstation needed to control the master. The agents can be installed from the workstation using the 'knife' tool that uses SSH for deployment, easing the installation burden. [38] [41]

2.6.2.4 Ansible

Ansible is one of the more simple solutions for configuration management available. It is designed to be minimal in nature, consistent, secure, and highly reliable, with an extremely low learning curve for administrators, developers, and IT managers. It has an agentless architecture and manages nodes over SSH or PowerShell and requires Python (2.4 or later) to be installed. Modules work over JSON and standard output and can be written in any programming language. The system uses YAML to express reusable descriptions of systems. Its open source edition is licensed under GNU General Public License.

Despite being a relatively new player in the arena when compared to competitors like Chef or Puppet, it has gained quite a favorable reputation amongst DevOps professionals for its straightforward operations and simple management capabilities. [38] [42]

2.6.2.5 SaltStack

SaltStack, like Ansible, is developed in Python. It was developed as an alternative to Puppet and Chef, in particular due to their relatively slow speed of deployment and restricting users to Ruby. Salt is a compromise between Puppet and Ansible. It supports Python, but also requires that the users write all CLI commands in either Python, or the custom DSL called PyDSL. It uses a master server and deployed agents called minions to control and communicate with the target servers, via implementation of ZeroMq messaging library at the transport layer. The result is quicker than Puppet or Chef. Its open source edition is licensed under the Apache 2.0. [38] [43]

2.7 Open Source Development

An important part of SONATA's work plan is to have its results shared with the community as open source. In anticipation of that release, the project has conducted a comparison study on best practices for open source development and community support.

The moment at which a project is delivered to the wider community as a set of code or executables is of importance to the successful uptake of its software. However, large projects developed in a consortium have a very different set of uptake goals when compared to small hobbyist projects. The dimension and complexity of the code to be released can have a strong influence over the way the general public perceives a project, and that influence can be manipulated via the moment that the project code is released to the public.

If the code and respective development process is public from day one, the wider community can start to notice the project and join early on. This is usually seen as a benefit by the managers of most open source projects that struggle for visibility and rely on voluntary efforts for their advancement. This open source model also removes emphasis from the periodic release schedule, as the most recent trends all aim to always keep a usable version of the project as the master or trunk of the public repository. By being open source from day one, the project also gains recognition as being truly open to contributions by the community.

On the other hand, for projects that do not depend on the efforts of random voluntary individuals or organisations, opening the development process to the wider public early on can be detrimental. The first software release is often composed of a minimum working set of tools. If the involved organisations have the required skill set and commitment, it is possible to develop a fully working solution before allowing the wider audience to contribute. This has the advantage of delivering a concrete implementation of the idea, something that the community can relate to and improve. This method also allows the coordination of the release with a marketing action geared towards generating hype or awareness about the project and the problems it aims to solve.

SONATA's team has the skills needed to develop the proposed solution. There is no need to capture external talent on specific areas, so it is not dependant on the knowledge or production of volunteers outside of the project. The first release also needs to be functional, to the level it can be demonstrated to interested parties. It can be accompanied by an appropriate dissemination effort, aimed at increasing its visibility among telecom operators, vendors, solution providers and other stakeholders.

The first public release of SONATA will be at M12, aligned with D5.2, the first integrated SONATA platform based on initial WP3 and WP4 output.

All code will be kept in a private repository until M12, and from then on the repository will be open to the public. Once the release is performed, all repository history will become public and the community will be welcome to participate on the development efforts. The selected version control tool must support this policy.

2.8 Background, Third-Party, Open Source Licensing Compliance

Introduction of partner background and external third-party code must comply with the Controlled License Terms clause of the SONATA Consortium Agreement. The relevant context of these terms is the protection of project results (and partners' use of such results) from proprietary, "copyleft" or other licensing terms that could hinder future plans.

Tools, registries, workflows and analysis must be put into place, following compliance practices that are compatible with (1) the Consortium Agreement's "Controlled License Terms", (2) the open source licensing being developed in parallel and (3) the open source communities that the consortium is focused on to help deliver impact. This is being done in the context of WP7 and the overall General Assembly of the project consortium.

2.9 Evaluation Criteria

To support CI/CD tools selection and workflow definition process, a list of specifications was defined, listed in equal priority:

- Easy implementation and integration with other tools
- Easy usage or knowledge of use exists among partners
- Economic and competitive in cost, and furthermore with a preference for open source solutions
- License – should be suitable for the planned usage (e.g. cannot be for academic usage only)
- Supports development of open source along with privacy capabilities (see Section 2.7)
- Supports a wide range of coding languages. including Java, Python and Ruby
- Supports distributed teams and work

3 Adopted Workflows and Tools

3.1 Overview

3.1.1 Development and Quality Flow Overview

In order to obtain a certain code quality and ensure that the official SONATA software can always be built and deployed, the consortium agreed on a specific software development workflow. D5.1 serves as an introduction to this methodology, with additional guides and tutorials available in the wiki for SONATA's development teams.

The basic premise is that all code that is pushed to the official source code repository has to be built and tested automatically. New code can and will only be accepted if its passes all tests. To this end, the high-level development flow, which is built on GitHub and Jenkins, can be found in Figure 1: SONATA's CI/CD Workflow, outlining the process.

