

CloudSocket

CLOUDSOCKET INTEGRATION REPORT

D4.9

Editor Name	Frank Griesinger (UULM)
Submission Date	June 30, 2017
Version	1.0
State	FINAL
Confidentially Level	PU



Co-funded by the Horizon 2020
Framework Programme of the European Union

EXECUTIVE SUMMARY

This integration report documents how a distributed and heterogeneous developed prototype has been integrated.

The nature of the project structure of CloudSocket promotes iterative and parallel development of new functionalities. These developments are created in different work packages. Those functionalities are either desired by the use case partners, by technological partner, or by international research endeavours in different intensity. Throughout the project, especially towards the end, the need intensified to have the latest prototypes integrated into the running and consolidated software stack. This was required to be able to demonstrate the bleeding edge functionalities that were developed by the different partners. This diversity of developer groups also demanded for a structured and automated continuous integration strategy.

We concluded that for a successful provisioning of the CloudSocket tool suite we need to provide *(i)* an demonstration and *(ii)* an integration environment¹. The demonstration environment is a setup of computing facilities that host the branch of the CloudSocket components that run through profound tests and can guarantee stability that is needed for demonstrations. Among others those demonstrations were already held in terms of conferences, workshops, and webinars. The integration environment is a place for every partner to test the latest developments. Some features are chosen by the use case partners and specified as highly wanted. These are then tested in automatic and manual tests and after passing those tests automatically delivered and deployed to the demonstration environment where it can be used in a stable installation.

Recently, DevOps tools emerged to solve different parts of the so called continuous integration or continuous delivery pipeline. Those pipelines define the way from the developed software artefact until the running instance. This comprises different stages like building, packaging, deployment and testing. In this document, we describe the CloudSocket-specific challenges towards those pipelines and which tools are used for which purpose.

Finally, we summarize our experiences and how the continuous integration pipeline will be used in the further course of the project. Furthermore, we will introduce the lesson learnt to sketch the improvements for an ideal setup for future projects that cope with similar problems, which makes this task even more valuable.

¹ These environments are not to be confused with the BPaaS environments that describe the logical connection of different components.

PROJECT CONTEXT

Workpackage	WP4: BPaaS Implementation
Task	T4.5: CloudSocket Integration and Consolidation
Dependencies	Input to D4.10

Contributors and Reviewers

Contributors	Reviewers
Frank Griesinger, Simon Volpert, Jörg Domaschka, Daniel Seybold, Athanasios Tsitsipas (UULM)	Robert Woitsch (BOC), Kyriakos Kritikos (FORTH), Joaquin Iranzo Yuste, Roman Sosa Gonzalez (ATOS)

Approved by: Joaquin Iranzo Yuste (ATOS) as WP 4 Leader

Version History




Version	Date	Authors	Sections Affected
0.1	March 15, 2017	Frank Griesinger (UULM)	Initial version, TOC
0.2	March 17, 2017	Simon Volpert (UULM)	all
0.3	June 1, 2017	Frank Griesinger, Simon Volpert (UULM)	all
0.3	June 1, 2017	Athanasios Tsitsipas (UULM)	Section 2.2 Tool Landscape
1.0	June 2, 2017	Frank Griesinger, Simon Volpert (UULM)	all
1.0	June 13, 2017	Andreea Popovici (YMENS)	Section 5.2 Usage Within CloudSocket
1.0	June 16, 2017	Frank Griesinger (UULM)	Integration of Internal Reviews
1.0-1	June 19, 2017	Frank Griesinger (UULM)	Lessons Learnt
1.0-2	June 26, 2017	Frank Griesinger (UULM)	Integration of Final Reviews

Copyright Statement – Restricted Content

This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of its content.

This is a restricted deliverable that is provided to the community under the license Attribution-No Derivative Works 3.0 Unported defined by creative commons <http://creativecommons.org>

You are free:

	to share within the restricted community — to copy, distribute and transmit the work within the restricted community
Under the following conditions:	
	Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
	No Derivative Works — You may not alter, transform, or build upon this work.

With the understanding that:

Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work.

This is a human-readable summary of the Legal Code available online at:

<http://creativecommons.org/licenses/by-nd/3.0/>

TABLE OF CONTENT

1	Introduction and Problem Statement	10
1.1	Need for Continuous Integration	10
1.2	Concept.....	10
1.3	Extending the concept with Continuous Delivery	11
1.4	Structure	12
2	Continuous Delivery and Integration Setup	13
2.1	Setup.....	13
2.1.1	Phase I: Build.....	13
2.1.2	Phase II: Pack.....	13
2.1.3	Phase III: Deploy	14
2.1.4	Phase IV: Test	14
2.1.5	Integration Pipeline	14
2.2	Integration Tool Landscape.....	14
2.2.1	JIRA.....	15
2.2.2	Jenkins	15
2.2.3	SonarQube	15
2.2.4	Sonatype Nexus	15
2.2.5	Packer	15
2.2.6	GitLab	16
2.2.7	GitLab CI and GitLab Runner	16
2.2.8	Consul	16
2.2.9	Docker	17
2.2.10	Foreman.....	17
2.2.11	Terraform.....	17
2.2.12	OpenStack.....	17
2.2.13	Vagrant.....	18
2.2.14	Maven	18
3	Current Setup	19
3.1	CloudSocket-Specific Challenges	19
3.2	Environments	20
3.3	CloudSocket CI Pipelining.....	21
3.3.1	Phase I: Build.....	23
3.3.2	Phase II: Pack.....	23
3.3.3	Phase III: Deploy	23

3.3.4	Phase IV: Test	23
3.3.5	Hand-over between Environments.....	23
3.3.6	Formalisation of CI Pipeline and Hand-over Process	24
4	Handbook for Developers and Operators	26
4.1	Pipeline Implementation.....	26
4.2	Pipeline Trigger.....	26
4.3	CI Scripts	28
4.3.1	Phase I: Build.....	28
4.3.2	Phase II: Pack.....	28
4.3.3	Phase III: Deploy	28
4.3.4	Phase IV: Test	28
4.4	Environment Variables	29
4.5	Passwords	30
4.6	Component Registry	31
5	Lessons Learnt	32
6	Conclusion	34
6.1	Roadmap	34
6.2	Usage within CloudSocket	34
	References	36
Annex A:	Test Script Example.....	37
Annex B:	Build Script Example.....	38
Annex C:	.gitlab-ci.yml Example Excerpt.....	39

LIST OF FIGURES

- Figure 1 Basic deployment pipeline. [2]..... 12
- Figure 2 Phases of the Continuous Delivery Pipeline..... 13
- Figure 3 Continuous Integration Pipeline..... 14
- Figure 4 Demonstration Environment and the counterpart infrastructure..... 20
- Figure 5 Dependency graph of the CloudSocket components. 21
- Figure 6 Overall pipeline for the CloudSocket tool suite..... 21
- Figure 7 The CI phases along with the tools used in CloudSocket..... 22
- Figure 8 Hand-over process from Integration to Demonstration Environment..... 24
- Figure 9 Different CI pipelines with different states. 24
- Figure 10 CI pipeline with the scripts that implement the phases on the platforms. 26
- Figure 11 Interface in GitLab to store passwords. 30

LIST OF TABLES

Table 1 Current local environment variables for the developers..... 29
Table 2 Execution of the phases per environment. 29
Table 3 Current global environment variables for the developers 30

LIST OF CODE LISTINGS

- Code 1 Pseudo-code of CI pipeline and hand-over process 25
- Code 2 Default pipeline definition 27
- Code 3 Terraform script..... 27
- Code 4 E-mail triggering..... 27

1 INTRODUCTION AND PROBLEM STATEMENT

In this document, we describe our intention, strategy, and the laboratory environment that was set up to achieve continuous integration that helps integrate research prototypes seamlessly and keep the software quality high. To come to this result, the integration platform must cater for appropriate deployment technologies and platforms, testing mechanisms, monitoring abilities, and status reporting.

Work package (WP) 4 mainly deals with the implementation of the CloudSocket platform, i.e. the development of the different CloudSocket components. The development in such a distributed team with different research aspects involves specific requirements. The development of different prototypes demands for an automatized and flexible integration strategy. This was since we worked on two integration issues in parallel: (a) normal integration of well expected and implemented code within the context of WP4; (b) extra integration of research prototypes, possibly in shorter time frame in some cases, to increase the added-value of the final CloudSocket implementation. This was the focus of the corresponding task T4.5 “Integration and Consolidation”.

In general, it is necessary that a stable version of the CloudSocket platform is available all the time. This is not an easy task, considering the high amount and frequent updates of the research prototypes. So a redundant infrastructure provisioning is also a requirement, which allows to have an unstable and a stable CloudSocket instance.

