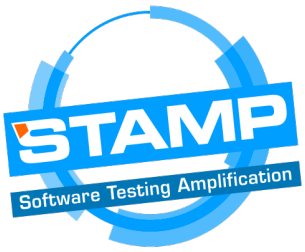

European Commission
Horizon 2020
Communications Networks, Content & Technology
Cloud & Software
Research & Innovation Action
Grant agreement n° 731529

Title:	WP5 – D5.3 – Industrial requirements and metrics v2
Date:	August 30, 2018
Writer:	Vincent Massol (XWiki), Assad Montasser (OW2), Jesus Gorronogoitia Cruz (Atos), Lars Thomas Boye (TellU), Mael Audren (Activeon), Anatoly Vasilevskiy (SINTEF)
Reviewers:	Arie van Deursen (TU Delft)

Table of Contents

1. Executive Summary

This deliverable contains updates to the 9 technical KPIs (KPI 01 to 09) as originally presented in the DoW, and as defined in deliverable D5.1 in more details.

2. Revision History

Date	Version	Author	Comments
28-August-2018	0.9	Vincent Massol (XWiki)	First version after taking into account comments from all partners and comments from the mid-term EU review.
30-August-2018	1.0	Vincent Massol (XWiki)	Include feedback from TUDelft for KPIs K07 and K09.
03-September-2018	1.1	Vincent Massol (XWiki)	Take into account review feedback from Andy Zaidman (TUDelft)
07-Sep-2018	1.2	Caroline Landry (INRIA)	Adding some comments Adding KPIs vs Objectives table in conclusion Fixing table of content issue
10-Sep-2018	1.3	Vincent Massol (XWiki)	Include feedback from Lars (TellU) & Caroline (INRIA)

3. Objectives

The objective of this deliverable is to review the KPIs as defined in deliverable D5.1, i.e., at the very beginning of the project and amend them based on the progress of the STAMP project.

In particular it's important to note that the original KPI definitions that were put in the DoW section 1.1, were done at a time when no tools were developed yet and thus at a time when the use case providers had not started measuring the effects of applying the STAMP tools on their code bases. This means there could have been mismatches with the developed tools (the tools not being able to address some specific metrics, or not in a satisfactory way, or the impossibility to measure the metrics in practice on the use cases. Alternatively, it could be the case that we discover new KPIs that were forgotten initially and that are interesting to measure.

It's also important to keep in mind that the purpose of KPIs is to measure the objectives. Thus one objective of this document is to ensure that the KPIs are still relevant and to amend them when this is not the case.

Another objective is to provide more details on the definitions of the KPIs; how to measure them and the processes to do so.

4. Introduction

This document contains updates on the 9 technical KPIs found in the table in section 1.1 of the DoW (those relating to objectives 1, 2 and 3).

The following considerations were taken into account:

- Whether the developed STAMP tools are relevant to each KPI and where it's not the case, modify the KPI to make it relevant and be able to measure the 6 objectives of the project, and to show both innovative results and industry-relevant data.
- After trying to measure KPI metrics, modify the processes to capture metrics where required to make this capture possible.
- In some cases, showing a percentage of improvement is not enough to gauge both the effort done by the use case providers to reach the target value and how the STAMP tools have made this possible. Indeed, depending on the use cases, a target KPI objective may be either very easy or very difficult to reach. As a fictional example, if you have an existing project with 1 configuration, reach 50% of improvement would just require to have one more configuration, while if you have 50 existing configurations, 50% would mean 75 configurations (i.e., 25 more). On the other hand, moving from 1 configuration to 2 is really the hard part since it means changing the build process and the build and deployment tools. Moving from 2 to 3 is then much easier. This is to say that, in order to provide a clearer vision of the progress, we've proposed, in those cases where it makes sense, to not only measure the percentage of improvement but also to report on the numbers of new items generated during the course of the project.

- KPI K07 was linked to task 3.2. “Log Data optimization for debugging”. As pointed out by the review report, the description of the task is not aligned with the results of the task. This comes from a (late) understanding that log filtering is trivial and does not represent the main issue for subsequent tasks. TU Delft is currently preparing an amendment to the DoW to redefine task 3.2 and realign this definition with the current state of the project. In consequence, KPI K07 is dropped as it is no longer meaningful.
- Last, we realized that one important characteristic brought by several tools (especially Descartes and DSpot) was not measured as a KPI: the improvement to the quality of the tests. Indeed, the main goal of Descartes and DSpot is to improve the quality of tests and thus to ensure that projects have tests that do what they’re supposed to do, i.e. detecting bugs and regressions.
- . Thus, we’re proposing to add a new KPI entitled “Better Test Quality”. This new KPI responds to objective 1 of section 1.1 of the DoW.