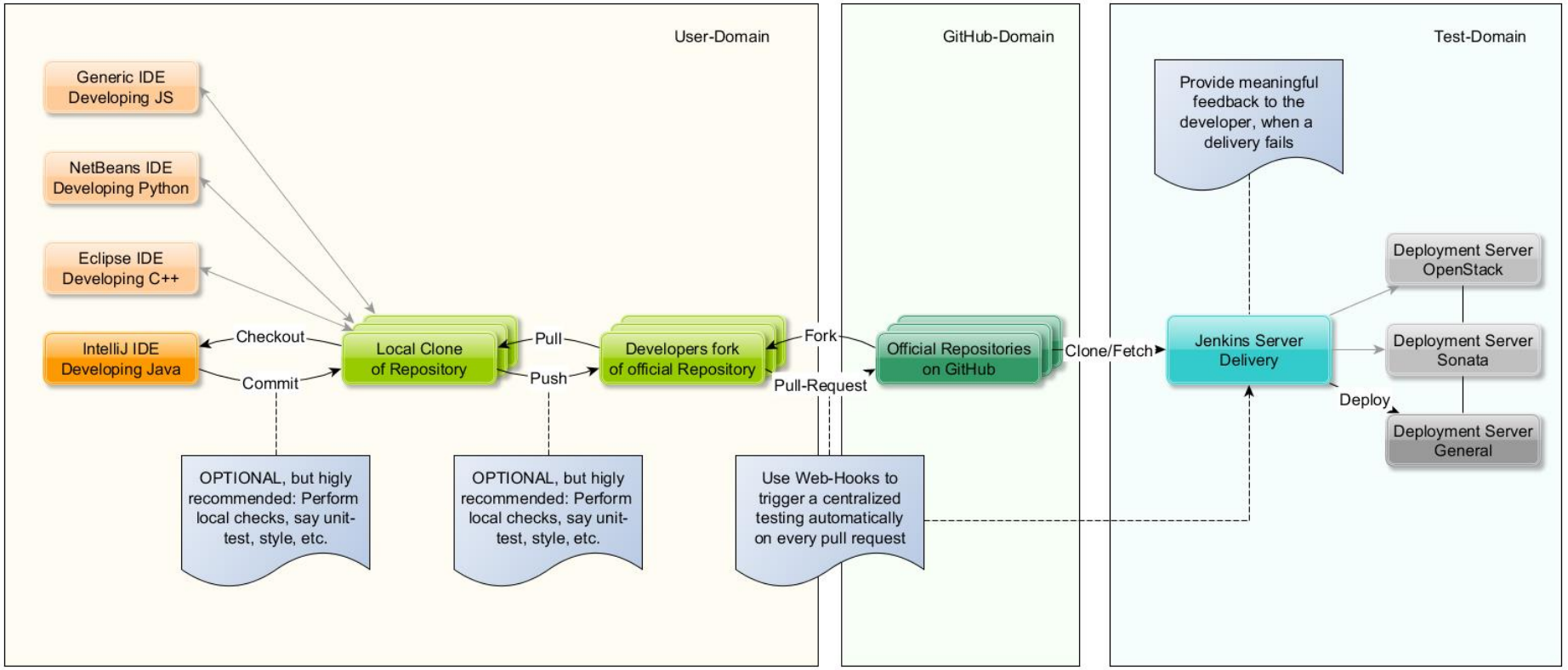

Figure 1: SONATA's CI/CD Workflow

From a high-level perspective, the workflow comprises of the following steps:

1. First, developers have to fork the official repository to their private GitHub account. It is recommended to follow a Git branching approach and create a private development branch to work on. In any case, the developers have to clone their private fork to their local IDE to start developing.

2. Next, developers write code and (unit) tests in their local IDE using the domain or project specific programming language. Optionally, but highly recommended, developers can run local tests, such as build tests, style checks, unit tests, test coverage, and others – perhaps domain or even user specific – before committing any code also to their private GitHub repository. This can also be automated, e.g. by using local Git hooks.

3. Also optional, developers can run additional checks when their code is pushed to their private GitHub repository. This also allows for a private integration of additional build tools, such as Travis-CI and Jenkins.

4. Finally, once the development of a bug fix or a new feature is finalised, developers can create Pull Requests such that the new code (including its tests) can be fetched and merged with the official repository. Every pull request initiated by a SONATA member triggers a Jenkins job on the official Jenkins integration system. Jenkins fetches the Pull Request changes, temporarily merges them with the current official repository, and automatically builds the software and runs some tests. If the build and tests are successful, the Pull Request can safely be merged with the official repository.

In-depth details of this process can be found throughout the Section 3, as well as the project wiki.

3.1.2 Roles Involved

The roles involved in the development life cycle are the following:

- **SONATA User** – those who use the SONATA platform and the SONATA SDK to create a network service. SONATA users are not addressed by this document as it focuses on the development of SONATA itself.

- **SONATA Organisation Owner** – those who own the SONATA GitHub repositories and are in charge for payments. Organisation owners have full administrative control over the GitHub repositories. They may edit that organisation's settings, from profile details to billing information. Moreover, they can create new repositories. (The “organisation” in reference is SONATA, not the individual partners.)

- **SONATA Project Admin** – those who are part of an admin team for a particular project. Usually these are the lead developers. In addition to regular user rights, the project admins can invite external collaborators to the repository. Moreover, they have admin rights for the related Jenkins Jobs. SONATA Project Admins are experienced developers who also lead a particular project within SONATA. They are responsible for reviewing and merging new code from pull requests to the official SONATA repository.

- **SONATA Developer** - those who write the code (and the unit and integration tests), submit pull requests, etc., to build the SONATA platform. These are distinct from the SONATA Users.

- **SONATA Infrastructure Manager** - those who work to guarantee a specific environment (e.g. development, integration, qualification and demonstration) is coherent, ready to host every build.

- **SONATA Release Manager** – those who are responsible for performing the end-to-end system tests and creation of the release packages. Expected to assist the Infrastructure manager in configurations of the development, integration, qualification and demonstration environments.

For the sake of simplicity, we are not considering a Deployment Manager as part of SONATA as this role will be played by the Release Manager, as responsible for the demonstration environment. Each person may play more than one of these roles in different moments in time.

3.1.3 Assets Involved

The set of assets involved in this development lifecycle are the following:

- **Feature Code** - the code that actually implements the 'feature' (meant here in general and not necessarily implying a feature based development methodology), the change to the existing code base.

- **Feature Configuration** - the configuration of the Feature Code, separated from it, and eventually one for each of the environments.

- **Unit Tests** - the code that tests the Feature Code, mocking and/or stubbing whatever component, service, etc., that is outside the component(s) implementing the feature. (Section 2.3.1.3).

- **Integration Tests** - the add-on to the set of tests covering how each component integrates with the remaining ones. (Section 2.3.1.4).

- **System Tests** - the add-on to the set of tests covering all the features the system under test supports or that cover the new feature. (Section 2.3.1.5).

- **Environment Configuration** - optional, only if the feature implies any change in the environment.

- **Build Script** - optional, only if the feature implies any change in the Build Script.

- **Build Configuration** - optional, only if the feature implies any change in the Build Configuration.

3.1.4 Responsibilities

The role responsibilities for these different assets are outlined in the table below.