1.1 Need for Continuous Integration

Researchers and developers often spend a lot of time developing their components while the application itself is in an unusable state. A simple reason might be that no one is interested to run the application as a whole until it is finished. This is especially true for research projects where the consortium is usually distributed across multiple countries. They usually develop their part of the application locally and hesitate running the application as a whole because of the complexity of this process. This inevitably leads to a very hard to deploy application, which in the worst case, does not even fit its purpose. The idea of continuous integration (CI) is that the whole application is always in a usable, working state. Every time someone commits any change, the CI system makes sure that the application is built and tested automatically. If that fails, the respective responsible of the faulty component becomes aware of that in order to fix the identified issue right away, before updating the demonstration environment. Continuous Integration has the ability to change the culture of a research group. While using a traditional approach of integrating the application seldom and only before major deployments, it is only sure that the application works at that point in time. If on the other hand, the application is continuously built, it is always clear that for every change someone makes, the application still works. This creates a positive culture of experimentation and innovation by spreading a motivating atmosphere among developers and taking away some fears and doubts.

1.2 Concept

To implement this CI strategy, some prerequisites need to be in place. The most important ones are described below.

- Version Control: Everything has to be part of one, centralised repository. This includes scripts for, building, testing and deploying the application and the application source code itself.
- Automated Build: Code building has to be fully automated. It should be possible for anybody to run a successful build without the use of an IDE.
- Team agreement: Everyone on the team has to agree that it is a number one priority to fix broken builds; otherwise that strategy will be of no success.

Two ways of integration have been selected to address the heterogeneous needs of the developer. One, where the software runs separately and is accessed via an interface, and one where the software is commonly built. In the latter case, it is important, that everyone on the team commits multiple times a day. This is in contrast to various feature branch driven development methodologies, but necessary because it is nearly impossible to do continuous integration for an application as a whole, when it is not clear, what the application as a whole is. The integration of separately installed components need continues testing, hence testing also plays a major role in a successful continuous integration strategy. The team build a comprehensive testing suite consisting of unit, component and acceptance tests. To decrease the time needed for those tests to run (especially the acceptance test) it might be worth to also include smoke tests which runs only some component and acceptance tests like the most important happy paths through the application. Short tests increase the probability that developers adapt and actually use these tests.

- Unit tests check very small portions of the application and do not require to run it as a whole. They are very fast and do not change any state, nor do they require state.
- Component tests check the behaviour of a system. They might require state and require longer to run. However, that state can be stubbed.
- Acceptance tests check the application as a whole and therefore requires to be run in a production-like environment.

1.3 Extending the concept with Continuous Delivery

Continuous Integrations (CI) is only a part of a bigger concept called Continuous Delivery. It combines the ideas of DevOps and Continuous Integration by creating a Pipeline for software delivery (see Figure 1). This pipeline maps the value stream of the application to multiple stages, with Continuous Integration being one step of it. It tries to create an end-to-end approach for delivering software. While CI focuses mainly on the developer, helping them to create good and reproducible builds, a lot of time in the whole delivery process is wasted on manual testing and deployment and their respective feedback cycles. To solve this, the idea is that testing teams have the ability to deploy the application into their environment with the push of a button. The same holds for the operations team. They should be able to easily transition an application from a staging to the production environment.

A typical pipeline consists of the following 4 steps:

- Commit stage. This is where most of the CI happens. Code is compiled, unit and component tests are run, and resulting artefacts are stored.
- Acceptance stage. Here an environment for the application is automatically created. Each created environment is versioned and reproducible. This is where the application is deployed and the automated acceptance tests are run. Upon success, the next step in the pipeline can be run.
- Manual Tests. This step is very similar to the previous one, except for the automated testing part. The creation of the environment is automated, but the testing team tests the necessary functionality manually. Upon success this team reports to the operations team.
- Release. At this point all the mandatory tests have passed and the application is functional in all the different environments. Since all environments are as similar as possible the operations team can easily transition the application into the production environment.

We aimed to capture as many of those aspects in the later described pipeline. Figure 1 shows the UAT (User Acceptance Test) and Capacity stage, both can be covered in the described manual tests step.

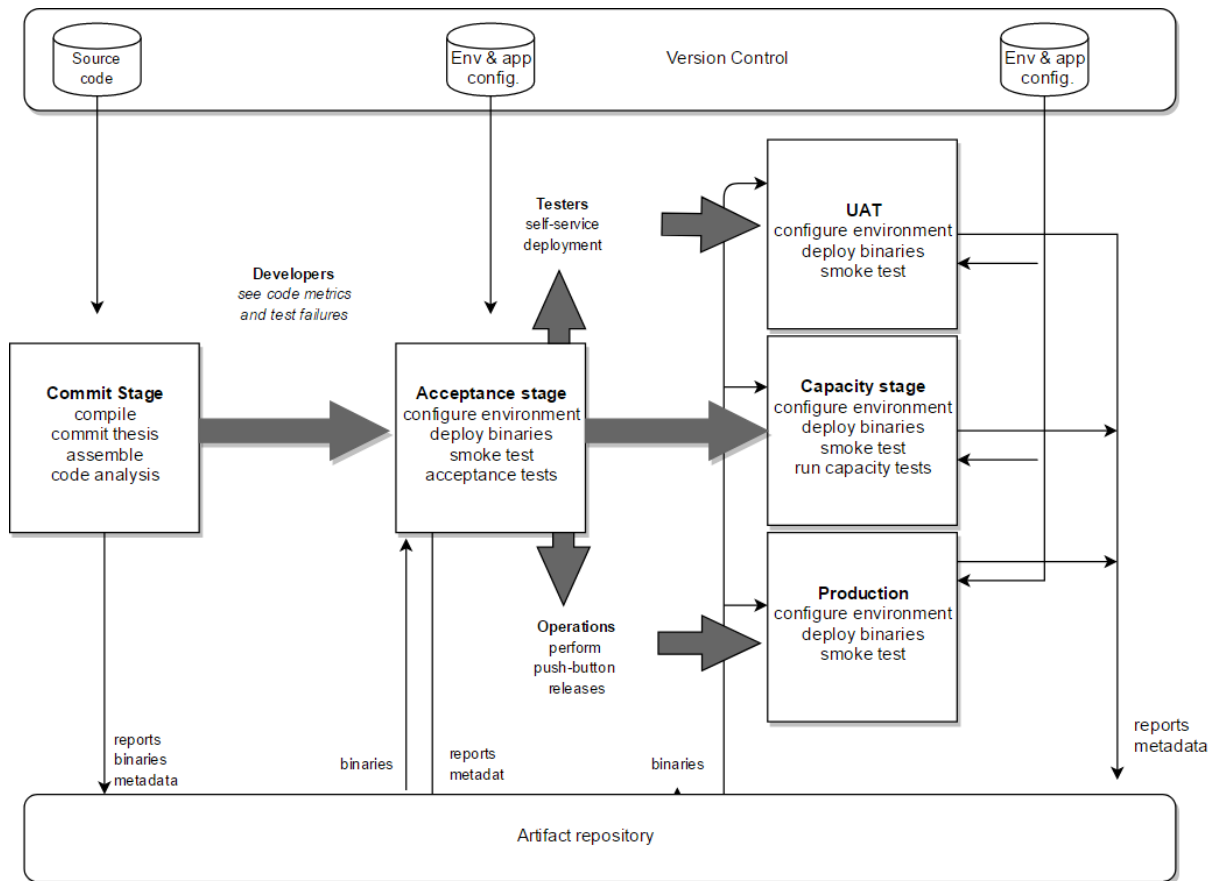


Figure 1 Basic deployment pipeline. [2]

1.4 Structure

The remainder of this document is structured as follows. Section 2 introduces the theoretical base of Continuous Integration (CI) and Delivery, as well as the tools that are employed to realize CI tasks and that we examined. Section 3 is a summary of CloudSocket-specific requirements and the platform we realised to fulfil those. Section 4 describes a handbook for the developers within such a CI platform. Section 5 is a refurbish of our current set up, introducing the lessons learned to the strategy. Finally, Section 6 concludes the document with an outlook and the future work.

2 CONTINUOUS DELIVERY AND INTEGRATION SETUP

The theoretical set-up of a Continuous Delivery (CD) and Continuous Integration (CI) pipeline is described in this section. The CD pipeline describes the steps from code to running component. The CI pipeline describes the combination of CD pipelines of all components (and their new features) to be integrated. First, we show the necessary steps in general, then the tools are shown that implement the activities that take place in the corresponding steps. This will help to justify our actual set-up and tool chain that we use for CloudSocket.

2.1 Setup

The Continuous Integration pipeline consists of the 4 steps (i) build, (ii) pack, (iii) deploy, and (iv) test. Figure 2 shows the workflow of these steps.



Figure 2 Phases of the Continuous Delivery Pipeline.

2.1.1 Phase I: Build

This is the first step of the build pipeline. It is triggered once a change to the codebase has happened and usually runs on a build server although this is not always necessary. Upon triggering this step, a build environment is created, which takes care of all the mandatory steps to build the application.

This can include the following steps:

- pull required build dependencies
- start compilation
- run unit and/or other tests
- prepare compiled artefacts for the next pipeline step

2.1.2 Phase II: Pack

The packing step bridges the gap between the build and deploy step. To create an automated deployment of an application it is usually not sufficient to just start the build artefacts from the previous step because they are not self-contained. Most applications need a very specific runtime environment with complex dependencies and requirements. The goal of the step is to create a self-contained "package" which is easily deployable as a black box without the necessity to know what is inside that package.