5. References

[0] Latest version of this document: d53_industrial_requirements_and_metrics_v2.pdf

[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

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 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 future collections for the validation reports at months 18, 30 and 36. The target is 40% improvement but computed as follows: a 40% improvement is a 40% reduction in the non-covered code. If you have 60% coverage now, the goal will be to reach 76%, not 84% and certainly not 100%.

Metrics

- Primary metric is global coverage.
- Secondary metrics are test coverages values brought by each tool individually (Descartes, DSpot, CAMP & EvoCrash).

Tools and strategies

- Use Clover as a preferred tool for the global coverage since it's easier with Clover to capture test coverage spanning several build modules and several executions.
- See WP1 deliverable (D1.2) which describes some Jenkins scripts to capture full coverage of test coverage data.
- In addition and since it's hard to compare Clover reports one with each other, a Groovy script was developed to harmonize the data before comparison. This script is also presented in D5.5.

Example on XWiki

- Capture of 2016-12-20: 74.07% --> XWiki should reach 84.4% to get a 40% reduction in non-covered code
- Capture of 2017-11-09: 76.28% --> Percentage improvement of 2.21% in under a year

STAMP tools helping improve the metric

- PIT/Descartes (by improving the test manually),
- DSpot (by generating more tests automatically),
- EvoCrash (by generating more tests semi-automatically),
- CAMP (test more paths thanks to various configurations)

Process to measure the global coverage

- Using Clover:
  - Set up Clover in your build
  - If your SUT consists of several separate builds, make sure to configure Clover to output data to a specific filesystem location so that each build will aggregate data in that location
  - Execute your build
  - Ask Clover to generate the report by pointing it to the specific filesystem location where data has been gathered, in order to generate a single coverage report
  - This allows to include contributions from functional tests, system tests and more generally to include contributions from a module onto other modules.
- Using Jacoco
  - Set up Jacoco in your build
  - If your SUT consists of several separate builds, use Clover instead (see above)
  - Execute your build
  - Ask Jacoco to generate the report

Process to measure impact from each tool

- For each test session:
  - Start by measuring and recording the current coverage for the target SUT (see above for how to measure the coverage)
  - Execute the STAMP tool and commit the test improvements brought by the tools
  - Measure again the current coverage for the target SUT and recording the coverage increase in percentage
- In order to provide a global tool contribution value per SUT, for each tool, sum up the coverage increases from each session.
- Note that for practical reasons it may not be possible to measure the full coverage for the full SUT for each test session (when the SUT is very large) and thus in these cases the coverage increase will be measured in sub-SUT only (code modules)
  - For example for XWiki: Fixing a test in module xwiki-commons/.../xwiki-commons-xxx would

require to measure test coverage before/after to compute the coverage contribution of this test to the overall coverage. Since the XWiki SUT is large, it would take well over 15 hours to do so, for each test session, which makes it impractical. Thus, the impact from each tool will be measured module per module (i.e. for xwiki-commons/.../xwiki-commons-xxx in this example).

Changes brought to KPI since D5.1

- Addition of secondary metrics to compute the coverage changes brought by each STAMP tool. This is interesting to gauge the effectiveness of each tool to increase the test coverage.

7.2. K02 - Less flaky tests

This is about dealing with tests which indeterminently and erroneously fail.

Definition: A test is considered flaky if it sometimes fails and sometimes passes, in a given environment (without the code under test nor the test being modified). Note that a test that passes in a given environment (e.g., in a given Database) but consistently fails in another environment (e.g., with a different Database) is not considered a flaky test.