Table 2: Software Lifecycle Development Roles, Assets and Responsibilities

Asset	SONATA Developer	SONATA Project Admin	SONATA Release Manager	SONATA Infrastructure Manager
Feature Code	X			
Feature Configuration	X			
Unit Tests	X			
Integration Tests	X	X	X	
Code Review		X		
Pull Request Evaluation		X		
System Tests		X	X	
Environment Configuration			X	X
Build Script			X	
Build Configuration	X		X	X
Package Releases			X	X

3.1.5 Selected Tool Summary

To carry out the workflow, the following tools were selected for SONATA's development. The tools were chosen by using the evaluation criteria referenced in Section 2.9, and fit the project's needs.

Table 3: Selected Tools

Domain	Tool	Comments	Selected Tools Detailed in Section...	Relevant Options Covered in Section...
Version Source Control	Git over GitHub	Organisation Account, Private repositories until M12, then Public	3.2	2.2.2
Quality Assurance	SonarQube	Will be used for code style and tests coverage checks	3.3	2.3.2, 2.3.3

Issue and Bug Tracking	GitHub Issues		3.4	2.4.1
Continuous Integration	Jenkins		3.5	2.5.1
Configuration Management	Ansible		3.6	2.6.2

3.2 Version Control

3.2.1 Version Source Control

The current trend among the open source community is clearly the use of Git. For example, Gentoo's Layman Overlay Manager contains a set of contributed repositories. The software is compatible with most VCS, leaving the choice up to the contributor. At the time of writing 396 of 426 are hosted on Git repositories [44].

Current Git hosts such as GitHub or BitBucket allow the integration of tools on the standard development process via standard web hooks, web addresses that are invoked when certain events occur. The modern CI tools have support for Git and can be invoked via the aforementioned platforms' web hooks [45], as well as via periodic polling of the repository. Other tools can be triggered either from web hooks or by the CI tool after the initial build.

It is no surprise that a consultation of the developers among the consortium showed a large bias towards Git as a VCS tool. The advantages of using both a familiar VCS and a common hosting platform are especially important to remove as many hurdles as possible from external contributions, given that the project development will follow an open source model. The most commonly used platform for open source Git hosting is now GitHub, with an ever expanding community and hosting for many of the most important open source projects. SONATA chose GitHub over BitBucket due to the popularity of the platform, in an effort to add visibility to the project and entice the community to join the development efforts of the consortium.

A bronze organisation account has been purchased [46] and private Git repositories will be defined within WP3 and WP4 (the repositories definition will be done by each WP). Access to the repositories will be possible only via specific invitation. Once it will be decided to open SONATA's source to the community the repositories will become public. Full documentation of GitHub organisation account can be found at [47].

This hosting platform allows the creation of an organisation with several repositories, as well as a permission management system that allows control over which developers have write access to each particular repository. This allows the organisation managers to delegate responsibility to a developer or set of developers on a per repository basis.

The precise definition of the roles for the organisation and for each repository will follow a consensus model similar to that of most open source projects, see Section 3.1.2. Further details of the management of the repository follow in the next section, Branching Methodology.

3.2.2 Branching Methodology

With the emergence of Git, the branching of a codebase became part of the daily routine and not something to perform only on release. In fact the ease of branching and merging using Git allowed the emergence of development models that easily accommodate both Continuous Integration and code review as mandatory steps for code contribution.

Authoritative Repository - In a distributed development environment, there is an authoritative repository. Not because it is central, but because the community agrees that an arbitrary repository is the definitive source, meaning that the code on that repository has been vetted by the community and should be considered public. This repository should be considered the upstream for all the developers that wish to contribute to the project.

The authoritative repository seldom, if ever, receives direct commits by a developer. Code contributions come in the form of pull requests from other repositories. Each branch on the authoritative repository will have one or more developers with permissions to merge those contributions to the appropriate branch. Those developers are the SONATA Project Admins.

3.2.2.1 Development Flow

The first step for a new developer is the creation of a personal copy of the repository, a fork. The developer can perform any changes he chooses to on the project. Any number of commits on each personal branch can be performed. Usually each personal branch relates to the development of a feature or issue fix. When the developer is ready to contribute the code to the authoritative repository, a pull request must be created. The pull request is a request for the SONATA Project Admin to fetch the set of changes contained on the developer's selected branch, and publish them as a part of the authoritative repository [48].

At this stage, the SONATA Project Admin can review the code, ask for more (or less) changes, and a CI build can be performed to check the code for functional problems. A series of other tools such as static code analysis or style checkers can also be plugged at this stage of the contribution to the project. By calling all the tools at this stage, and requiring a human to accept the code manually, this process eventually becomes a review before commit process. For further details see Section 3.3.4.

After the pull request is accepted, all the other developers can rebase their forks and local repositories against the upstream repository, effectively re-applying their changes on top of the new project tree in order to avoid possible merge conflicts. If this step is skipped and a pull request is created from a repository, Git is good enough to check the merge for conflicts and only abort the process if any are found. If that is the case, the developer should rebase the repository and resubmit the pull request [49].

This development methodology allows the SONATA Project Admin to keep a simple repository layout, with tags for releases and branches for each release hotfixes. With each pull request carefully inspected and tested, this workflow helps in the construction of high quality software.

The granularity of each pull request can vary greatly, from simple trivial typo fixes to fully matured features; it is up to the Project Admin to decide how granular each accepted pull request should be.

This methodology is a mix of branching and forking, with the intent of keeping the number of people with write access to the main repository of each project to the minimum possible. This

allows the selection of experienced developers for the task of maintainers, and fosters an environment where those developers can pass knowledge to those that have less experience, improving the overall quality over time.

As with all projects, the initial choice of development methodology may at times prove inadequate. As the shortcomings and deviations are identified, they must be analysed, their impact measured, and if relevant, the solution should be applied to all the repositories and teams facing the same problems. This means that the branching methodology at the end of the project may be quite different from what is used in the beginning, but those changes and the reasoning behind them should be detailed on the relevant deliverables.

3.2.2.2 Feature Branch

Complex features that need a cooperative work by a team may have a feature branch on the authoritative repository. Alternatively, it can be developed on a branch of the lead developer of the feature project fork. When the time to contribute the new feature to the main repository arrives, the whole feature goes through the same process as a single developer pull request, ensuring two step verification, once by the feature's lead developer, and a second time by the Project Admin. The downside of developing a feature on a separate fork is that the tools used to inspect and test code on the main repository are not available on the new fork, either the feature is developed without access to those tools and only tested upon contribution, or the lead developer for the tool needs to set up a testing environment for that repository. The choice is usually done on a per-feature basis, as the size of the feature may not require the effort to set up the tools.