This step usually consists of the following sub-steps:

- obtain build artefacts from the previous step
- pull required runtime dependencies
- prepare the application to automatically start with a given configuration (this configuration will be injected in the next step)
- run the component and/or other tests
- prepare the packaged artefact for the next pipeline step

2.1.3 Phase III: Deploy

This step takes the package from the previous step to deploy it on an infrastructure. This is not limited to a single infrastructure. Developers are actually encouraged to deploy packages to different infrastructures to create personal environments.

This step usually consists of the following sub-steps:

- obtain the package artefact from the previous step
- deploy the artefact to an infrastructure while injecting an appropriate configuration, i.e. mostly the replacement of configuration files in the running system

2.1.4 Phase IV: Test

This final step verifies that the previous step was successful by running various tests, such as integration and acceptance tests. Such tests should not be confused with the component-specific unit tests performed during the Build step.

2.1.5 Integration Pipeline

The integration pipeline includes a higher level pipeline that concatenates the pipelines of each component, and a hand-over process from a potentially unstable staging environment to a release environment.

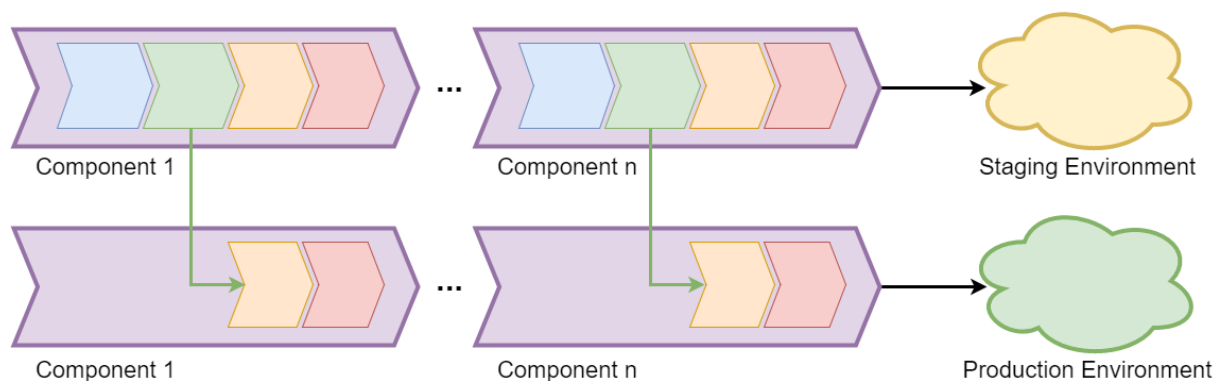


Figure 3 Continuous Integration Pipeline.

Figure 3 shows the CD pipelines (as seen in Figure 2) per component and how they are concatenated to form the CI pipeline per environment. In case of successful testing and manual verification of the features that should go into the production environment, the Pack artefacts will be re-used in the CI pipeline of the production environment. The Test phase will be re-run in the production pipelines to be sure that the system works with the state of the production environment as well. Further (feedback) loops such to cater for roll back strategies and similar are out of scope of this description.

2.2 Integration Tool Landscape

In this section, we present tools that were in our focus when trying to implement the different phases of the Continuous Delivery and Integration. Those tools have to provide functionality from the areas of deployment, testing, resource management, build, configuration management, service discovery, registries, and so on. The fact that there exists a plethora of tools that do not clearly implement one exact function, but cut the edges of multiple CI phases, made us to do bigger analysis to find a combination that meet our requirements the most. Further we describe the tools' strategy, i.e. how we integrated and use them in the CI system.

2.2.1 JIRA

JIRA is an enterprise-level bug tracking platform from Atlassian. It is used in the software industry, to track feature requests and bugs in a software product. The central functionality of the program is to produce relevant information about a problem or a request as well as to carry such information through a process. Each issue can be assigned to the most appropriate staff member, moved through successive statuses towards resolution, as well as accrue comments and link to other issues as needed.

Strategy

The commits of the developer will be linked to the issues/tickets (if the commits are tagged with the prefix "CS-**", where CS stands for CloudSocket as JIRA project, and ** stands for the ID of the issue/ticket).

2.2.2 Jenkins

Jenkins is an open source tool to perform continuous integration and build automation. Its basic functionality is to execute a predefined list of steps. The latter include: (a) perform a software build with Apache Maven or Gradle, (b) run a shell script, (c) archive the build result and (d) afterwards start the integration tests. Jenkins also monitors the execution of the steps and allows to stop the process if one of the steps fails. Jenkins can also send out a notification about the build success or failure.

Strategy

Some projects built by Jenkins are pushed to a maven repository. This is mostly replaced by the GitLab CI by the means of a bash script.

2.2.3 SonarQube

SonarQube does static code analysis to ensure the technical quality of the software projects. The developers get hints on how to improve the code. SonarQube is a web based code quality analysis tool for Maven based Java projects. It covers a wide area of code quality checkpoints which include the seven core axes of code quality: design/architecture, duplications, comments, unit tests, complexity, potential bugs, and coding rules.

Strategy

Projects are added to SonarQube, when they are built by Jenkins. This helps to have a static code analysis and find code of a lower quality.

2.2.4 Sonatype Nexus

Nexus provides a repository for result artefacts, which could either be libraries or release compilation, e.g. in terms of Java binaries. During software development, the build system can download dependencies from Nexus and publish artefacts to Nexus creating a new way to share artefacts within a project.

Strategy

This is needed as an artefact repository to exchange compiled dependencies between independent CI pipelines, we need this as artefact repository.

2.2.5 Packer

Packer automates the creation of any type of machine image. It embraces modern configuration management by using automated scripts to install and configure the software within Packer-made images. Packer offers fast infrastructure deployment, where a user is able to launch completely provisioned and configured machines in

seconds, rather than several minutes or hours. Moreover, Packer enacts deployment to multiple platforms as it creates identical images that can run in any environment (e.g. Amazon AWS, Openstack, VMware etc.). At last, a rigorous pre-build “compilation” of the image assists in early discovery of bugs in the scripts.

Strategy

Packer is used to create the VM images, which are the output artefact of the Pack phase and further used for the deployment.

2.2.6 GitLab

GitLab is an application to code, test, and deploy code together. It provides Git repository management with fine grained source control, code reviews, issue tracking, activity feeds, wikis, and continuous integration. The source control in GitLab enables the user to record and manage changes to projects, files and documents. It helps to recall specific versions later and track the contributions from a collaborative project. Additionally, GitLab contains features that can further assist a collaborative project, such as Issues and a Wiki, and are translated to pending/open points with respect to that project. The GitLab software is updated frequently – a minor release is established every month and a major every year. The frequent updates include features and fixes that will further help the user to gain power and integrate further with other systems that are also a viable solution for a collaborative environment, such as a more advanced ticketing system (cf. Section 2.2.2).

Strategy

The usage for GitLab is manifold. Mainly we host the open-source parts of the CloudSocket platform, but also we use it for ticketing and documentation (repository-specific wikis).

2.2.7 GitLab CI and GitLab Runner

GitLab introduced in version 8, continuous integration and continuous deployment capabilities. With continuous deployment whenever there is a change to the code, it is deployed or made live immediately. This is in contrast to continuous integration, where code is continuously being merged in the mainline and is always ready to be deployed, rather than actually deployed. GitLab has a free and open-source tool to automate the testing and building of projects, namely GitLab Runner, which provides to the users the freedom to experiment with different approaches to build the optimal pipeline for their needs. It can be deployed separately and works with GitLab CI through an API.

Strategy

This tool runs the CI pipelines for each CloudSocket component.

2.2.8 Consul

Consul is a tool for discovering and configuring services in an infrastructure. It provides service discovery, using DNS or HTTP, such that the applications can easily find the services they depend upon (e.g. an API or a service such as database). Additionally, it offers health checking associated with a given service (“is the webserver returning 200 OK”), or with the local node (“is memory utilization below 90%”). This information can be used by an operator to monitor cluster health; it is also used by the service discovery components to route traffic away from unhealthy hosts.

Strategy

Consul is used as a component repository.

2.2.9 Docker

Docker is a platform used to build, ship, and run any applications. Docker provides an easily composable and lightweight container that can change dynamically without disrupting the application as a whole. Docker containers are frictionlessly portable across development, test and production environments running locally on physical or virtual machines, in data centres, or across different cloud service providers. Containers allow users to package large or small amounts of code and their dependencies together into an isolated package. This model then allows multiple isolated containers to run on the same host, resulting in better usage of hardware resources, and decreasing the impact of misbehaving applications on each other and their host system. Docker containers are built from container images using layered file systems ensuring that the container itself contains only the elements needed to run the application.

Strategy

The early phases of the CI pipeline runs mandatory in Docker containers, i.e. the Building step runs in a container. Later stages and the application itself can be run in Docker containers.

2.2.10 Foreman

Foreman is an open source project that helps system administrators manage servers throughout their lifecycle, from provisioning and configuration to orchestration and monitoring. Foreman provides comprehensive, interaction facilities including a web frontend, CLI and RESTful API which enables you to build higher level business logic on top of a solid foundation. A user can discover, provision, and upgrade an entire bare-metal infrastructure or manage instances across private and public clouds (e.g. Amazon EC2, Google Compute Engine, Libvirt, OpenStack, oVirt and RHEV, Rackspace, VMware.).

