Horizon 2020
Communications Networks, Content & Technology
Cloud & Software
Research & Innovation Action
Grant agreement n° 731529

Title:	WP5 – D5.1 – Industrial requirements and metrics v1
Date:	May 30, 2017
Writer:	Caleb James DeLisle (XWiki), Vincent Massol (XWiki), Stéphane Laurière (OW2), Jesus Gorronogoitia Cruz (Atos), Lars Thomas Boye (TellU), Paraita Wohler (Activeon), Hui Song (SINTEF)
Reviewers:	Daniele Gagliardi (Eng)

Table Of Content

1. Executive Summary

This deliverable contains descriptions of the 4 use cases as well as clear definitions of the 9 technical KPIs as originally defined in the DoW. Furthermore this deliverable contains initial "baseline" values for the 9 technical KPIs before any of the researched technologies have been applied. Finally this deliverable contains descriptions of the KPI metric collection methodologies used by the use case providers in order that they can be easily reproduced during the next data collection period.

2. Revision History

Date	Version	Author	Comments
5-jan-2017	0.01	Caleb James DeLisle (XWiki)	First draft of KPI descriptions and data collection table.
10-May-2017	1.00	Caleb James DeLisle (XWiki), Vincent Massol (XWiki), Stéphane Laurière (OW2), Jesus Gorronogoitia Cruz (Atos), Lars Thomas Boye (TellU), Paraita Wohler (Activeon), Hui Song (SINTEF)	KPI collection and use case and collection methodology descriptions.
31-May-2017	1.10	Caleb James DeLisle (XWiki), Vincent Massol (XWiki), Stéphane Laurière (OW2), Jesus Gorronogoitia Cruz (Atos), Lars Thomas Boye (TellU), Paraita Wohler (Activeon), Hui Song (SINTEF)	Final draft after review

3. Objectives

The objective of this deliverable is to provide clear definitions for the KPIs initially described in the DoW section 1.1 and to provide initial values of these KPIs in order to have a baseline for measurement of the project's impact within the internally provided use cases. The sourcing and documentation of industrial requirements is also an objective but this deliverable concentrates on the KPIs in order to leave the researchers free to identify before-unimagined ways to build tools which make benefit for the use cases and the software testing industry.

4. Introduction

This document contains elaboration of the 9 technical KPIs found in the table in section 1.1 of the DoW (those relating to objectives 1, 2 and 3). Secondly it contains a description of each of the use cases, including per-use-case industrial requirements where applicable. Finally it contains a clearly described methodology for the collection of each of the 9 technical KPIs and the result of a first collection (before amplification has been deployed). This result will be used as a baseline in future deliverables in order to measure the progress of the project.

5. References

[1] Project reference: Grant Agreement-731529-STAMP.pdf

and the corresponding proposal, same content but document organization and presentation differ: stamp_ec_Proposal-SEP-210342849.pdf

[2] STAMP quality plan: d71_stamp_quality_plan.pdf

[3] D51 – Industrial requirements and metrics: d51_industrial_requirements_and_metrics.pdf

A link to the most recent version of this document.

6. Acronyms

EC	European Commission
DoW	Description of Work
KPI	Key Performance Indicator
LoC	(number of) lines of code
QA	Quality Assurance
API	Application Programmer Interface
IoT	Internet of Things

7. Project evolution metrics

7.1. K01 - More execution paths

This KPI is about test coverage over the code. This can be collected using a tool such as Jacoco or Clover, what is important is that the same methodology/tool is used in each periodic collection. The tool should yield some kind of a percentage which can be used for getting a percent of improvement. The target is 40% improvement which means a 40% improvement is a 40% *reduction in the non-covered code*. For a project which has 60% coverage at the beginning, the goal will be to reach 76%, not 84% and certainly not 100%.

7.2. K02 - Less flakey tests

Flakey tests (sometimes called "flickering tests") are tests which indeterminately and erroneously fail. Flakey tests confuse developers and waste their time by indicating failures when there are none. They can also delay the discovery of real bugs by "training" developers to ignore test failures. We foresee this KPI will be collected by running an automated test suite a number of times in sequence, either manually or by making use of the builds being run on the project's Continuous Integration server, if the Continuous Integration server is being used, the use case providers must take care that no actual regressions are introduced during the period while the sequence of test suite runs is being carried out.