3.3 Quality Assurance

3.3.1 Development Lifecycle

The software lifecycle for any code developed as part of SONATA includes:

- **Ready for review** - the code is ready (and unit tests ran successfully) for someone else to review it. This step addresses most commonly (and desirably) a single component that has been changed, less often more than one component, but never a significant part of the system. Thus it already contains automatic style checks, running unit tests, an automatic build (just for the single component), manual code review, and - if successful - accepting the Pull Request.
- **Ready for build** - the code review went well, a new build will include it. This build addresses not only one component but - more or less - the complete system.
- **Ready for deploy** - there's a new build available for deploying.
- **Deployed** - a new version of the system is deployed, including the change that originated this version.

It is a known fact that the cost of removing a defect increases exponentially as the artifact carrying the defect evolves in its lifecycle. It is therefore crucial to invest time and resources in finding defects in the earliest phases of that lifecycle.

3.3.2 Style Checks

In order to further increase the code quality SONATA will implements Google Coding Styles for all the various languages used by SONATA. All the Google style guides will include a formal description, which can be used by automatic style checkers and IDE's to assure correct styling [50].

To enforce the usage of the correct coding style, SONATA will leverage Jenkins and connect it to SonarQube.

3.3.3 Tests Coverage

SONATA will automatically calculate the code coverage for all its software modules. To this end, SONATA will leverage Jenkins and connect to it SonarQube. The calculation will be triggered by every pull request, and for every new piece of code a coverage report will be provided to the developer as well as to the SONATA Project Admin of the particular project. The SONATA Project Admin may decide to reject code that is not tested thoroughly.

3.3.4 Code Review

The whole Quality Assurance workflow starts when a developer considers his/her new or updated piece of code ready to be included in the Code Repository. This means that:

- The code was written/updated in a branch of the authoritative repository it is meant to be included in.
- The code was written/updated to implement a specific feature or correct a specific bug.
- The code was written/update taking into account the guidelines in place for the programming language(s) used.
- Unit tests were written/updated, and are executable, that prove the feature works correctly.
- If meaningful, integration tests were written/updated, and are executable, that prove the feature works correctly when it is integrated with the other 'features'/modules it depends on.

Since the automatic tests can not reveal all the potential shortcomings of code, SONATA implements a manual review processes whenever new code is integrated. Thus, new code is merged to the central GitHub repository manually by the SONATA Project Admin after a careful review incorporating all the reports of the automatic tests. To this end, new code must only be merged if it passes at least the build test successfully. It is crucial that all the code in the main repositories can be built at any time. New code should never break or prevent builds of the main branch. Other tests, such as style checks and test coverage, might be considered optional. That is, code might be accepted by the SONATA Project Admin even if these tests fail. However, this should only happen rarely.

Review Request - The Review Request is done by making a Pull Request of the changes (see Section 3.2.2).

Who are the reviewers - SONATA Project Admins

The review process can result in either two conclusions:

a) The reviewed code is ok - The Pull Request is accepted, the pull requester is notified and the reviewed code is integrated into the main branch of the Code Repository. The associated unit tests (and Integration Tests, if meaningful) will be reviewed, as well.

b) The reviewed code is NOT ok - The Pull Request is not accepted and the pull requester is notified about the reasons. He/she should take the reviewer's comments into consideration and restart the process.

3.3.5 Build

In order to have the shortest feedback loops possible, builds should be continuous and take the least time possible. The most effective way to achieve this is by building just what was impacted with the change. This, of course, depends on the design of the system to be built, as there are several ways to implement such a system. One of them is to split the whole system into atomic, simple and independent 'modules', each providing a focused 'service', a software architectural approach called Microservices [51], which should be considered by WP3 and WP4 when designing their software structure.

Obviously Continuous Builds only make sense if they are automated, based on a Build Script, which is ran to achieve the build.

The build run can result in either two conclusions:

a. The build is ok - When the Automated Build is ok, a new version of the system is made available to the Continuous Deployment process.

b. The build is NOT ok - When Automated Build fails, the Release Manager must check why that failure occurred.

Possible causes of build failure are usually one or more of the following:

i. The environment is inconsistent: for example, when there's a library missing, or the only available version is not compatible with change that is being introduced in the build. In this case, the Release Manager must check whether it is feasible to eliminate the inconsistency and when;

ii. The change to the current build is wrong: 'wrongness' can be many things, like a dependency that is being used in the wrong way (e.g. the number of parameters is wrong, etc.). This kind of issues is usually caught with a good Code Review. The SONATA Developer must in this case be called to correct the situation and issue a new Pull Request with the patch;

iii. The Automated Build script is wrong: especially in the beginning of the SONATA platform development, when many changes will potentially have to be incorporated, this may be the most common cause of Automated Build failure. When most of the system is stable this will less and less be a possible cause of failure for the Automated Build. It is the Release Manager's responsibility to maintain the Automated Build script.

The above mentioned roles are notified of the Automated Build failure: the SONATA Developer author(s) of the last Pull Request (possibly the author(s) of other prior developments may also have to be involved, after a first analysis of what happened), as well as the Release Manager.

A new build will be submitted to for System Tests. This means that each build must have its Functional Tests defined and ran after the Automated Build ends successfully.

3.3.6 Deploy

As stated before, having a brand new build may not imply to immediately deploy it: depending on the use case, a phased approach may have to be used (in the Service Platform case) or an explicit intervention of the user (in the SDK case) might be needed.

Phased deployment - Depending on the specific use case (i.e. the SDK vs. the Service Platform), and on the adopted architecture, deployment may have to be done first for only a subset of the users (e.g., users of a given role first, new users first, etc.). For the SDK, deployment may depend on the SONATA Developer to download the new version.

Deployment might be one of the two following kinds:

i. There is no state to migrate: data generated and stored with the previous version will still be valid in the new version (e.g. the difference between the previous version and the new version is the implementation of an algorithm).

ii. There is a state to migrate: data generated and stored with the previous version needs to be transformed before being considered valid in the new version.