Strategy

Foreman can be used for resource provisioning on IaaS clouds. Currently we did not integrate it in the pipeline and preferred Terraform for this.

2.2.11 Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Configuration files describe to Terraform the components needed to run a single application or your entire datacentre. For example, the infrastructure is described using a high-level configuration syntax, similar to a blueprint of a datacentre. The description that Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components, such as DNS entries, and SaaS. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what has changed and create incremental execution plans which can be applied.

Strategy

Virtual Machine images are used in Terraform to deploy the application's components.

2.2.12 OpenStack

OpenStack provides the tools and technologies to abstract the underlying infrastructure in an easy and standardized consumption model. OpenStack sits on virtual or physical Compute, Network and Storage technologies and provides the APIs and tools to access these resources in an agile and programmatic manner. OpenStack interfaces with the underlying infrastructure through open source or vendor provided drivers. Additionally, it provides services such as identity management, orchestration, and metering accessed in the same programmatic manner through

the API. High availability can be architected into an OpenStack infrastructure. In next-generation infrastructures, failure handling is designed into the applications to reduce hardware implementation costs. In contrast, an infrastructure-level, high-availability architecture may require redundant hardware, storage and networking, third party software and custom code.

Strategy

It is used as the IaaS cloud on which all virtual machines, needed for the Integration platform, are run.

2.2.13 Vagrant

Vagrant provides an easy way to configure, reproducible, and portable work environments built on top of industry-standard technology and controlled by a single consistent workflow to help maximize productivity and flexibility. A user can test shell scripts, Chef cookbooks, Puppet modules, and more using local virtualisation such as VirtualBox or VMware. Then, with the same configuration, these scripts can be tested on remote clouds such as AWS or RackSpace with the same workflow. Overall, Vagrant gives a disposable environment and consistent workflow for developing and testing infrastructure management scripts.

Strategy

Vagrant was analysed to be later integrated in order to allow the description of the steps not only as shell scripts. It is currently not integrated.

2.2.14 Maven

The de-facto standard for dependency management and building of Java projects. It also manages the creation of Java executables. During this Build process it downloads dependencies from Maven repositories.

Strategy

It is used in the Java projects for the Build phase.

3 CURRENT SETUP

In the following, we describe the project-specific challenges towards continuous integration, as well as our current setup of available environments and DevOps tools.

3.1 CloudSocket-Specific Challenges

Recently, Koetter et al. [1] identified challenges of research projects regarding the collaborative development and in turn integration and operation of the research prototypes.

It is common in research projects that different cultures have to be integrated due to decentralized teams that work on the same or similar issues. They warn of just letting the teams work independently and just a couple of days before the demonstration perform the integration of all components. This is also called the big bang integration, where all components are integrated at the same time – which usually does not work. Koetter et al. propose to integrate new developments seamlessly and fluently as soon as possible. To this end, this is also the approach we decided to go. However, there are more project-specific challenges that are analysed in the following.

Consortium members pursue **commercial** as well as **academic** objectives and therefore, different license models have to be integrated into one strategy. In plain words, this means that not all code can be hosted in one open-source repository platform. Therefore, we have integrated meta-repositories. These do not contain the actual code, but allow to checkout the source code from other sources and be seamlessly integrated into the CI platform. If the code is not open-source at all, we allow to just define the artefact of the Build phase and do not build it on our platform.

Another issue is the **hosting of the components**. It is mainly done in UULM's OpenStack installation. However, the license model of some partners does not allow the external transfer of the artefacts, not even the binaries. This leads to the issue that not all deployments can be fully controlled. This is solved by enforcing the partners to cater in their component hosting (i) the running of multiple instances, and (ii) the offering of an interface to reset an instance externally with a new configuration. For example, in respect to the marketplace developed and hosted by YMENS, the component provides a REST interface that implements this functionality. The use of this method is then introduced in the testing scripts of the corresponding meta-repository. Future iterations of the presented strategy should also introduce interface functionalities that support rollback as well as similar tasks.

Due to the high amount of prototypes, it is also necessary that an **easy service discovery** is supported by the integration system. Unlike the high level service discovery that is developed in the project that caters for semantic findings, we need a way to find the service / components of the tool suite. This means that one partner's component needs to know what the IP address of another component as it interacts with the latter component. This can change frequently in such a dynamic environment. Therefore, a component registry must be available to partners and to the automatic CI pipeline as well. For this purpose, we employ a registry in the form of a key value store. This registry gets updated when a new deployment occurs.

Finally, as we have different environments for different purposes, we need a **state handling** that is dependent on each such purpose. In turn, the pipeline must be agile enough to allow different configurations and processing which implement the application of state transfer or state creation differently. In our case, we need one environment that keeps its state throughout respective demonstrations and usage during different conferences, and similar activities. Furthermore, there is also the need to be able to have one environment that executes tests on reproducible and procedural data generated by the CI platform. The implemented solutions regarding this are two-fold: (i) externalisation of state components, and (ii) enabling different paths in the integration workflow based on the environment. By state component externalisation, it is meant that the state component (mainly database) is not part of the deployed software artefact but hosted outside the CI pipeline. The different paths in the CI pipeline are

determined by a variable that holds the information about the current environment in which the CI pipeline is executed.

3.2 Environments

From the DevOps view, an environment is an instantiation of all components, on physical or virtual infrastructure, that are necessary to have a full system. In the CloudSocket case, an environment is the entirety of all component instances. These environments are not to be confused with the BPaaS environments (see [3]) that describe the logical connection of different components.

Currently, the CloudSocket infrastructure maintains two environments: (i) Demonstration and (ii) Integration.

The *Demonstration Environment* has the purpose to maintain a stable state of the BPaaS environments, i.e. the CloudSocket platform. It consists of a stable branch (mainly *master* in the Git repositories) of each component. This environment guarantees a demonstrable prototype. This was already employed on conferences, workshops, lectures, and webinars.

The *Integration Environment* consists of the components build on the branches chosen by the developers to be tested with the latest features. Mainly this is the *development* branch of the components.

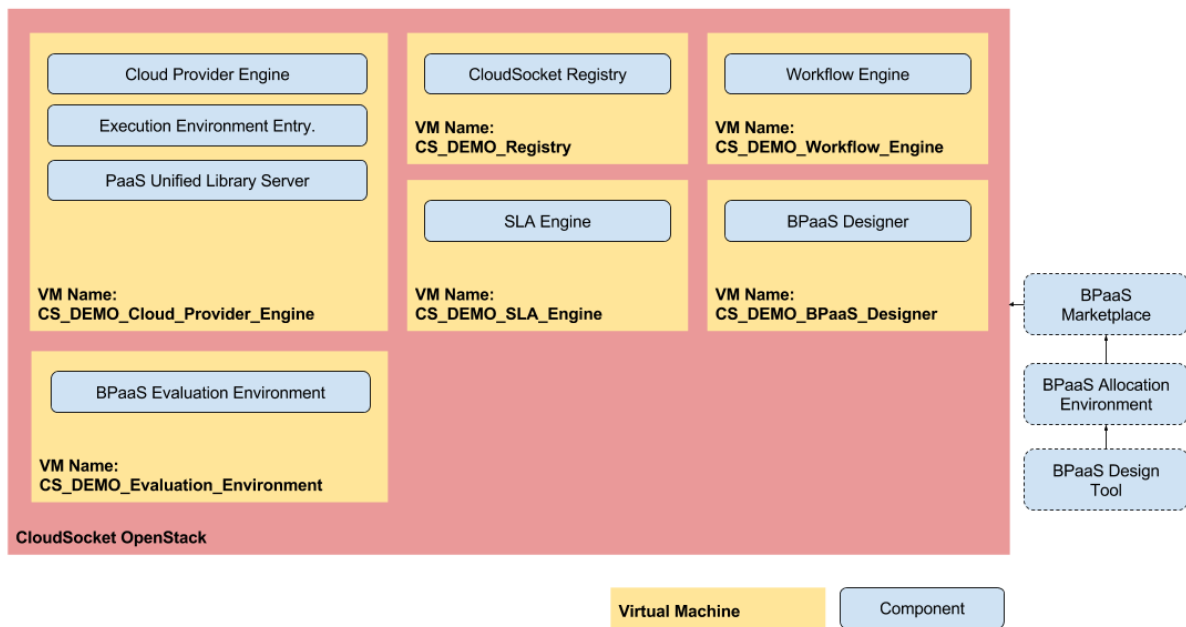


Figure 4 Demonstration Environment and the counterpart infrastructure.

Figure 4 shows the CloudSocket’s components and infrastructure counterparts for the demonstration environment. The integration environment has the same components with a different version (see previous paragraph) on separate VM instances. The BPaaS marketplace is hosted on YMENS facilities. The BPaaS Allocation Environment (i.e. the Allocation Tool) is hosted on FHOSTER facilities. The BPaaS Design Tool is hosted on BOC facilities. The names of the VMs in integration environment are the same, but exchanging “DEMO” with “INTEGRATION”.