Examples of flaky tests:

- Inadequate SUT internal state:
  - Test influence in SUT internal state: Inadequate test class (@BeforeClass) or test case (@Before) setups or cleanups (@Afterclass, @After) that interfere with other test classes, leaving the SUT in an inconsistent state. Tests run with wrong setups/cleanups should not be showing eventually different results, but considering that test case execution is random, they sometime succeed because the order favor their success, but under a different random execution, they failed. These are flaky tests according to the definition, but once their setups/cleanups are fixed, their flakiness disappear.
  - SUT internal state modified by external factors: sometimes flaky tests pop up because the SUT internal state is modified by external agents within a micro-service architecture (or other external services invoking the SUT). In other cases, other test classes executed within the test suite are modifying this internal state, which is not reset to the test baseline state), interfering with other test classes executed afterwards.
  - Unavailability/Modification of test inputs: sometimes some tests fail because the temporal unavailability of expected resources as input or intermediate work products.
  - Race conditions in multithreaded SUT. A typical example are functional UI tests not waiting for elements being present before asserting values (and those elements are modified by javascript)

These so-called "flaky tests" confuse developers and waste their time by indicating failures when there are none and can also delay the discovery of real bugs by "training" developers to ignore test failures. Thus it's important to identify and manage flaky tests so that they don't send false positives. This metric is about collecting the number of flaky tests identified and handled (and thus not failing the CI build anymore while at the same time being recorded, for example in an issue tracker, so that they can be fixed at a later stage). We're also hoping to reduce the number of flaky tests by 20% by the end the project.

Metrics

We record 3 metrics:

- Number of flaky tests identified and handled (and thus not failing the CI build anymore while at the same time being recorded, for example in an issue tracker, so that they can be fixed at a later stage)
- Number of flaky tests fixed during the period
- Percentage of flaky tests fixed compared to the total number of tests. This is computed with: % = FT * 100/TT, where:
  - FT (Flaky Total) = Number of not fixed flaky tests at the moment of capture
  - TT (Total tests) = Total number of tests at the moment of capture

Note that we record the "number of flaky tests fixed during the period" since this shows the effort done while the percentage value will depend on different factors and could be seen as not representative in some cases:

- If there are no existing flaky tests identified in the project, or a low number. In this case having a 20% increase is going to be relatively easy
- Since STAMP focuses on adding new tests to the projects, it's likely that flaky tests will also be included in the midst and thus the target percentage of 20% could be hard to reach in some cases

Tools and strategies

- See WP1 deliverable (D1.2) which describes some strategy to capture flaky test numbers, and to register them in the JIRA issue tracker
- Example on XWiki
  - Number of flaky tests identified and handled: 15 as of 2018-08-20. Note that the addition of the "Flickering Test" custom field in JIRA was done for STAMP and thus all issues listed correspond to work done to identify and handle flaky tests when working on STAMP
  - TT at start of STAMP: 9962.
  - TT as of 2018-04-03: 10508.
  - FT at start of STAMP: 5, before 2016-12-31.
  - FT as of 2018-04-03: 12.
  - Thus % at start = 5*100/9962 = 0.05% of flaky tests
  - % as of 2018-04-03 = 12*100/10508 = 0.11% of flaky tests
  - Analysis:
    - Before 2016 the XWiki project was not recording flaky tests in its CI, which accounts for the low number at the end of 2016.
    - By working on STAMP and increasing the number of tests we have, and especially functional UI tests, we also increased the number of flaky tests. The increase so far has been larger than our ability to fix flaky tests. Note that the positive aspect is that we're now handling flaky tests and they don't cause CI build failures and thus developers don't loose faith in CI and tests.

STAMP tools helping improve the metric

- Handle Flaky Test tool (XWiki Tool), see D1.2.

Process to recognize flaky tests

- Manually, by noticing tests failing and passing (in your CI for example)
- A good heuristic for detecting flakiness is executing tests 10 times. If one or more of the executions gives a different result for a test, it is likely to be flaky.
- Record them somewhere (suggestion: in an issue tracker)

Process to handle flaky tests

There are various solutions. Here are two:

- Solution 1: Mark the flaky test using the `@Ignored` JUnit annotation
- Solution 2: Keep the tests from executing but configure your CI jobs so that they don't send failure emails for false positives. This can be done with a Groovy post-build script or with a Jenkins Pipeline step.

Process to fix flaky tests

- Manually by doing an analysis/debugging of the flaky tests and fixing the problem.

Changes brought to KPI since D5.1

- Added one metric which is counting the number of flaky tests identified and handled.
- Added the metric about number of flaky tests fixed during the period to be fair and have some value of the work done even in case when it's easy to reach the % objective

- Improved the description of the 20% percentage improvement computation

7.3. K03 - Better test quality

This is similar to K01 which is using coverage as a measure of code quality. Here the goal is to increase the quality of the tests (and hence the quality of the code under test). Specifically, this can be achieved by computing the mutation score of tests and increasing this score. We're aiming to get an improved mutation score of at least 20% over the course of the project (for the same set of mutators).