The deploy process can result in either two conclusions:

a. The deployment is ok - A new version of the SONATA system is deployed. Again, having a new version deployed does not necessarily imply that users are using it: for the SDK, it will probably depend on the user downloading the new version, and on the Service Platform side, it will depend on the chosen strategy for the (phased) deployments mentioned earlier.

b. The deployment is NOT ok - If the deployment fails after all the care it has been given, something is terribly wrong. Every artifact that was changed because of this deployment has to be rolled-back. Some frameworks, such as Ruby-on-Rails, include out-of-the-box the rollback back feature for data. When this does not happen, specific scripts must be included together with the deployment script.

3.3.7 Build Monitoring

There may be some issues that are only detected when the new build is deployed and running, even when everything was perfect until the new or updated piece of code has reached its users. And even then, some more issues might be noticed only by users, which mean there should be some 'feedback' mechanisms allowing users to alert about wrong things happening to the system. Therefore there is a need for monitoring and support in this context.

Monitoring systems may have to be adapted or re-configured in order to start looking at the newly deployed system and feature(s). Often it is enough to monitor resources consumed, but some features require specific monitoring parameters that ease the task.

Monitoring is especially important if multiple versions running simultaneously are supported, since tests usually do not cover interactions between different versions of the system.

Changes in the monitoring parameters have to be dealt with high care, since it can jeopardize existing metrics and Key Performance Indicators (KPIs). Depending on who is running the monitoring systems, this change in the parameters to be monitored may be manually implemented or be part of the Deployment Script.

The monitoring process may lead to either two conclusions:

a. Monitoring is ok - When monitoring is ok there is really nothing else to do. Developers are encouraged to grab the next feature to be implemented, although they can still be called in the case of sudden changes in monitoring parameters.

b. Monitoring is NOT ok - A different number or set of numbers or graphics than the one(s) from before the last change introduced does not necessarily mean that a bug was introduced. As new features are added to the system, measurements naturally evolve.

It is usually very time consuming to analyse and reach the cause of bugs that only reflect in the monitoring phase of a system. One of the most common examples of this kind of bug is Memory Leaks: they often reveal themselves only by analysing long series of memory consumption values.

After the still suspicious bug is noticed, developers may be called in order to help identify possible causes, since teams that monitor systems are usually distinct from the developers.

Support - The usage of the SONATA platform will surely reveal issues that will have to be dealt with. Every user can notice something that looks out of the ordinary, and report an issue in the Bug Tracking tool.

3.4 Issue and Bug Tracking

GitHub Issues will be used throughout SONATA's development lifecycle [52]. In order to identify and describe each issue/bug, the labels below are created:

- **Status Labels**
  - In Progress
  - Confirmed
  - Can't Reproduce
  - Won't Fix
- **Severity Labels**
  - Urgent
  - High
  - Medium
  - Low
- **Type Labels**
  - Bug
  - Enhancement
  - New Feature
  - Question
  - Duplicate

3.4.1 Issue and Bug Lifecycle

Creation of the Issue or Bug: Once an issue is created on GitHub, the creator has to give a detailed description of it in the comment area and assign a type label onto it.

- All issues related with erroneous behaviour of SONATA should be labelled as **Bug**.
- All issues related with a planned enhancement work of a feature/part of SONATA should be labelled as **Enhancement** SONATA.
- All issues related with a planned development of a SONATA feature should be labelled as **New Feature**.
- All issues replicating another one that is open or had been opened before should be labelled as **Duplicate** and closed. Replicated issue should be referenced in the comment area by simply typing the issue number (e.g. Duplicates #253)
- All other should be in the form of a question and should be labelled as **Question**.

Handling of the Issue or Bug: Once a new issue or bug appears on the list, it should be assigned to an appropriate person, or a related person with the described feature/part should assign it to herself/himself.

- All issues that are marked with **Bug**
  - should be labelled with an appropriate Severity Label.
  - should be reproduced and then labelled with either **Can't Reproduce** or **Confirmed**. For the ones which cannot be reproduced, a comment should be added asking for a detailed list of steps to reproduce the behaviour.
- All confirmed bugs and enhancements should be labelled with **In Progress** once the assignee starts working on it.
- All issues which have a relation to another issue should be linked by mentioning the related issue in the comment area. (e.g. Related to #432, Blocks #643, Blocked by #435, etc.)
- For every commit related with the implementation or fix of an issue, the issue number has to be stated in the commit comment. (e.g. Fix for #542, Partially fixes #6632, etc.)

Setting Milestones: A milestone with or without a due date can be assigned for a group of issues. Once a milestone is created any issue can be assigned with it by selecting the issue and then selecting the milestone from the Milestones dropdown list. The status of the milestones can be viewed by clicking the Milestones tab on the Issues page.

Closing the Issue: An issue can either be closed from the GitHub UI or by simply adding "Fixes #<IssueNumber>" into the comment text of the commit fixing/completing the issue.

3.5 Continuous Integration Software Development

Both Jenkins and Travis-CI are widely used CI software with proven capabilities. SONATA's choice is to use **Jenkins** as its CI platform. A motivation for the decision was due to the project's plan to begin with private Git repositories on GitHub at the start if development and only on M12 move to public. There is no real advantage of using Travis as it is quite expensive for private repositories and the above mentioned comparisons do not show value that justify that cost. Moreover, among SONATA's partners there is much more

experience in using Jenkins. Project infrastructure from WP6 has already deployed it in November 2015.

3.6 Configuration Management

The main questions put in place for the selection of the most appropriate CM system for SONATA (in addition to the ones mention in Section 2.9) are:

- How will the application stack/package look like?
- What is the size of the infrastructure and the future growth needs?
- What is the preferred scripting language?
- How much in-depth will the engineering team be involved in writing recipes, manifests, or playbooks?

Based on the above questions Ansible was chosen by SONATA as its CM tool. Consortium knowledge was again a key factor, with several partners already experienced with Ansible. It was analysed that the learning curve for creating complex scripting manifest and recipes for Puppet and Chef could slow the project down, and the consortium preferred to rely on current expertise. Moreover, Ansible, while keeping a lot of functionalities, could prove easier for scripting (based in YAML) and compatibility over OpenSSH (without agents).

Regarding the size of the anticipated WP6 infrastructures, the integration and qualification deployments are small to medium in size (i.e. just a few servers).