We foresee most use case providers will collect this KPI via three distinct metrics:

1. First the total number of automated tests in the project or module under scrutiny.
2. The number of times that the test suite is run in sequence (taking care that this number of test cycles must be repeated again for collecting before/after KPI values).
3. The number of test cases which failed and then passed, if a single test case failed and then passed and then failed again and then passed again, it should still only be counted once because it's only one test case.

We're hoping to reduce the number of flakey tests by at least 20% before the end of the project.

7.3. K03 - Faster Tests

Clearly the objective here is to get more value out of the tests for every second of time spent running those tests. If ways are found to run the tests in parallel, this is a valid improvement but since it requires additional system resources, it should be noted that some of the metric improvement was due to parallelization. In most use cases, we expect that the measure of test performance will be based on the number of lines of production code which are executed during the test (test coverage in LoC) divided by the amount of time taken to perform the test. This should be accessible from a code coverage tool such as Clover or Jacoco. The goal of the project is to increase the number of lines per second by at least 20%.

7.4. K04 - More unique invocation traces

This is a measure of the number of unique sequences of service invocations which are triggered by running the test suite. The measurement was designed for a microservices structured system which are composed by a set of loosely coupled services, and each task (a user request or a test suite) may have several services involved, e.g., a UI service accepts the request and invoke a business logic service, which in turn queries a database. We call the sequence of services (UI, business, db) invoked for a specific task an invocation trace. In such a microservices system, the topology between the (micro-)services are flexible, due to functional reconfiguration, horizontal scaling, etc., and therefore, the service invocation trace can be different for the same test suite. The objective of measuring the number of unique invocation traces is to analyse how the architecture flexibility is covered by the testing. For example, if we run the test suite on two configurations, a base one and a scaling one that has multiple copies of a service, but the number of unique invocations traces is the same as testing only the base configuration, then it means that the test suite always use one copy of the scaled service in the second configuration, and therefore the second configuration does not really "amplify" the configuration testing. We foresee that this KPI can be recording the sequence of services being involved during testing.

7.5. K05 - System-specific bugs

This is about measuring the observed bugs which happen only on a particular system (for example XWiki running on Oracle Database might have a bug which is not observed on MySQL). For a project/module which does not have multiple different configurations, this will be "not applicable". We foresee this KPI will be collected by defining a matrix of supported configurations (For Example: [Firefox, Internet Explorer, Chrome] + [Oracle, MySQL, Postgres] + [Tomcat, Jetty]). Since bugs from these configurations are identified over time, we foresee the KPI being collected via two distinct metrics: the first being the timespan in which bugs are reported and the second being the number of bugs which are identified in that timeframe. This KPI will be specific to bugs which are specific to one or more configurations, not bugs which occur on all different configurations. We will be seeking a 30% improvement on this metric over the course of the project.

7.6. K06 - Faster deployment for testing

This is a measurement of the amount of time that is needed to set up the software for the purpose of manual testing, this does not include the time required for compiling the source code to binary. We foresee this KPI being collected in the form of a single metric: The average amount of time that is needed in order to set up / deploy the software for manual testing. Our objective is to reduce the time for deployment by 30%.

7.7. K07 - Shorter Logs

This is a measure of how much the logs can be decreased in size *after* they have been made bigger by test amplification, therefore this metric will not be collected in the first period because testing amplification has not yet been implemented in the use cases. We foresee the logs will be collected on a production or production-like deployment and then used in order to amplify tests further. The particular way this KPI will be evaluated is left to the specific use cases but we foresee some documentation of the procedure of deployment and usage procedure for generating the logs and the size of the resulting logs in characters. The goal of the project will be to reduce the size of this log by "an order of magnitude" (therefore a reduction by 90%).

Note: 7.7.1 *Log file size will not be accessible until after we have begun to amplify tests because the amplification will increase log size and this metric is intended to measure the amount by which the log size can be decreased*

7.8. K08 - More crash replicating test cases

This is a metric which represents the number of automated test cases that replicate "crashes". While the exact definition will be left to the use cases, we might consider a crash to be an error which yields a stack trace, is unexpected, and causes visible failure for the user. The only metric which we expect to collect for this KPI is the number of existing automated test cases which replicate crashes that were observed in production or manual testing. The objective of the project is to increase the number of crash replicating test cases by at least 70%.

Note 7.8.1: As the occurrence of actual *crashes* (segmentation fault, etc) in Java is extremely low, some use case providers choose to define a crash to mean an unexpected exception/error behavior, for example that which aborts the process of rendering a webpage for a user request.