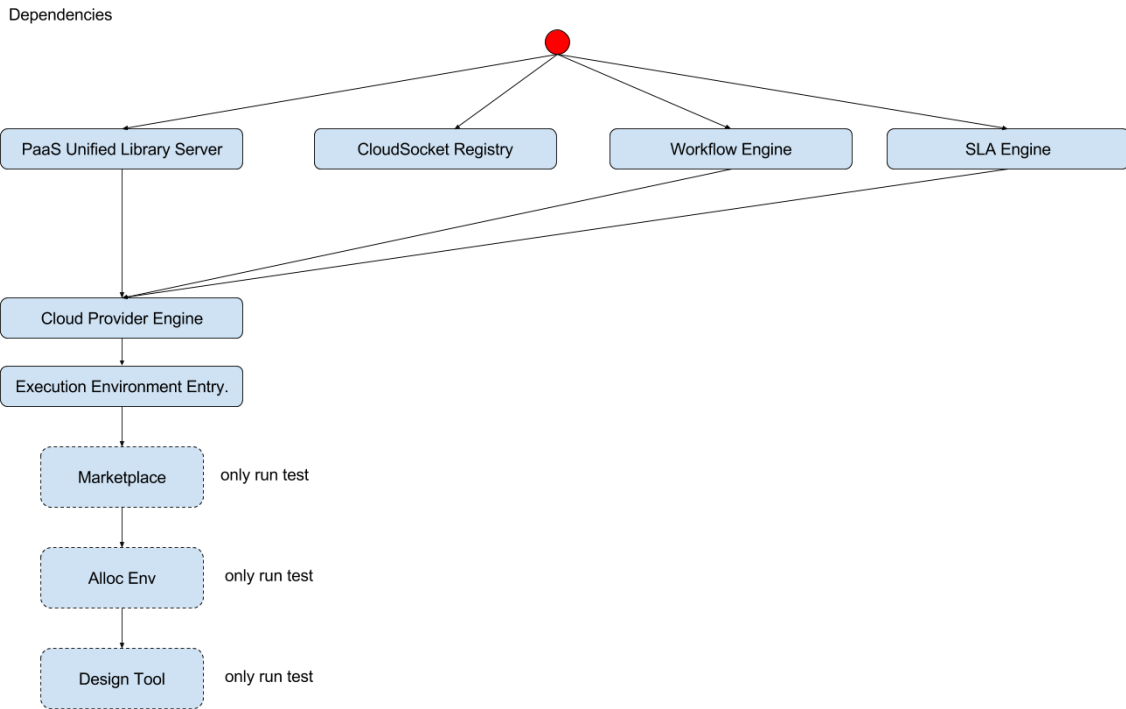


Figure 5 Dependency graph of the CloudSocket components.

Figure 5 shows the dependencies between the components that are needed during the different CI phases. The dependencies show which component's pipeline can run after another. This allows us to identify in which order the different pipelines have to be executed, which results in a higher-level pipeline as shown in Figure 6.

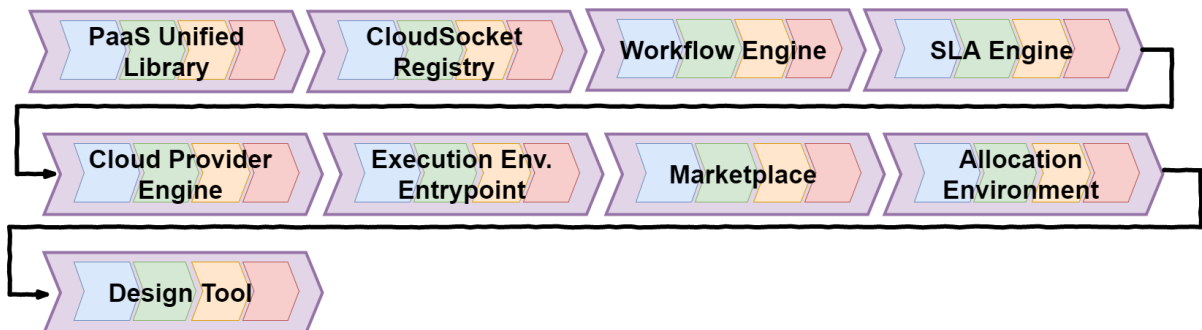


Figure 6 Overall pipeline for the CloudSocket tool suite.

For convenience reason, the prefix “BPaaS” is omitted in the component names, e.g. Allocation Environment instead of BPaaS Allocation Environment.

3.3 CloudSocket CI Pipelining

This section will describe the implementation of the CI pipeline described in Section 2. Figure 7 shows the CI phases along with the tool chain. Each phase creates an artefact that is used in the subsequent phase. The chosen tools were not the only possible solution for the different tasks. However, they match a running and stable CI pipelining strategy that fits the needed flexibility of research projects.

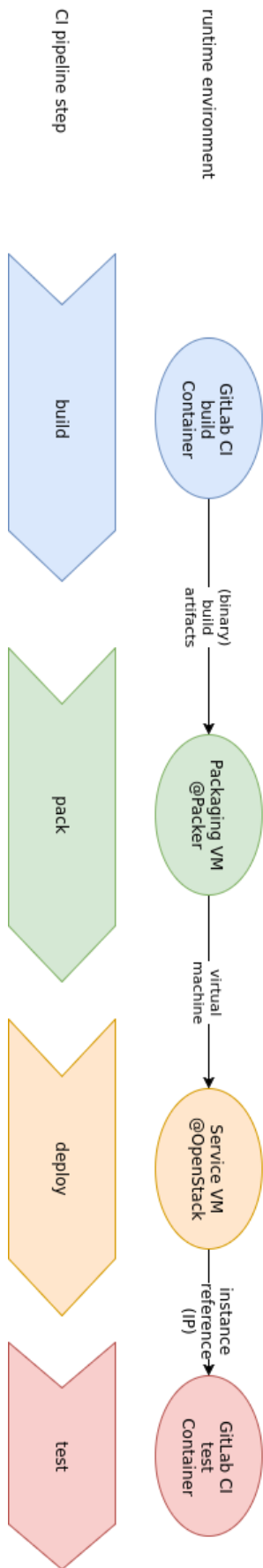


Figure 7 The CI phases along with the tools used in CloudSocket.

The following describes how we have implemented the 4 CI phases, i.e. which tools we used and how we combined them.

3.3.1 Phase I: Build

In this very first pipeline step, a customized build environment is created. We achieve this by utilizing different Docker containers with respect to the applications build requirements. For example, a maven container is used to build Java projects and a NodeJS container is used to build NodeJS applications. The containers can either be official ones from the Docker-Hub or custom built.

It is also possible to use Docker within these Docker containers to build custom Docker containers and publish them on a private registry.

Each Docker container runs on a huge virtual machine within UULM's OpenStack platform, managed by a GitLab runner instance. This runner instance can schedule multiple jobs at once and is able to scale up its resources if necessary.

3.3.2 Phase II: Pack

For packing we use a software called Packer. This tool creates (amongst others) virtual machine images, ready to be deployed on UULM's OpenStack platform. It does so by reading a manifest file, describing the VM characteristics and scripts to run within the image.

After starting the custom VM with the scripts provided, a snapshot of the VM is created. This snapshot is then ready to be deployed in the next step.

As the artefact in this stage is the concrete VM image, we decided to go for a pipelining strategy with images as the opposing strategy with a declarative description of the system, which could be realised by using configuration management tools such as Puppet or Chef. The image pipelining approach caters for a higher reproducibility and an easier less error-prone processing. On the downside, VM image creation is more cost intensive (in terms of storage and processing) as containers or configuration management tool descriptions.

3.3.3 Phase III: Deploy

For the deployment, we use Terraform, which handles the infrastructure management on top of UULM's OpenStack installation. This component ensures the flashing of the VMs with the images created in the Pack phase. The artefact produced in this phase is the running component instance.

3.3.4 Phase IV: Test

Tests are currently implemented in terms of Shell scripts. Further details will be elaborated in the upcoming testing and documentation report D4.10. Those tests run after the component and its dependencies are instantiated. The produced artefact in this stage is the status of the integration: success or failure.

3.3.5 Hand-over between Environments

We applied a hierarchical responsibility system for the CI pipelining. The top level responsible decides when the roll out to the Demonstration environment happens and with which features. Then there is one responsible per component. Each component responsible gets triggered *(i)* when manual tests have to be performed, *(ii)* when tests failed, and *(iii)* when it has to be decided if the integration of a certain feature of the component can be integrated – as he/she has to judge the maturity of the respective feature implementation.

Each component responsible will be informed in terms of emails. The CI pipeline caters for anchor points to manually go to the next step.