Metrics

- Mutation score. Note that it could be hard and extremely long to measure a global mutation score (several days/weeks on the XWiki code base for example...). Thus the mutation score should be measured per module for practicability.
- Sub-metric: measure the number of pseudo tested + partially tested methods. These are the most interesting methods.

Tools and strategies

- Mutations to execute: ideally all the mutators defined by Descartes by default. If not, it's important to use the same set of mutators so that results can be compared across session measures.
- For each test session
  - Start by measuring and recording the current mutation score for the target SUT, using Descartes
  - Execute Descartes and/or DSpot and improve the tests
  - Measure again the current mutation score for the target SUT and record the mutation score increase in percentage

STAMP tools helping improve the metric

- PIT/Descartes (by improving the test manually),
- Dspot,
- CAMP (indirectly when committing new test cases - if the test cases (written as a consequence of CAMP finding issues) have a mutation score higher than average)
- EvoCrash (indirectly when committing new test cases - if the test cases (written as a consequence of EvoCrash reproducing bugs in production and after the bugs have been fixed) have a mutation score higher than average)

Changes brought to KPI since D5.1

- New KPI. It took the place of the previous K03 which was about “Faster Tests” and which has been included in K06 since it’s been retargetted for CAMP. See K06 for more details
- There was no KPI related to test quality and several of STAMP tools improve test quality, and thus it's interesting to measure their effectiveness in this area.
- This new KPI fulfills objective 1 of the DoW which is “Provide an approach to automatically amplify unit test cases when a change is introduced in a program”. Descartes and DSpot are the main tools which will participate to increasing this KPI and they’re both about improving test quality. Thus we still have 3 KPIs fulfilling objective 1 (K01, K02 & K03).

7.4. K04 - More unique invocation traces

This is a measure of the number of unique invocations between services thanks to executing it under various configurations.

For micro-services projects

This metric is useful for applications with the distributed architecture where an inter-process communication (IPC) may take place between various components, e.g., a typical client-server architecture. Each component may combine both client and server. IPC can be realized via remote procedure calls (RPC), http requests, publish-subscribe models etc.

A side effect of this exercise could be an increase of test coverage, since the application code can contain if-else clauses for a specific configuration, e.g., different databases.

The objective of the project is to increase the number of unique invocation traces by 40%.

Process for micro-services projects

To measure this metric, application code should be instrumented to capture cause-effect relationships between components of the system, e.g., OpenTracing or AOP with AspectJ or other. The cause-effect relationship can also be inferred and therefore it does not require any instrumentation of the code. We also require a third-party that allows you to capture and analyze these relationships at runtime which we call traces, e.g., Jaeger. By executing an existing test suite, we can gather a set of traces which constitute a cause-effect relationship between services. Generating a new configuration and executing the same test suite would allow us to gather another set of traces which should cover other services.

For monolithic projects

Even though the spirit of this metric is especially true for micro-services projects, it can be made to work for monolithic projects by computing the additional test coverage resulting from executing the tests under various configurations. This will demonstrate the extra paths related to configurations and thus increase the coverage.

The objective of the project is to increase the code paths (i.e., coverage of the related modules) by 20%. It should be noted that the coverage increase will depend on the architecture of the software (for example a software using an ORM library will not have code that depends on the database used and thus various DB configurations won't increase the coverage).

Process for monolithic projects

- Generate a Clover report with a default configuration. Note down the global test coverage
- Make sure to configure Clover with a defined output directory so all traces are logged in a single place
- Then, run the tests again, but under the various configurations, using CAMP or TestContainers. This will accrue the Clover logs in the defined output directory.
- Regenerate the Clover report and find the new global test coverage. The difference with the original test coverage will provide the coverage increase.

Important note: For monolithic project (e.g., XWiki), it's not expected that the measure will result in a great coverage increase since XWiki uses lots of third party technology to abstract the environments/configurations (e.g., using Hibernate to abstract DBs and Servlet API to abstract Servlet Containers).