7.9. K09 - More crash replicating test cases (% of suite)

This metric is the same as **K08** but the objective is to reach 10% of the entire test suite being test cases which replicate observed crashes. The only metric to collect in addition to what is collected in **K08** will be the total number of test cases in the test suite.

8. Use case descriptions

8.1. XWiki use case

XWiki is an Open Source Enterprise Knowledge Management and Collaboration Solution with extensibility for custom implementations and application development. XWiki is developed by its Open Source community, with XWiki SAS being the main contributor by far. For the purposes of the STAMP project evolution metric collection, XWiki is using the aggregate of the xwiki-commons, xwiki-rendering, xwiki-platform and xwiki-enterprise projects for metric collection. As a highly complex and extensible platform, XWiki requires stringent testing to ensure that API compatibility is not broken for extensions and customizations. XWiki consists of a codebase of 241 thousand lines of code written mostly in the Java programming language and contains 4738 test methods, a number of which are parameterized, making a grand total of 9997 automated tests. In addition, the XWiki SAS QA team carries out additional manual testing of each release version before it is finished. XWiki automated tests fall into three categories: Unit tests which test at the method level and in isolation from the rest, integration tests which test several components together, and functional tests which test the XWiki Runtime. A special type of functional tests are UI tests which use a real web browser and Selenium/WebDriver in order to test the software end-to-end. In addition performance tests are also conducted manually and on regular basis. Between all of the tests, XWiki boasts a 73.2% test coverage rate which is already good but improvements are still possible. Running the test suite takes around 2 hours and 11 minutes which slows down development. Furthermore we need to increase the amount of testing which is possible automatically without requiring the time of the XWiki SAS QA team. Finally we need to reduce the amount of bugs which are observed in production rather than being detected in the testing phase so the XWiki use case fits well with the objectives of the STAMP project and the KPIs explained in section 1.

8.2. Tellu use case

Tellu provides cloud services for collection and processing of data, with a focus on IoT. We provide TelluCloud to service providers and other partners in different domains, for integration in their solutions. The core of our system is a generic platform for collection and processing of data. The use case is this core part of the system, which is now made up of a set of service components (micro-services). The processing starts with receival of data at edges, continues with filtering, storing and rule engine processing and with various possible outputs, such as sending of messages to users or external systems. Service components pass messages to other components. There can be multiple instances of each service component, so load balancing is needed, and in some cases messages need to go to a specific instance. Flexibility with regards to deployment has also been very important, for instance supporting various implementations of the message channels. We need to support everything from running the whole system on a single desktop machine to running in Amazon's cloud infrastructure with auto-scaling and persistent queues. We need to find ways to deploy the system based on different configurations, and to automate integration and system testing, including deployment of micro-services and the needed infrastructure.

The use case code consists of about 77000 lines of code in the early phase of the project. There are 243 unit tests, and most of the code has low test coverage. Increasing test coverage is a need, and a goal in the project. This system was recently refactored from a more monolithic system into the set of micro-services. The code is closed-source and located in a private Git repository (access will be given to STAMP researchers). This repository was set up for the refactored version, and so does not hold history of previous incarnations of TelluCloud. At the date of this report, this new version – 3.1 – is still in development and testing, not quite production ready. Production instances of TelluCloud are still running version 2.9, from before the refactoring.

8.3. OW2 use case

OW2 hosts approximately 50 open source projects in the domains of the Open Cloud, Big Data, Security and the Future Internet. The OW2 use case focuses on the integration of STAMP outcomes into the

OW2 quality platform, on their application to a subset of OW2 projects and on the evaluation of this integration by the project leaders. OW2 runs a quality program named OSCAR, standing for Open source Software Capability Assessment Radar. This program makes it possible to evaluate the Market Readiness Level (MRL) of a project. The STAMP libraries will be integrated into OSCAR so as to refine the way each project's MRL gets computed and to broaden the STAMP approach to all OW2 projects.

Due to the large number of hosted projects at OW2 and to the complexity of the use case, OW2 has decided to narrow down the metric collection to one single project: SeedStack. SeedStack is a Java, general purpose, development solution. It can be used to build different types of projects, and particularly REST-based microservices and applications. SeedStack is an integration solution, bringing together the best open-source libraries in a clean and consistent architecture. To do so, it relies on an extensible kernel/plugin architecture that is capable of automatically activating the plugins present in the classpath. SeedStack is available under the Mozilla Public License. The project was initiated by Peugeot SA. SeedStack uses two main testing frameworks: JUnit and AssertJ. The SeedStack developers spend approximately 30% of their time to maintain approximately 1600 unit tests. Every time a bug is reported, a new issue is created in the SeedStack bug tracker. SeedStack is tested against various environments through Docker images. Beside the testing suite results, the following quality metrics are monitored systematically: code coverage, code duplication, code complexity, major violations.