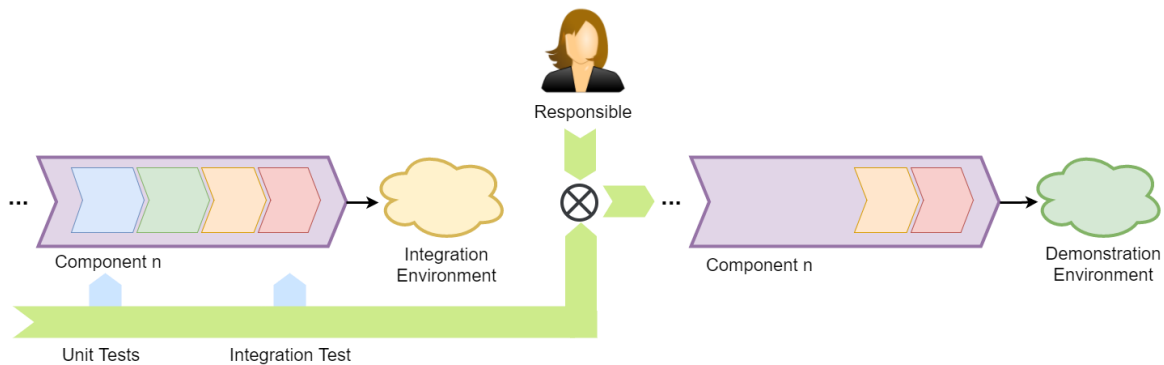


Figure 8 Hand-over process from Integration to Demonstration Environment.

The GitLab CI caters for the needed status reporting as each phase gets a status after it is processed. This can be seen in the GitLab CI GUI (see Figure 9). The shown UI indicates which stages (middle) have which state, e.g. fail (red) or success (green). The integration responsible persons can see if a stage has been passed or failed.

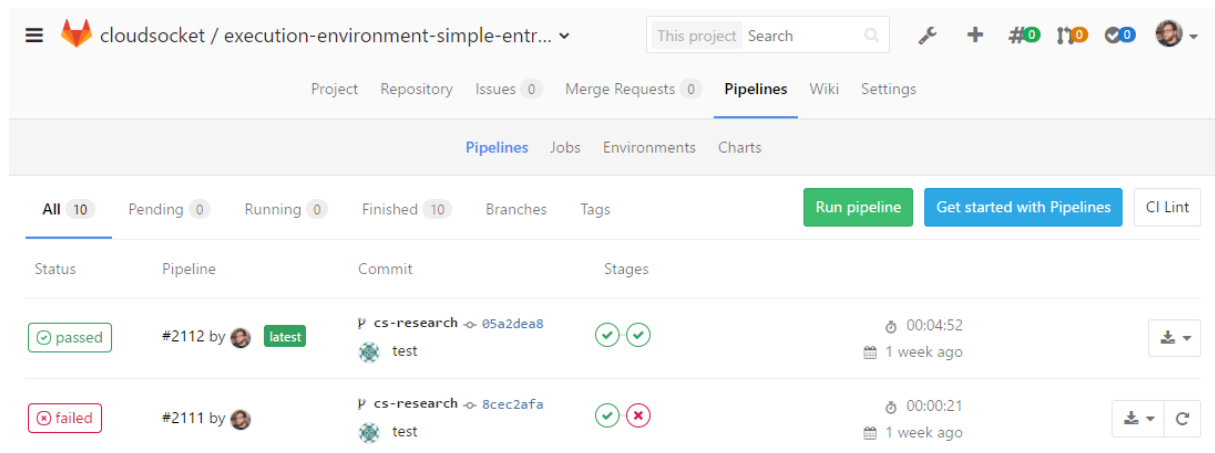


Figure 9 Different CI pipelines with different states.

3.3.6 Formalisation of CI Pipeline and Hand-over Process

We already shown how the pipeline of each component works and how they are interconnected. Finally, a hand-over between Integration and Demonstration Environment happens. Code 1 shows the CI pipeline and hand-over process as pseudo-code.


```

Event ChangePushedToRepository(component) {
    StartCDPipeline(component)
}

Event ManuallyTriggeredCDPipeline(component) {
    StartCDPipeline(component)
}

StartCDPipeline(component) {
    // YAML is read from repository and executed by GitLab CI
    ExecuteBuild()
    ExecutePack()
    ExecuteDeploy(INTEGRATION ENVIRONMENT)
    ExecuteTest()
    If component has manual test
        Inform Test responsible by email
        Wait for successful test
    If all tests successful and component.nextComponent exists
        StartCDPipeline(component.nextComponent)
    Else if component is last
        TriggerHandover()
}

TriggerHandover() {
    Inform top-level responsible via mail that all run through
    For each component
        Inform component responsible that handover can be done
    If all agree to go
        For each component
            ExecuteDeploy(DEMONSTRATION ENVIRONMENT)
            ExecuteTest()
        Inform all responsibles about the state
}

```

Code 1 Pseudo-code of CI pipeline and hand-over process.

4 HANDBOOK FOR DEVELOPERS AND OPERATORS

This section explains what the developers need to know to work with aforementioned integration infrastructure of CloudSocket. This includes mainly, how to write the scripts (shown in Figure 10, upper row) and which conventions to follow.

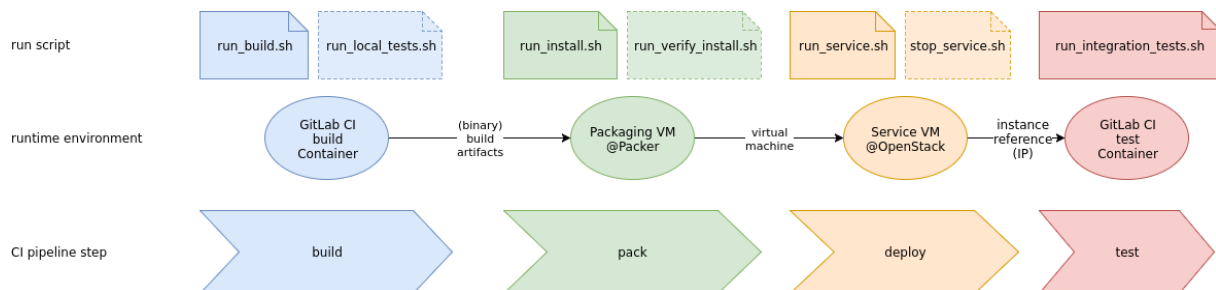


Figure 10 CI pipeline with the scripts that implement the phases on the platforms.

As seen on Figure 10, we have two scripts per CI pipeline step. These are briefly discussed in the following.

4.1 Pipeline Implementation

The actual CI pipeline is performed using the GitLab CI/CD feature. An in depth documentation can be found here². Each repository has its own pipeline, describing it in a special file in its root directory called `gitlab-ci.yml`. It is used to describe all the build stages and the actual (sub-)steps done in each step. Code 2 shows a stripped-down version of most commonly used pipelines.

All the stages are performed one after the other; if one fails, the whole pipeline fails. Each pipeline step can have a different run-time environment by using Docker containers. For java projects for example, we use a special java Docker container with maven. This can be specified by using the `image` directive (see Code 3).

4.2 Pipeline Trigger

The pipeline is usually triggered on each commit and automatically transitions through all stages. However, we do have some meta-repositories, containing only the pipeline definition. This is useful for components, which are either closed source or externally hosted. In these cases only automatic tests are run or mails are sent to trigger the manual running of tests. Manual tests can be specified by using the `when` directive (see Code 4).

When a stage is declared as manual, the component responsible is informed so as to perform her manual work; upon successful component check, she triggers the stage manually in the GitLab CI GUI. This serves as an anchor point for processes that cannot be fully automatized.

² <https://docs.gitlab.com/ce/ci/>

```
stages:
  - build
  - pack
  - deploy
  - test

maven:
  stage: build
  script:
    - maven build

packer:
  stage: pack
  script:
    - packer packerfile

terraform:
  stage: deploy
  script:
    - terraform apply

test:
  stage: test
  script:
    - curl $IP:$PORT
```

Code 2 Default pipeline definition.

```
terraform:
  image: hashicorp/terraform
  stage: deploy
  script:
    - terraform apply
```

Code 3 Terraform script.

```
test:
  stage: test
  script:
    - ssmtp someone@example.com < message.mail
  when: manual
```

Code 4 E-mail triggering.

4.3 CI Scripts

As recommended in [2], we provide at least one separated script per phase of the continuous integration. Those scripts live in the “.ci” folder of the root of each repository. There are some exceptions:

- (i) For components that comprise multiple sub-components in different repositories, we create a meta-repository that includes all corresponding projects and contains only the CI scripts.
- (ii) For components with source code not managed within UULM’s GitLab, a meta-repository is also created which contains only the CI scripts.

The scripts that have to be provided by the developers are triggered in the general pipeline script (see the file “.gitlab-ci.yml” per project in Annex C).

4.3.1 Phase I: Build

In the Build phase, we have the following two scripts:

run_build.sh: This script contains the logic that is needed to build the component from source code to mainly compiled sources. In case this component is built in another environment (e.g. Jenkins), this script just downloads the artefact.

run_local_tests.sh: This script contains the logic to process the component-specific local tests.

4.3.2 Phase II: Pack

In the Pack phase, we have the following two scripts:

run_install.sh: This script installs the component on the operating systems. Here all dependencies that are needed on the underlying platform are installed. Moreover, the needed files and file structures are also generated.

run_verify_install.sh: This script executes a verification, e.g. a dummy call to a port that should be listened to and the response is checked to verify that the installation worked.