STAMP tools helping improve the metric

- CAMP
- XWiki's experiment based on TestContainers

Changes brought to KPI since D5.1

- Added the support and target for monolithic projects since the KPI was originally only targeted to micro-services type of projects

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". To identify system-specific/configuration-specific bugs, first you must define the matrix of supported configurations (For Example: [Firefox, IE, Chrome] + [Oracle, MySQL, Postgres] + [Tomcat, Jetty]). Since bugs from these configurations are identified over time, the first field is the timespan in which bugs are reported and the second field is the number of bugs which were identified in that timeframe. The only bugs which should be reported here are those which were 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, which means finding 30% more configuration-related bugs than what existed before STAMP.

Metrics

- Metric #1: Number of new configuration-related bugs discovered
- Metric #2: Percentage improvement vs. the baseline

Note that we record the number of new configuration-related bugs discovered since this shows the effort done while the percentage value (30% improvements) will depend on the current number of known issue related to configurations, and depending on the use case/project this can be a low value and thus it could be relatively easy to come up with a large % value.

Tools and strategies

- Initial measure: Identify in issue tracker number of bugs related to configurations
  - Example (XWiki): Find all issues labelled with "mysql", "postgresql"
  - Note that if the project doesn't have a pre-existing issue tracker with such information, it's also possible to use CAMP to generate a baseline by recording all bugs from configurations.
- Run CAMP, execute existing tests and find new bugs related to configurations. Record them
- Compare with the original baseline and generate a % improvement
- Note: each industrial partners needs to define what they mean by different configs.

STAMP tools helping improve the metric

- CAMP

Changes brought to KPI since D5.1

- Added metric about counting the number of new configuration-related bugs discovered

7.6. K06 - More Configuration/Faster Tests

The objective is to be able to execute more tests than before per amount of time, especially when comparing with manual tests. The main scope for this metric is configuration testing and being able to test automatically more configurations, thus allowing to get more value per unit of time compared to the traditional strategy of running configuration tests manually under various environments. The secondary scope is to decrease the time it takes to deploy the software under various configurations. Globally the goal is to increase the number of automatically tested configurations by 50%.

Metrics

- Metric #1: Number of new configurations tested. Any significant change in a configuration file, in the number of micro-service instances or in the infrastructure used to run the system constitutes a new configuration.
- Metric #2: Time it takes to deploy the software under a given configuration vs. time it used to take when done manually before STAMP

Note that we record the number of new configurations tested since this shows the effort done while the percentage value (50% more configurations) will depend on the current number of configurations being tested and this can be a low value, thus making it relatively easy to reach the objective. Still it's important to note that moving from one configuration being automatically tested to 2 is a major achievement since it means automating the whole process of testing on various configurations (build changes, CI changes, provisioning process, creation of Docker images, etc). Moving from 2 to N configurations is comparatively easy.

STAMP tools helping improve the metric

- CAMP
- XWiki's experiment based on TestContainers

Strategy example (XWiki)

- Right now the automated XWiki tests are being executed only in a single environment (1 configuration), namely: XWiki deployed in latest version of Jetty, using the latest version of HSQLDB and on the latest version of Firefox.
- However there are manual tests existing, that are performed on various configurations but in adhoc manner (i.e., selecting tests here and there since it's impossible to run a substantial number manually for lack of manpower). However the automated tests are not executed on those configurations.
- For the purpose of STAMP we should assume that XWiki is currently tested only on one configuration since that's the number of configurations on which all the automated tests execute.
- The goal is to run those tests on more configurations and have them executed in the CI too: Tomcat, MySQL, PostgreSQL, Chrome, Firefox, etc.
- Thus for Metric #1 we expect to have test configurations matching the supported XWiki configurations (between 4 to 8):
  - https://dev.xwiki.org/xwiki/bin/view/Community/BrowserSupportStrategy
  - https://dev.xwiki.org/xwiki/bin/view/Community/DatabaseSupportStrategy
- For metric #2, we developed XWiki images during STAMP and went from 2 hours to 5 minutes
  - Date of collection: 2017-01-01
    - Time to download the XWiki WAR + time to download a DB (MySQL or PostgreSQL) + time to download a servlet container (Tomcat, etc) + time to set it up = 2 hours
  - Date of collection: 2017-02-22
    - Install Docker once and use the XWiki Docker image (for MySQL + Tomcat) = 5mn

Strategy example (Atos)