8.4. Atos use case

Atos Research & Innovation department (ARI) is currently participating on more than a hundred R&I projects funded under the H2020 program, where different software prototypes are being developed. Out of these prototypes, some key assets are promoted within other Atos departments, not only located in Atos Spain, but also in other worldwide GBUs (Global Business Units). In the context of STAMP, a couple of these projects have been initially selected, namely, SUPERSEDE and SMART-FI. SUPERSEDE project develops an autonomic computing supervisor for adaptive systems, based on multi-channel context monitoring. SMART-FI project belongs to the FIWARE ecosystem of accelerator projects, where an IoT-based platform for Smart Cities is developed. Atos brings a pilot named City2Go developed for Malaga city.

In the context of the SUPERSEDE project we have selected a couple of main components for experimentation, namely the Decision Support for Dynamic Reconfiguration, and the Dynamic Adaptation Enactment System, whose source code is available at https://github.com/supersede-project/dyn_adapt. A third candidate component is the SUPERSEDE Integration Framework, which provides frontend-backend interoperability of the entire platform (source code available at https://github.com/supersede-project/integration). SUPERSEDE has been selected for unit test amplification experimentation as SUPERSEDE is based on Java unlike SMART-FI, which is Python based. However, being a complex backend platform, most of existing test suites cannot be considered as unitary, but rather as system/integration tests, as they are demanded for testing its global behaviour in pre-industrial cases. Nonetheless, the Integration Framework does contains a significant number of unitary tests that covers the complete APIs exposed by the integrated backend components. In this preliminary collection of metrics, the first to components have been used, since they provide a more interesting industrial case for Atos, despite their test cases are not unitary, spanning their execution over multiple backend components.

In the context of the SMART-FI project, the platform backend has been selected for experimentation, consisting of the City2Go Dashboard and backend components, some of them being FIWARE GEs (see https://catalogue.fiware.org/ for a FIWARE catalog). Source code is not publicly available, but in the restrictive ARI GitLab repository (https://gitlab.atosresearch.eu). This platform is a perfect candidate for deployment configuration and execution tracing amplification, as the backend components are deployed onto the FIWARE Atos Tenerife Node (http://infographic.lab.fiware.org/) by adopting diverse technologies, including container-based (i.e. Docker) and their optimization (w.r.t. different non-functional properties).

Concerning the specific **K02** requirement**,** in the contest of SUPERSEDE project, but extrapolable to other projects, we may face flakey tests related to concurrent execution and race conditions, but most probably due to test preparations that produces inadequate testing contexts, whose state are invalid as far as the method or module under test concerns. Therefore, test amplification techniques that produces amplified testing environments may identify test failures produced by inadequate testing setups, helping us to detect and correct flakey tests.

In the following paragraphs, we provide Atos specific proposed requirements (KPIs):

K-ATOS-01 - Increased Test Input Space Coverage

Test suites include test cases that are passed with concrete input data sets. Single tests (on concrete method/modules) can be passed successfully with concrete input data sets, but they can failed when executed with other valid (e.g. according to the method specification) input data sets, either because the tested methods were not well designed to cover all possible input data or because the kind of passed data propagates the method execution through other paths not covered by passed tests (see **K01**). This requirement requires to amplify the number of test cases over a concrete software method/module, covering a wider range of input data space, so potential method malfunctioning caused by mismatches between expected and provided input data sets are detected.

This requirement can be assessed by transforming this requirement into the **K01**, therefore, measuring the code coverage under testing. Alternatively, assuming that the valid input data space if specified, S|data, for each data type, as a data partition, for a given method/module under test, the metric defines the number of amplified test cases generated for each partition in the input space with valid input sets. Alternatively, it can be computed as metric the number of tests failing because wrong test input data.