3.7 License Compliance

A licensing strategy is being developed in the parallel WP7. However, initial guidelines have already been set up between WPs for compliance issues, including WP5:

1. WP3/WP4 partners will be responsible to alert the license and dependencies of any background or external code that they wish to introduce to SONATA's collaborative development.

2. WP3/WP4 leaders will support the process by ensuring that all manual and automatic analysis are taking place within their WP.

3. WP1 and WP5 will consolidate WP3/WP4 input on the wiki, essentially an audit of all developing components in the project.

4. WP7 will provide IP consulting and licensing analysis/management.

5. WP5 is analysing tools that could 1) help automate parts of the workflow and 2) provide greater insight on dependencies. Such tools include Open Hub (former Black Duck) [56], WhiteSource [57] and Fossology [58].

The workflow defined will have developers identify the background and third-party dependencies (and their licenses) in the wiki, so that their respective work packages for implementation (WP3 and WP4), integration (WP5) and exploitation/dissemination (WP7) can analyse proactively. A pending item is the adoption of a more automated system (point #5), which is to be discussed and potentially integrated into the workflow around January 2016.

4 Infrastructure

SONATA will iterate its development effort on three infrastructure variants namely integration, qualification and demonstration. The need for this stems from the fact that at initial stages, where the core development of the SONATA framework is happening, there is a need for an infrastructure that will facilitate the integration of all the project artefacts and components.

When the integration phase is successful, the integrated components will be released as a package for deployment over a qualification infrastructure. The qualification infrastructure will allow further refinement and testing of the released package using a controlled laboratory based environment where the SONATA framework will operate upon. It is anticipated that the qualification infrastructure will allow testing of real-life operations of SONATA framework. Upon successful completion of the qualification tests the release is considered as verified for deployment in the demonstration infrastructure where the selected Use Cases relative to SONATA will be executed and validated. The demonstration infrastructure should have expanded view that will allow demonstration of the added-value SONATA capabilities.

4.1 Requirements for WP6

WP6 will provide WP5 the integration infrastructure in the project's early stages. The integration infrastructure is a means to provide the required components for smooth development, integration and testing of the project's implementations as well as repurpose it to be used for the DevOps support as part of SONATA's framework. The figure below (Figure 2: SONATA's CI Infrastructure) illustrates the anticipated CI infrastructure. This infrastructure will be deployed on the SONATA designated test-beds. The infrastructure comprises of:

1. SONATA test-domain, used for the deployment of the Jenkins master-slave nodes and the integration tests repository. It is anticipated that the aforementioned infrastructure will be scaled according to the performance requirements influenced by the number of developers, the number of tests and the duration of each test. In most of the cases scaling issues may be tackled by introduction of slave Jenkins nodes will address the scaling problems.

2. GitHub-domain comprises of the Git repositories, which as decided will use GitHub public infrastructure. Given that SONATA is releasing open source software, the choice of a public Git repository to host the code from the initial stages is straight forward. The consortium will be moving from private repository (initially used) to public, as soon as the first integrated release is ready in M12.

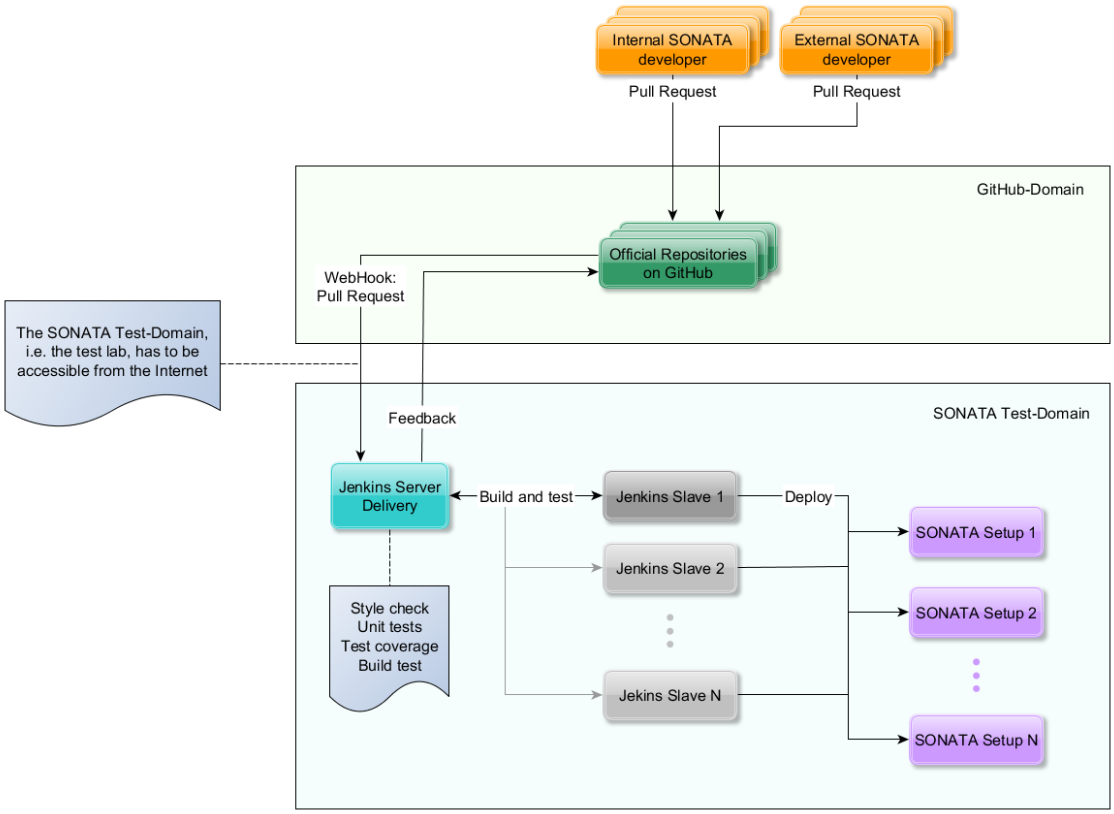

Figure 2: SONATA's CI Infrastructure

The table below details the infrastructures requirements needed to run SONATA's CI. If at any stage there will be a need for additional infrastructure (e.g. as the development efforts increase), the VMs will be upgraded to a stronger flavour.