4.3.3 Phase III: Deploy

In the Deploy phase, we have the following two scripts:

run_service.sh: This script starts the component as a systemd service.

stop_service.sh: This script stops the service.

4.3.4 Phase IV: Test

In the Test phase, we have just the following script:

run_integration_tests.sh: The so called acceptance tests or integration tests are implemented in this script. These tests check if the interplay with other depending components still works.

These are implemented by the component-responsible persons. The tests will be documented in the upcoming test documentation referred as deliverable D4.10. These tests should not be confused with the unit tests in the build phase as unit tests are performed in the scope of the component and integration tests are performed in the scope of the whole application.

4.4 Environment Variables

As the developers need to write scripts that build, configure, install, and run their components, they must know of some properties of their environment and their connected components. For convenience reasons, the developer does not have to crawl for such information, but the CI pipeline offers the required information to the developers in terms of environment variables. Such environment variables, as those provided by the operating system, are directly accessible in the scripts. Such information is basically the own IP address and the IP addresses of the interconnected components. Further, as some configurations differs from the demonstration to the integration environment, we allow the developer to know in which environment the script is currently executed.

Table 1 shows the local environment variables. They are specific to each component instance.

Name	Value	Build	Pack	Deploy	Test
PUBLIC_IP	The instance public IP	no	no	yes	yes
INTERNAL_IP	The instance internal IP	no	no	yes	yes
CONTAINER_IP	The instance container IP (if available)	no	no	yes	yes
CURRENT_ENV	DEMO or INTEGRATION	yes ³	yes ⁴	yes	yes

Table 1 Current local environment variables for the developers.

Table 2 shows when each script is executed. As it can be seen, the build and pack steps are only executed in the integration environment. This is due to the reproduce-ability. Those exact artefacts that were tested and agreed on during the integration phase can be later re-used in the demonstration environment, with possibly some different configurations.

Pipeline Step	Integration Environment	Demonstration Environment
Build	yes	no
Pack	yes	no
Deploy	yes	yes
Test	yes	yes

Table 2 Execution of the phases per environment.

Table 3 shows the global environment variables, which are available for all components. Such variables might change throughout the project as they refer to components that might be decoupled on different deployment infrastructures. Those variables do not directly correspond to the IP but to a list, as we also cater for having multiple instances of a component; if only one instance is available the array has a length of 1.

Name	Value	Build	Pack	Deploy	Test
SLA_IP	The SLA Engine's public IP	no	no	yes	yes

³ Is only used in INTEGRATION environment. See table below.

⁴ Is only used in INTEGRATION environment. See table below.

SLA_PORT	The SLA Engine's Port, length is equal to #IP * different Ports. If a component only uses 1 Port, then the length is equal to the # of IPs, i.e. length of SLA_IP	no	no	yes	yes
WFE_IP	The Workflow Engine's public IP	no	no	yes	yes
WFE_PORT	The Workflow Engine's public IP	no	no	yes	yes
CPE_IP	The Cloud Provider Engine's public IP	no	no	yes	yes
CPE_PORT	The Cloud Provider Engine's public IP	no	no	yes	yes
UPL_IP	The Unified PaaS library's public IP	no	no	yes	yes
UPL_PORT	The Unified PaaS library's public IP	no	no	yes	yes

Table 3 Current global environment variables for the developers .

An up-to-date version of the environment variables can be found in the corresponding wiki page⁵.

4.5 Passwords

During the different phases, it can happen that scripts need passwords, e.g. to download secured sources or other kind of secure communication. Passwords will be available as environment variables, but opposing to those mentioned above, they are managed in the GitLab CI. This ensures that they are not part of a publicly available repository. As such, passwords can be injected to the build environment, by utilizing a small key-value store integrated by GitLab. These passwords are then represented by environment variables and can be used in any pipeline step. These variables can be set by navigating to the -project settings, choosing "CI/CD Pipelines" and providing the corresponding values under "Secret Variables" (See Figure 11). The key of the passwords is the name of the later environment variable.

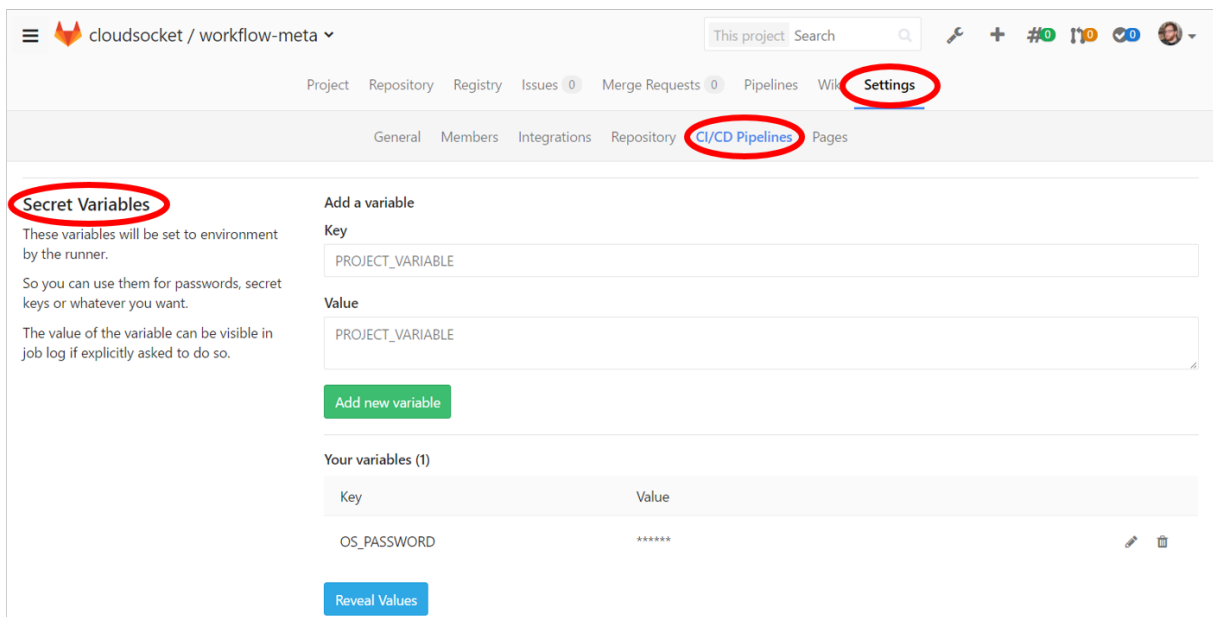


Figure 11 Interface in GitLab to store passwords.

⁵ <https://omi-gitlab.e-technik.uni-ulm.de/cloudsocket/ci-cd/wikis/env-vars>

4.6 Component Registry

The CI platform must cater for a way to keep track of the addresses (mainly IP and port) of the components. This is done via a component registry for which we employ a key-value store, implemented by Consul. This is transparent for the developer as such information is supplied in the aforementioned environment variables.

5 LESSONS LEARNT

The pipelines and engaged technologies for the continuous integration have changed a lot throughout the project. Some decisions were made at points of the projects, when respective implications could not be known because of the changes throughout the project due to the continuous development. Moreover, the implementation of the setup had some time constraints that influenced the choice of the tools. That means, there was no exhaustive analysis of such tools available, but had to be realised during the project lifetime, which resulted in a pre-selection of tools to evaluate to fit the time constraints. In this chapter we summarize the lessons we have learnt, and what improvements can be made to reach an optimized setup.

On the front-end side of the CI system, we found the **graphical user interface** (GUI) not adjustable enough. This was detected in the GUI for the triggering of the next CD pipeline after manual testing has succeeded. So, for instance, currently after component C1 is tested manually, the responsible has to trigger the follow-up project C2 in the GitLab CI GUI. In the future, we desire to have (i) a button in project C1 to trigger the follow-up projects, and (ii) a link in the info mail to the testing responsible that allows to trigger the process with one click.

Furthermore, we found it useful to have a **monitoring dashboard** that is not integrated with the code repositories. This is especially important when the testing and integration responsible persons are not part of the developing teams, which was also the case here. Having said that, it would be useful to have an external monitoring dashboard that shows all projects that belong together in one page along with their CI state. This would help to enhance the ease of use for customised management.

We came across the issue that some components need SaaS applications that should not be accessed from component instances that are only in a temporary state, e.g. to not mix the state that is handled by the SaaS application. To exemplify that, an accounting component queries an external CRM system and creates documents, which remain in the CRM system. This would be a behaviour we would not want to trigger from the accounting component in the Integration Environment. That's why the overall CI system should allow to describe **mock-up services**. Currently this can be done by implementing and deploying such services within the same component that needs them. We found it helpful if such services could be managed by the CI system such that they can be continuously available but are not part of the pipelines. For clarification, a mock-up service is a service that allows requests and gives predefined responses to emulate an actual service that is part of an external application, which is not handled by the CI system.