The expected metric improvement is to reach an improvement of 50% on input data space test exploration

K-ATOS-02 - Reduce Cost for Test Setup

Individual tests (unit, regression, system, integration, and so on) within test suites require dedicated setups before launched, including the setup of the software under test (a method, a class, a module, a component, a system), what involves different software development lifecycle phases: compiling, configuring and deployment. Leaving this apart, test suites must configure their @before setups prior to the test being executed, typically preparing the software under test for concrete test configurations and context. This setup can be complex enough requiring significant amounts of work. Additionally, as test execution order cannot be guarantee, the adequate test configuration and context cannot be prepared by other test, so a test in a test chain may depend on the system configuration and context left by a previously executed test.

As the cost of test setup and preparation is high and in a significant number of cases, deficient setups are introducing false test failures (e.g. flakey tests, see **K02**), improving the precision of test setup and reducing their cost in terms of efforts and time required for setup is a highly demanded feature that could fit into the concept of test amplification. Reducing this cost (or time) around 30% is expected. Current proposed metric is the average time percentage required for developing correct test setups for each test over the total time required to develop the entire test suite.

8.5. Activeon use case

Activeeon develops Proactive Workflows & Scheduling which is a toolbox that includes a job scheduler, a resource manager and other useful services (studio, job planner, etc). It is used to orchestrate jobs and tasks across a managed grid. Proactive follows a micro-service architecture but Activeeon decided to restrict the scope to the scheduler and resource manager only, as it's the most critical part of the main product. Today, we spend approximately 2 hours (15 minutes for the unit tests, the rest is for the functional tests) to tests before merging any new changes to our master. This takes time as people are submitting pull requests every day so we need to be sure that our test pipeline is as fast as possible and remain stable at all time. For that, we want to increase the number of relevant tests that are ran during our Pull-Request phase and overnight (when we build snapshots). We also expect to have configuration tests so we cover more ground during tests. Doing so should help us monitoring bugs and regression over time.

STAMP will help us improve our CI cycle:

- more tests added to our test suite
- better bug/regression detection

9. First Data Collection

9.1. XWiki

XWiki collects metrics from its testing infrastructure, including Sonar, Clover and JIRA as well as data which is manually collected from the XWiki developers using the xwiki.org documentation wiki. For **K01** we rely on Clover to collect the test coverage data. For **K02** we collect total test count of 9997 from Clover as well. The number of tests which failed and then passed later on without having been fixed (flakey tests) is collected manually because for each flakey test there is a relevant bug report in the JIRA issue tracker. **K03** amount of time spent on testing is also provided by Clover, it is important to note that this time is measured using the *clovered* code (code modified for calculating test coverage) and so the time spent in testing may be slower, however we plan to calculate the time spent in testing the same way in future measurements so the evolution of this metric will remain unharmed. The number of lines of code which are covered is calculated as the test coverage (from **K01**) multiplied by the total number of lines of code in the project (excluding whitespaces and comments). **K04** does not apply to the XWiki use case as we do not make use of internal services (see **Note 9.1.1**). **K05** has been collected from the XWiki SAS QA team who identified issues which appear on one or more custom configurations but not all (See http://dev.xwiki.org/xwiki/bin/view/Drafts/Test+Metric+Report/#HSystem-specificbugs). **K06** was also requested from the XWiki SAS QA team by asking them how much time they typically spend on setting up XWiki for the purpose of manual testing. **K07** is not ready to be evaluated yet as you can see in **Note: 7.7.1**. **K08** was collected from the JIRA bug tracker by filtering issues with associated stack traces which were fixed and which a test was constructed to prove the fix (See http://dev.xwiki.org/xwiki/bin/view/Drafts/Test+Metric+Report/#HCrashreplicatingtestcases). Finally **K09** was simply calculated as a function of the data collected in **K02** and **K08**. You can find out more about the XWiki test metric collection methodology by examining the test metric report here: http://dev.xwiki.org/xwiki/bin/view/Drafts/Test+Metric+Report/.

Note 9.1.1: XWiki does not make use of internal (micro) services, it is instead a monolithic software application and therefore unique invocation traces are not applicable to this use case.

Note 9.1.2: In our collection methodology, rather than carrying out a specific number of test cycles to detect flakey tests, we collect reports of flakey tests from JIRA bug tracker so all examples of flakey tests which are detected by developers from the Continuous Integration server are counted. (See http://dev.xwiki.org/xwiki/bin/view/Drafts/Test+Metric+Report/#HFlickeringTests) for more information.