- Compute baseline: given a a) SUT, Docker enabled (Docker image(s) for SUT available, a Docker compose configuration available in case on multi-container SUT), b) a test suite, do the following for a given number of target configurations:
  - a- generate a new configuration as a variant of the given one as input (Docker compose, Dockerfile)
  - b- instantiate the SUT using the generated configuration
  - c- conduct the test suite against the SUT
  - Iterate until the number of target configurations or a time limit has been reached. Collect test results for configuration. Consolidate them.
  - This process can be done (semi)automatically using the state of the practice
- Using CAMP automate the test execution (see above bullet) on the SUT for each generated configuration. Collect test results for configuration. Consolidate them.
- Compare results: for both 1) and 2), compare the time require to complete each test (total time) until reaching the number of target configurations. Alternatively, for a given time limit, compute the number of SUT configurations that could be tested using the given test suite.

Changes brought to KPI since D5.1

- Renamed from "Faster Tests" to "More Configuration/Faster Tests" since there are no tools developed during STAMP that would increase speed of tests. Quite the opposite, mutation testing and advanced tools being developed reduce the speed of the tests, with the added value of increasing test value and quality. Thus we refocused this KPI more towards counting the additional configurations being tested automatically thanks to CAMP. Note that this has an impact on execution speed since before STAMP the projects were testing various configurations manually and in adhoc manners, thus taking a lot more time than an automated execution.
- Note that we've also kept the original K06 by keeping the deployment time metric. However it's a secondary metric since it's something relatively static that doesn't need to be measured continuously

(it's a one time effort) and it's a prerequisite for being able to automate testing on various configurations (the primary metric).

7.7. K07 - Shorter Logs

- KPI K07 was linked to task 3.2. "Log Data optimization for debugging". As pointed out by the review report, the description of the task is not aligned with the results of the task. This comes from a (late) understanding that the log filtering is trivial and does not represent the main issue for subsequent tasks. TU Delft is currently preparing an amendment to the DoW to redefine task 3.2 and realign this definition with the current state of the project. In consequence, KPI K07 is dropped as it is no longer meaningful.

7.8. K08 - More crash replicating test cases

This is a metric which represents the number of automated test cases that replicate "crashes". In Java software, a "hard crash" (e.g., Segmentation Fault) is exceedingly rare and therefore a "crash" is defined as an unexpected exception/error behavior, for example that which aborts the process of rendering a webpage for a user request. 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 field in this section 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: As the occurrence of actual crashes (segmentation fault, etc) in Java is extremely low, we define crash to mean an unexpected exception/error behavior, for example that which aborts the process of rendering a webpage for a user request.

Tools and strategies

- Count the number of existing stack traces that have been fixed already and that have automated tests written to prove that the problem was fixed.
  - To do this, use the project's tracker to search for stack traces for closed issues
  - From the list, only keep the issues that have tests
  - Let's call this number FT (Fixed with Tests)
- For example for XWiki:
  - JQL (JIRA query): `description ~ '.java:' AND description ~ 'Exception' AND category = "Top Level Projects" AND createdDate >= '-365d' and resolution not in ("Cannot Reproduce", Duplicate, Inactive, Incomplete, Invalid, "Won't Fix") ORDER BY created DESC`
  - Date of collection: 2016
    - Number of issues with stack traces (after removing the false positives): 17
    - Number of issues with stack traces and tests to prove the fix: 3
    - % test suite = 3 / (total test number) = 3 / 9997
      - Total test number: 9997
- Note: For projects not having pre-existing issues in a tracker, it'll be necessary to generate issues. This can be done for example by looking at production logs and finding stack traces in them.
- Then for all issues that are still open and that have stack traces and for all issues closed but without tests, run EvoCrash on them.
- Count the number of successful EvoCrash-generated test cases. Let's call this number: ET (EvoCrash-generated Tests)
- Compute the percentage improvement as follows: % = ((FT+ET)/FT – 1)*100
  - Example with FT = 50 and ET = 35 (35 tests reproduced by EvoCrash)
  - % = ((50+35)/50 - 1)*100 = 70%

STAMP tools helping improve the metric

- EvoCrash

Changes brought to KPI since D5.1

- No change

7.9. K09 – More production level test cases

This metric measures the percentage of test cases in a test suite that have been built, based on the behaviors observed during software execution. The objective is to amplify the test suite with regression tests checking that the current usage of the system is ensured by future evolutions.

The objective of the project is to enhance existing test suites with 10% of production-level test cases.