Table 4: List of CI Infrastructure Requirements for WP6

Component	Operating System	RAM	CPU	Storage	Comments
Jenkins Master	Any Linux OS	8 GB	2	60 GB	Storage needs to be maintained, i.e. old build logs should be deleted
Jenkins Slave	Any Linux OS	4 GB	2	50-100 GB	Storage size Depends on the size of the Repository
SonarQube	Any Linux OS	4 GB	2	40 GB	SonarQube

Initially the Jenkins system will be deployed at NCSRD premises in the datacentre infrastructure that will co-host the rest of the components. A virtualised environment will be used for the hosting of both master and slave node, along with auxiliary network services as Domain Name Server (DNS) that will serve the integration Infrastructure domain. As the utilisation of the system scales up, new slave nodes will be added to the system as appropriated by the performance requirements. The evolution of the Integration Infrastructure will be the Qualification Infrastructure where small scale NFVI infrastructure will be provided in order to test the operational capability of the developed SONATA components.

4.2 Definition of Infrastructure Variants

This section details the 3 types of environments SONATA will run. For each Infrastructure the segmentation bellow is considered:

- **CI Infrastructure** domain supporting the Continuous Integration (CI) components (Input from WP5)
- **Auxiliary Infrastructure** domain hosting the SONATA Components (i.e. Gatekeeper, SDK etc)
- **Network Function Virtualisation Infrastructure** domain with a set of components escalating depending on the Infrastructure flavour.

4.2.1 Integration Infrastructure

The integration environment of SONATA will be a stripped-down infrastructure version exploited for continuous integration activities and verification tests described in this document in the sections above. It contains the minimum required hardware and software components or mock-up versions of real ones. The preliminary implementation of this variant is presented in Section 4.3 below.

4.2.2 Qualification Infrastructure

For SONATA's qualification the infrastructure will be a small-scale laboratory deployed infrastructure comprising hardware and software components, where packaged modules having succeeded in the integration and verification tests will be deployed for qualification activities. A vanilla NFVI/VIM deployment will be the starting point, followed by adaptations/enhancements where necessary.

4.2.3 Demonstration Infrastructure

For demonstrating SONATA the used infrastructure will be a larger-scale pilot architecture, based on a scaled-up version of the qualification infrastructure, enlarged in order to demonstrate UC, PoC and end-to-end system evaluations. The Demonstration infrastructure will serve as the basis for the integration of SONATA pilots.

4.3 Preliminary Integration infrastructure specification

The initial technology choice for the Virtualised Infrastructure Manager (VIM) and the supported NFV Infrastructure envisaged at each NFVI-PoP is the combination of OpenStack [59] and OpenDayLight [60] as it considered to be the de-facto standard in the provision of NFV infrastructure at the moment. Granted the above decision, the figure below illustrates the preliminary version of the SONATA Integration infrastructure variant.

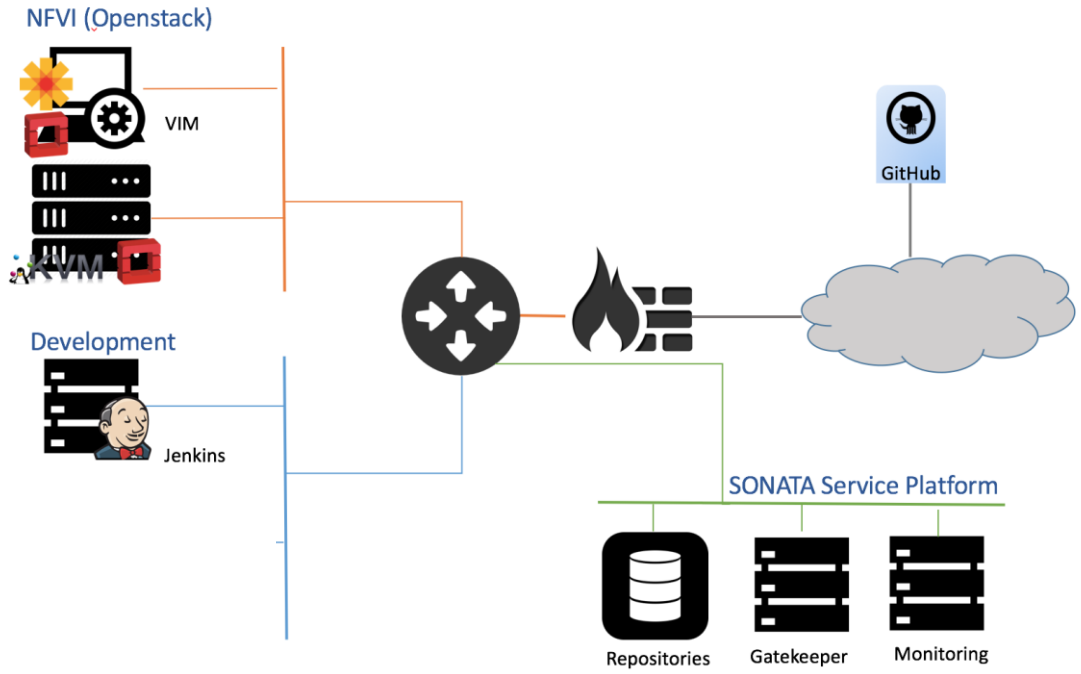

Figure 3 Specification of the preliminary Integration Infrastructure

As the above figure illustrates, the NFVI domain runs on a datacentre controlled by OpenStack (i.e. Nova service) and it may or may not include (depending on the integration and development progress) an OpenDayLight controller integrated with the OpenStack environment using ML2 plugin between the OpenStack network services (Neutron) and the controller. The above environment will be deployed on a single node mode, executing readily available recipes using known Configuration Management software as Vagrant [61] or DevStack [62] . This choice will allow easy replication of the NFVI environment wherever it is required in order to ease the development efforts.

The development domain will comprise the Jenkins nodes (master-slave). Later, the development domain will evolve and encompass all the required tools specified by WP5.

Finally, the SONATA Service Platform domain comprises of the Gatekeeper, the monitoring and a number of repositories as specified by the SONATA architecture and the outcome of WP3 and WP4. It should also be added that the Git repositories will be hosted on GitHub.

The whole infrastructure will be initially hosted at NCSR Demokritos premises and will allow remote access to the developers via a VPN infrastructure based on CISCO Anyconnect VPN technology. The creation of overlay VPN topology is also supported in case it is required for expansion of the above infrastructure towards other labs.