Use Case Provider	XWiki
Collection Number	1
K01 - More execution paths	
Tool used for measurement	Clover
Collected value	73.2%
K02 - Less flakey tests	
Total test count	9997
Number of test runs in sample	N/A See: **Note 9.1.2**
Number of tests which failed and then passed	12
K03 - Faster tests	
Time taken during execution of tests	7894 seconds
Test coverage (number of lines of code)	240628 loc (328727 * 73.2%)
K04 - More unique invocation traces	
Number of invocation traces between services	N/A See: **Note 9.1.1**

K05 - System-specific bugs	
Time period of collection	1 year (2016 for the baseline)
Number of configuration-specific bugs detected	139
K06 - Faster deployment for testing	
Time required to deploy software for testing	60 minutes
K07 - Shorter logs	
Description of deployment for sampling	N/A See: **Note: 7.7.1**
Log file size	N/A
K08 - More crash replicating test cases (See: **Note 7.8.1**)	
Number of observed automated test cases which replicate crashes	3
K09 - More crash replicating test cases (% of suite)	
Total number of automated test cases	9997
Number of observed automated test cases which replicate crashes	0.03%

9.2. Tellu

We have set up a toolchain for collecting metrics in STAMP, with a SonarQube server and Maven plugins. The unit test metrics of **K01** - **K03** were collected with Jacoco and SonarQube, except for the time, which were reported directly by Maven. **K04** is calculated based on having a chain of three services with three instances in each, but we are not yet able to verify that all combinations are seen in practice. **K06** is an approximate time for manual deployment. **K07** is not applicable at this time. Looking at our existing unit tests, we do not yet have any crash replicating test cases (**K08** - **K09**).

Note 9.2.1: The microservice version of TelluCloud is new and not yet in production, and we do not yet have any structured form of configuration testing. We are therefore not able to report on **K05** at this time.

Use Case Provider	TellU
Collection Number	1

K01 - More execution paths	
Tool used for measurement	Jacoco + SonarQube
Collected value	19.1%
K02 - Less flakey tests	
Total test count	243
Number of test runs in sample	5
Number of tests which failed and then passed	0
K03 - Faster tests	
Time taken during execution of tests	73 seconds
Test coverage (number of lines of code)	6318
K04 - More unique invocation traces	
Number of invocation traces between services	27
K05 - System-specific bugs	
Time period of collection	N/A See: **Note 9.2.1**
Number of configuration-specific bugs detected	N/A
K06 - Faster deployment for testing	
Time required to deploy software for testing	40 minutes
K07 - Shorter logs	
Description of deployment for sampling	N/A See: **Note: 7.7.1**
Log file size	N/A
K08 - More crash replicating test cases (See: **Note 7.8.1**)	
Number of observed automated test cases which replicate crashes	0
K09 - More crash replicating test cases (% of suite)	
Total number of automated test cases	243

Number of observed automated test cases which replicate crashes	0

9.3. OW2

SeedStack collects metrics from Coveralls, SonarQube and JaCoCo. The SeedStack test suite comprises 1600 unit test classes, which are executed at each build. The duration for running all unit tests is approximately 10 seconds while integration tests require 15 minutes. The time taken during the execution of tests was computed by using Maven and Travis. The number of lines covered by a test was computed using JaCoCo.

K01 was collected using Coveralls and JaCoCo. **K02** is the number of test methods found in the test suite, this is collected by searching for the @Test annotation using the ack bash script such as `ack --java -ch "@Test"`. The development team was asked and did not have documentation of any flakey tests which were discovered in the previous year and generally, flakey tests were not seen as a significant issue for this project. **K03** time spent on testing was collected by observation of the time spent in maven when building on the travis-ci (10 seconds for unit tests and 15 minutes for integration tests) and the number of lines of code was provided by JaCoCo. **K06** was collected by polling the developers on how much time it takes in order to setup the software for the purposes of manual testing when they perform such testing. **K08** was collected by examining the history of reported bugs which contained crash information which were later fixed and with new tests included in the fix patches as there were no such bugs discovered in the bug tracker, this number is zero. Finally, **K09** is derived from the collections in **K02** and **K08**.

Note 9.3.1: The project below does not make use of internal (micro) services, it is instead monolithic software application and therefore unique invocation traces are not applicable to this use case.

Note 9.3.2: **K05** was not collected because this project runs on top of the Java virtual machine and does not use significantly different configurations.