The main output artefact of the packaging phase is a Virtual Machine (VM) image. This implies that each component has in its non-divisible run-time environment a VM itself. This means, that each deployment employs the start of a VM and the installation of the corresponding image. Due to the transmission of big images and provisioning delays, this can take up to several minutes. A light-weight alternative to this could be the use of **containers**. That means, instead of having a VM image as artefact, we could have, e.g., a Docker image. This would reduce the resource consumption in terms of physical machines, and therefore the costs. In a future implementation we can focus more on containers as a packaging artefact, but still have to allow running specific components without a container, in case such components cannot be run in a container. An optimal setup allows both variations at the packaging stage.

Another issue occurred, when the **state of components** had to be managed. This involved mostly database dependencies for some components. In the current set up we cover this by externalizing the state from the CI system. This means sub-components that only hold the state (databases, filesystems, etc.) are extracted from the actual component, and managed outside the CI system. Those sub-components are therefore handled as SaaS services. For a future implementation, we propose to integrate these as infrastructure services provided by the CI

system. This means that the CI system allows the developer to describe the state (e.g., MySQL and a file with the data) and then runs them as a service that is also versioned, and reproducible (if needed) or permanent.

During an international project, you have often many people involved in teams from different partners that have to work on the same system. To successfully work and test the developed code, everyone should be able to setup the system easily. So a developer should have a running system as quickly as possible. The same applies also for open-source projects in general, since it should be forced to allow different people to have an easy start into the development. For this reason, it would be perfect to have an **out-of-the-box environment** per developer. This means, that the CI systems allows per one-click to deploy an own component instance for any developer. So each developer has her own component instances to work on and test their new developments. This could be easier implemented if containers are exploited in the process. Currently, this is not implemented as all team members were already incorporated in all of the components as needed, and it would be hard to allocate all those resources in terms of VMs.

This principle could also be even made more fine-granular, in terms of allowing **one environment per feature**. This would allow the developer to preserve an environment, while switching to another feature. This was found to be very useful, as we have been developing several features simultaneously in CloudSocket.

6 CONCLUSION

Due to spending many hours in working on the Continuous Integration (CI) strategy, important lessons were learnt and, based on them, certain solutions were achieved. Such results should be exploited in the future. In this section, we draw the future plans for the CI strategy and laboratory environment within as well as after CloudSocket.

6.1 Roadmap

Throughout the development of the Continuous Integration (CI) strategy, we examined many problems that occur specifically and frequently in research projects. We adopted state-of-the-art technologies and methodologies in the CI area and adapted them to our specific needs. This represents valuable knowledge for future projects. Currently, we work on publishing this strategy to provide solutions for research projects in general. Furthermore, we are also writing a book chapter regarding this topic.

As already been described in the previous Section, the lessons learnt will be implemented in the CI infrastructure for future projects. This will also employ a further evaluation of the concepts. CloudSocket has proven that the concepts work, but applying them from the beginning should result in an even better performance of the continuous integration of newly created features into a running application.

6.2 Usage within CloudSocket

The objectives of Task T4.5 are aligned with the Testing and Documentation done in Task T4.6. In T4.5, as described in this document, we set up a platform that is later used. The objective of a testing iteration is to test all CloudSocket components and their integration. During testing, defects will be raised and logged using an excel format provided by YMENS by the corresponding testing responsible, where each test-case will be assigned to a testing responsible.

The testing of CloudSocket integrated platform is performed in iterations. A testing iteration is triggered by the developers each time it is decided to move to a new release in the Demo Environment. Upon triggering, the new testing iteration is organised by the Testing Coordinator. Testing iterations are performed on the Integration Environment.

Each testing iteration will end up with the decision of whether moving the new release to the Demo Environment (Go / No Go). The test report will be used as a base for this decision. Each iteration will include 2 different testing flows:

- Automated tests (unit tests and integration tests) – run by UULM with support from the partners for each component and the integration between components
- Manual tests – which will cover the successful path on the whole lifecycle of a bundle (design, allocation, marketplace, executions, evaluation)

After moving to the Demonstration Environment, a SMOKE test will be performed manually:

- Similar with the manual testing flow – for new bundles
- Partial flow for existing bundles – performed by BWCON / MATHEMA

Testing Coordinator will sequence the needed activities in order to achieve the main testing goal in time.

Each defect will have a measure of criticality and will be linked to a certain test case by the testing responsible. Each test case will be evaluated by the testing responsible as GO or NOT GO, depending on the criticality of the defects found while simulating it.

Testing reports will be stored by testing responsible in a specific folder of the project SVN⁶ and the testing coordinator will be notified for the finalisation of the testing iteration.

The Testing coordinator will centralise and analyse the testing results in an iteration test report and will present it to the Consortium (by email or conference call).

⁶ https://www.cloudsocket.eu/svn/CloudSocketProject7_Workpackages/WP4/T4.6/
Copyright © 2017 UULM and other members of the CloudSocket Consortium
www.cloudsocket.eu

REFERENCES

[1] Falko Koetter, Monika Kochanowski, Florian Maier and Thomas Renner: "Together, Yet Apart - The Research Prototype Architecture Dilemma", CLOSER 2017.

[2] Jez Humble, David Farley: "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", Pearson Education, 2010, 0321670221, 9780321670229.

[3] J. Iranzo et al., 'First BPaaS Prototype - D4.2 - D4.3 - D4.4', CloudSocket project deliverable, July 2016.

ANNEX A: TEST SCRIPT EXAMPLE

```
#!/usr/bin/env sh
# This shell-script is run after the deployment of the ready-packed image and
should be considered as black-box-testing.
# -----
# Purpose: Specify all the steps that are necessary to perform the tests for your
software in combination with foreign apis.
# Please use the "CustomTestCode-Section" in this script, this will ensure that
the script has "set -e" and will return a non-zero exit code on failure, which is
important to prevent release of buggy-software.
# -----
# Author Name: Ferdinand Birk
# Author EMail Address: ferdinand.birk@uni-ulm.de
# -----
set -e

###START CustomTestCode

#echo 'Test Services to ensure Image does work correctly'
journalctl -u workflow-meta.service
sleep 60 # because curl retry doesn't happen on "Connection refused", i.e. when a
dependency isn't up yet
journalctl -u workflow-meta.service
curl --head --max-time 5 --retry 5 "http://localhost:8080/activiti-explorer" |
grep -e "HTTP"

###END CustomTestCode

exit 0
```

ANNEX B: BUILD SCRIPT EXAMPLE

```
#!/usr/bin/env sh
# This shell-script is run as the first step in the CI-Pipeline
# -----
# Purpose: Specify all the steps that are necessary to build your software within
a clean environment.
# Please use the "CustomBuildCode-Section" in this script, this will ensure
that the script has "set -e" and will return a non-zero exit code on failure,
which is important to prevent release of buggy-software.
# Every new generated file under "dist/target/" is promoted to the next stage
and will be accessible under the same "dist/target/"-Directory there.
# -----
# Author Name: Ferdinand Birk
# Author EMail Address: ferdinand.birk@uni-ulm.de
# -----
set -e

###START CustomBuildCode

DEBUG=""
echo 'Running apt update and install...'
#sh -c 'export DEBIAN_FRONTEND=noninteractive; add-apt-repository ppa:openjdk-
r/ppa'
sh -c 'export DEBIAN_FRONTEND=noninteractive; apt-get update'
sh -c 'export DEBIAN_FRONTEND=noninteractive; apt-get install --yes openjdk-8-
jre-headless openjdk-8-jdk'
# Install a recent maven
wget -O "/tmp/maven.tar.gz"
http://www.mirrorservice.org/sites/ftp.apache.org/maven/maven-
3/3.5.0/binaries/apache-maven-3.5.0-bin.tar.gz
( cd /usr/local && tar xzf /tmp/maven.tar.gz )
export PATH=/usr/local/apache-maven-3.5.0/bin:$PATH
echo 'Building workflow-engine'
( cd workflow-engine && bin/bootstrap.sh && dist/bin/make-dist.sh skip-tests )
echo 'Building workflow-parser'
( cd workflow-parser && dist/bin/make-dist.sh )

###END CustomBuildCode

exit 0
```

ANNEX C: .GITLAB-CI.YML EXAMPLE EXCERPT

```
variables:
  OS_USERNAME: cloudsocket_ci
  #OS_PASSWORD: <set password in gitlab repository for security purposes>
stages:
  - build
  - pack
  - deploy
build:
  image: $CI_BUILD_IMAGE
  stage: build
  before_script:
    - git submodule sync -recursive
    - git submodule update --init --recursive
  script:
    - chmod +x .ci/run_build.sh
    - .ci/run_build.sh
    - chmod +x .ci/run_local_tests.sh
    - .ci/run_local_tests.sh
#declare artifacts which should be passed to the next stage of the pipeline
artifacts:
  paths:
    - workflow-engine/dist/target
    - workflow-parser/dist/target
packer:
  stage: pack
  dependencies:
    - build
  before_script:
    - git submodule sync -recursive
    - git submodule update --init --recursive
  script:
    - echo "starting packing"
    - cd .ci/INTERNAL/packer
    - packer build -machine-readable packerfile.json |tee output
    - grep -e "openstack,artifact,0,id," output |cut -d"," -f6 >
    ../../../../image_id
    - echo "artifact output image_id"
    - cat ../../../../image_id
#pass image id to next stage
artifacts:
  paths:
    - image_id
...
```