Note

- This metric cannot be evaluated for the moment since it's related to task 3.4 which starts at M24 (and goes till M32).
- A new tool or a new feature of EvoCrash will be developed. Since we won't know before the work starts, we've mentioned EvoCrash as the STAMP tool helping improve this metric below.
- It's also expected that this KPI definition will become more accurate after the work for it has started and especially after the discussions between the tool developers and the industrial partners have taken place.

Tools and strategies

- Count the total number of existing test: *T*
- Count the number of production level tests generated by EvoCrash: *P*
- K09 = *P* / *T*

STAMP tools helping improve the metric

- EvoCrash (new feature)

Changes brought to KPI since D5.1

- There was a misunderstanding when D5.1 was delivered and what written back then didn't correspond to what was in the DoW. Thus we've put back the original version of this KPI as it was defined in the DoW.

8. KPIs vs Tools

List of tools covered by each KPI:

Tool / KPI	**K01**	**K02**	**K03**	**K04**	**K05**	**K06**	**K07**	**K08**	**K09**
Descartes	Yes		Yes						
DSpot	Yes		Yes						
CAMP	Yes			Yes	Yes	Yes			
EvoCrash	Yes						Yes	Yes	Yes
Flaky Script		Yes							

9. KPIs vs Use Cases

List of use cases measuring each KPI:

UC / KPI	K01	K02	K03	K04	K05	K06	K07	K08	K09
AEon	Yes	Yes	Yes	Yes	Yes	Yes	KPI dropped	Yes	(Yes)
ATOS	Yes	Yes	Yes	Yes	Yes	Yes	KPI dropped	Yes	(Yes)
OW2	Yes	(Yes)	Yes	Yes	Yes	Yes	KPI dropped	Yes	(Yes)
TellU	Yes	Yes	Yes	Yes	Yes	Yes	KPI dropped	Yes	(Yes)
XWiki	Yes	Yes	Yes	Yes	Yes	Yes	KPI dropped	Yes	(Yes)

Notes:

- For K02, OW2 is still looking for a project having Flaky tests, in order to be able to measure K02.
- For K09, the KPI is not fully defined yet (will be once task 3.4 has progressed enough, starts at M24)

10. Use Cases & Measures

- D5.1 was providing a first definition of the use cases (UC) and some baseline measures.

Industrial UC definitions are now given in D5.4 (also in D5.5).

- Baseline KPI measurements are reported in D5.5 and updated versions will be reported in D5.6 & D5.7 since the baseline measurements are computed as part of the evaluation activities.

11. Conclusion

The definition and relevancy of the KPIs have been adjusted to match better the tools developed and so that they can be measured, while retaining to the maximum the original intents and targets/objectives. In addition we've added new metrics and new KPIs where we thought that they were missing (e.g., measuring test quality). Here is an updated version of the KPIs vs Objectives table you can find in the DoA, for KPIs which are in the scope of this document. The changes are in blue.

Objective							KPIs
ID	1	2	3	4	5	6	
KPI1	X						Increase the diversity of execution paths covered by 40% Secondary metrics to compute the coverage changes brought by each STAMP tool.
KPI2	X						Decrease by 20% the number of flaky tests Metrics to count the number of flaky tests identified and handled, and to count the number of flaky tests fixed
KPI3	X						~~Increase by 20% the number of lines of product code, which are executed for each second of time spent running tests.~~ Improve mutation score of at least 20% over the course of the project.

KPI4		X					Increase by 40% the number of unique invocation traces between services in a global perspective Measure this KPI also for monolithic projects
KPI5		X					Increase by 30% the number of valid bugs detected during testing which are specific to the generated configurations Metric to count the number of new configuration-related bugs discovered
KPI6		X					Increase the number of automatically tested configurations by 50%. Reduce by 30% the time on configuring and deploying products for testing purpose: compare manual testing with automated testing
~~KPI7~~			~~X~~				~~Reduce the size of log files by an order of magnitude, keeping all essential information~~ Dropped
KPI8			X				Increase by 70% the number of crash replicating test cases
KPI9			X				Enhance existing test suites with 10% of production-level test cases

Next steps:

- Use case providers need to take a new baseline, using the updated metrics, and report this as part of D5.6.
- Whenever a measure is taken, the use case providers must ensure to record the session times and importantly define precisely the scope of the SUT (System Under Test), and ensure that all future measures use the same SUT for a given KPI.