Project name	SeedStack
Collection Number	1
K01 - More execution paths	
Tool used for measurement	JaCoCo + Coveralls
Collected value	70%
K02 - Less flakey tests	
Total test count	1600
Number of test runs in sample	1600
Number of tests which failed and then passed	0

K03 - Faster tests	
Time taken during execution of tests	15min
Test coverage (number of lines of code)	171630
K04 - More unique invocation traces	
Number of invocation traces between services	N/A See: **Note 9.3.1**
K05 - System-specific bugs	
Time period of collection	N/A See: **Note 9.3.2**
Number of configuration-specific bugs detected	N/A
K06 - Faster deployment for testing	
Time required to deploy software for testing	10 seconds
K07 - Shorter logs	
Description of deployment for sampling	N/A See: **Note: 7.7.1**
Log file size	N/A
K08 - More crash replicating test cases (See: **Note 7.8.1**)	
Number of observed automated test cases which replicate crashes	0
K09 - More crash replicating test cases (% of suite)	
Total number of automated test cases	1600
Number of observed automated test cases which replicate crashes	0

9.4. Atos

As mentioned in section 8.4, this preliminary collection of metrics is focused on the SUPERSEDE project and its platform for dynamic reconfiguration and adaptation, whose source code is available at https://github.com/supersede-project/dyn_adapt. This platform consists of a number of thirty projects of different nature, imposed by the baseline adopted technology: Eclipse Plugin-based, Spring-based, etc. These projects can be classified by the adopted project management technology, namely Maven, Gradle or Eclipse OSGI (Manifest-based). The majority of sub-projects are Eclipse OSGI based and they are managed in Maven by using the Tycho plugin.

The test coverage in **K01** has been computed in Eclipse IDE using the Eclipse EclEmma Plugin (Jacoco) and the Clover plugin for Eclipse, over all available test suites. Average figures have been computed out of individual project coverage reported from both tools, but only Jacoco metrics are included in the table. Average coverage has been computed over all sub-projects containing tests (e.g. 12 sub-projects) as the SLOC-weighted average of sub-project coverage:

*average_coverage = sum (coverage(i) * SLOC (i), for each i project) / total SLOC*

In next iterations of baseline metrics computing (to be reported in the next version of this deliverable) the Clover plugin for Maven/Gradle will be used instead, but this is not possible at current state, since test suites for Eclipse OSGI-based projects are embedded within the main source folder, requiring them to be externalized as Eclipse project fragments (see Running Eclipse plugin test with Maven and Tycho)

The number of test suites and tests in **K02** have been computed from GitHub sources by issuing the following Bash scripts:

*find . -name "*Test*.java"* | *wc -l*

*find . -name "*Test*.java" -exec grep -i '@Test' {} \;* | *wc -l*

K03 test execution time are reported by both Jacoco and Clover Plugin for Eclipse. Provided figure is the sum over all test suites executed. SLOC covered in test are computed by summing over the figures reported by Jacoco Plugin for Eclipse. **K05** SUPERSEDE dynamic reconfiguration and adaptation platform adopts a fix architecture, but it can deployed in different configurations of J2EE containers. Indeed we have suffered of some system specific bugs depending on the runtime J2EE containers (i.e. Tomcat/Jetty) and the deployment type (WAR, Maven Spring boot launcher, etc). **K06** time required to deploy software for testing is reported by Jenkins during the CI/CD of the entire SUPERSEDE platform. This computed deployment time aggregates the time required to conduct some few test suites during Maven/Gradle building. **K08** number of observed automated test cases which replicate crashes are computed by running all test cases (e.g. as done for **K01** and **K03**) and inspecting their logs for risen exceptions. **K09** is computed combining **K08** metrics with number of test passed in **K01** and **K03.**

Note 9.4.1: During the development of this platform, we have experienced a number of tests failures that could be included under the denomination of "flakey tests", mostly due to an incorrect test preparation, typically occurring because: i) our tests required complex test preparation that could incur in failures when moving from one platform (i.e. developer platform) to another (i.e. another development platform, integration platform, pre-production platform), ii) inability to guarantee the same testing context (i.e. different storage snapshot), iii) non-deterministic execution. Given figure is an estimation of the percentage of flakey tests over the total, based on previous experiences of the development team, but not based on a systematic and precise accounting. In next version of this document, a more systematic and precise accounting for this metrics will be conducted.

Note 9.4.2: SUPERSEDE dynamic reconfiguration and adaptation platform consists of up to 4 frontend-backend services that are interacting to each other during normal operation, exchanging messages, but it can not considered as adhering to a flexible micro-service architecture, therefore **K04** does not apply.