5 Next Steps

With the established SONATA CI/CD workflow and tools in place, several immediate next steps will be taken in parallel through SONATA's WP5 and WP6.

- The CI/CD methodologies described in this document will be rolled out to all SONATA's developers. The kickoff for this rollout was in the plenary meeting that took place in Athens in the last week of November 2015, during which the methodologies were presented followed by a tutorial of the workflows.

- During M6, December 2015, a Developer Hub will be created in the wiki [63], consisting of additional guides, tutorials, etc. to support the developers with a centralised area for all available resources.

- During the first weeks of M7, January 2016, a walkthrough session will be performed, similar to the one done in Athens, refining the process from feedback and additional WP5 discussion.

- In this document general guidelines were defined to control the use of external code. Once SONATA's licensing strategy is proposed by WP7 and approved by the General Assembly, further instructions will be defined in order to ensure compliance. Automated tools towards these objectives are being analysed and potentially adopted for January 2016.

- The required infrastructure and systems will be set up, work that will be done by both WP5 and WP6. A step-by-step approach was selected, in order to enlarge or modify the Integration Infrastructure specification as the project evolves. Initial deployment is scheduled for December 2015.

6 References

[1] https://en.wikipedia.org/wiki/Continuous_integration, Retrieved 16.11.2015
[2] https://en.wikipedia.org/wiki/Continuous_delivery , Retrieved 16.11.2015
[3] Martin Fowler. Continuous Integration. (2006) Article published online at: http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration, Retrieved 08.11.2015
[4] https://en.wikipedia.org/wiki/Unit testing , Retrieved 16.11.2015
[5] https://en.wikipedia.org/wiki/Integration testing , Retrieved 16.11.2015
[6] https://en.wikipedia.org/wiki/System_testing , Retrieved 16.11.2015
[7] http://oss-watch.ac.uk/resources/versioncontrol, Retrieved 16.11.2015
[8] https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control, Retrieved 16.11.2015
[9] http://www.bitkeeper.com/
[10] https://subversion.apache.org/
[11] https://git-scm.com/
[12] http://www.nongnu.org/cvs/
[13] http://darcs.net/
[14] http://bazaar.canonical.com/en/
[15] https://www.mercurial-scm.org/
[16] http://nvie.com/posts/a-successful-git-branching-model/, Retrieved 16.11.2015
[17] http://bryanpendleton.blogspot.pt/2014/07/git-clone-vs-fork.html, Retrieved 16.11.2015
[18] http://www.agilemanifesto.org/principles.html, Retrieved 16.11.2015
[19] http://agilemethodology.org/, Retrieved 16.11.2015
[20] DZone, DZone's 2015 Guide to Code Quality and Software Agility, 2015, Tech Report published online at: https://dzone.com/guides/code-quality-and-software-agility-2015-edition, Retrieved 16.11.2015
[21] Martin fowler, Mocks aren't Stubs, January 2007, Tech Report published online at: http://martinfowler.com/articles/mocksArentStubs.html, Retrieved 16.11.2015
[22] http://checkstyle.sourceforge.net/
[23] http://www.sonarqube.org/
[24] http://cobertura.github.io/cobertura/
[25] Joel Spolsky (November 8, 2000). "Painless Bug Tracking". Retrieved 29 October 2010; via https://en.wikipedia.org/wiki/Bug_tracking_system, Retrieved 12.11.2015
[26] https://en.wikipedia.org/wiki/Issue_tracking_system
[27] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, “Improving bug tracking systems,” in proc. of the International Conference on Software Engineering - Companion Volume, may 2009, pp. 247 –250.
[28] https://guides.github.com/features/issues/
[29] http://www.redmine.org/; https://en.wikipedia.org/wiki/Redmine, Retrieved 16.11.2015
[30] https://www.mantisbt.org/; https://en.wikipedia.org/wiki/Mantis_Bug_Tracker, Retrieved 16.11.2015
[31] https://www.atlassian.com/software/jira; https://en.wikipedia.org/wiki/Jira_(software), Retrieved 16.11.2015
[32] https://bugs.launchpad.net/
[33] https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins; https://en.wikipedia.org/wiki/Jenkins_(software) Retrieved 16.11.2015
[34] https://dzone.com/articles/building-modern-software-delivery-pipelines-with-j, Retrieved 16.11.2015

[35] https://travis-ci.org/, Retrieved 16.11.2015
[36] http://inform.tmforum.org/features-and-analysis/featured/2014/08/google-applies-sdn-configuration-management
[37] http://technologyadvice.com/it-software/configuration-management-systems/smart-advisor/
[38] https://www.scriptrock.com/articles/the-7-configuration-management-tools-you-need-to-know
[39] https://cfengine.com
[40] https://puppetlabs.com
[41] https://www.chef.io/chef/
[42] http://www.ansible.com
[43] http://saltstack.com
[44] https://wiki.gentoo.org/wiki/Main_Page, Retrieved 16.11.2015
[45] https://help.github.com/articles/about-webhooks/, Retrieved 16.11.2015
[46] https://github.com/pricing, Retrieved 16.11.2015
[47] https://help.github.com/articles/creating-a-new-organization-account/, Retrieved 16.11.2015
[48] https://help.github.com/articles/using-pull-requests/, Retrieved 16.11.2015
[49] http://jeffkreeftmeijer.com/2010/the-magical-and-not-harmful-rebase/,
[50] http://github.com/google/styleguide
[51] http://martinfowler.com/articles/microservices.html
[52] https://guides.github.com/features/issues/, Retrieved 16.11.2015
[53] http://thoughts.wallproductions.com/2014/01/jenkins-vs-travis/, Retrieved 16.11.2015
[54] http://stackoverflow.com/questions/32422264/jenkins-vs-travis-ci, Retrieved 16.11.2015
[55] http://www.slant.co/topics/799/compare/~circleci_vs_jenkins_vs_codeship, Retrieved 16.11.2015
[56] https://www.openhub.net/, Retrieved 16.11.2015
[57] http://www.whitesourcesoftware.com/, Retrieved 16.11.2015
[58] http://www.fossology.org/projects/fossology, Retrieved 16.11.2015
[59] https://www.openstack.org/
[60] https://www.opendaylight.org/
[61] https://www.vagrantup.com/
[62] http://docs.openstack.org/developer/devstack/
[63]http://wiki.sonata-nfv.eu/index.php/Developer_Hub