Note 9.4.3: Reported number of bugs is an estimation collected after surveying the development team.

Note 9.4.4: Metric has been computing by manually running the same tests executed when collecting K01 and K03 and by accounting for those triggering exceptions, after inspecting their execution logs.

For Atos specific requirements, namely K_ATOS_01 and K_ATOS_02, we provide the following estimated metrics:

K_ATOS_01: proposed metric is the number of failing tests caused by wrong test input data. This is estimated, after a team discussion, as a percentage of the total tests passed (see K02), reporting a number.

K_ATOS_02: proposed metric is the average time (in percentage over the total test development) required for developing correct test setups for each test. This is estimated, after a team discussion, as a percentage over the total time required for developing complete test suites.

Use Case Provider	ATOS
Collection Number	1
K01 - More execution paths	
Tool used for measurement	Eclipse EclEmma Plugin (Jacoco)
Collected value	22,13%
K02 - Less flakey tests	
Total test count	94
Number of test runs in sample	79 Tests See: **Note 9.4.1**
Number of tests which failed and then passed	5
K03 - Faster tests	
Time taken during execution of tests	600 sec
Test coverage (number of lines of code)	10711 (22,13% * 48404)
K04 - More unique invocation traces	
Number of invocation traces between services	N/A See: **Note 9.4.2**
K05 - System-specific bugs	
Time period of collection	1 year (last development year)
Number of configuration-specific bugs detected	5 (See: **Note 9.4.3)**
K06 - Faster deployment for testing	
Time required to deploy software for testing	9 min, 24 sec

K07 - Shorter logs	
Description of deployment for sampling	N/A See: **Note 7.7.1**
Log file size	N/A
K08 - More crash replicating test cases (See: **Note 7.8.1**)	
Number of observed automated test cases which replicate crashes	7 See: **Note 9.4.4**
K09 - More crash replicating test cases (% of suite)	
Total number of automated test cases	79
Number of observed automated test cases which replicate crashes	7
K-ATOS-01 - Increased Test Input Space Coverage	
Number of failing test caused by wrong test input data	10
K-ATOS-02 - Reduce Cost for Test Setup	
Average Test setup time percentage (per test)	30%

9.5. Activeon

Activeeon has a running SonarQube instance which is polled several times a day (for every Pull Request on our repositories) and keep track of our code coverage and other metrics. **K02** is collected by running 10 times our unit-test suite and checking/comparing the results every time. **K03** is collected by calculating the average time of the last 10 unit-test suite runs. LoC is computed by our SonarQube instance. **K04** is not applicable with our code. The selected component for STAMP has been stripped from its micro services. So it can be seen as a monolithic application. **K06** is collected by going to our Jenkins instance and calculate the average of our last 10 deployment job.

Note 9.5.1: We have only one environment (one configuration) used in our system-tests so we cannot compare any detected bugs here against another environment.

Note 9.5.2: the projects below do not make use of internal (micro) services, they are instead monolithic software application and therefore unique invocation traces are not applicable to these use cases.

Use Case Provider	Activeon
Collection Number	1

Tool used for measurement	SonarQube
Collected value	30.6%
K02 - Less flakey tests	
Total test count	1225
Number of test runs in sample	1225
Number of tests which failed and then passed	0
K03 - Faster tests	
Time taken during execution of tests	14 min 32 seconds
Test coverage (number of lines of code)	23373 (76633 * 0.305)
K04 - More unique invocation traces	
Number of invocation traces between services	N/A See: **Note 9.5.2**
K05 - System-specific bugs	
Time period of collection	N/A See: **Note: 9.5.1**
Number of configuration-specific bugs detected	N/A
K06 - Faster deployment for testing	
Time required to deploy software for testing	7 min 40 seconds
K07 - Shorter logs	
Description of deployment for sampling	N/A See: **Note: 7.7.1**
Log file size	N/A
K08 - More crash replicating test cases (See: **Note 7.8.1 & Note 9.5.2**)	
Number of observed automated test cases which replicate crashes	0
K09 - More crash replicating test cases (% of suite)	
Total number of automated test cases	1225
Number of observed automated test cases which replicate crashes	0

10. Conclusion

This is the first deliverable containing metric collections, in future deliverables we will be re-collecting the same metrics using the same procedures described in section 9 and we will show percentage differences as defined in section 7 and the DoW (from which these KPIs originate). We have also included the industrial requirements in the use case descriptions and these requirements will be used in the development of the resulting software.