



Title	WP5 – D5.6 – Use Cases Validation Report V2
Date	November 30, 2018
Writers	Pascal Urso, Activeeon Pedro Velho, Activeeon Mael Audren, Activeeon Jesús Gorroñoitía, ATOS Fernando Mendez, ATOS Lars Thomas Boye, Tellu Kenneth Skaar, Tellu Vincent Massol, XWiki Assad Montasser, OW2 Cédric Thomas, OW2
Reviewers	Benoit Baudry (KTH), Valentina Di Giacomo (ENG)
Online version	https://github.com/STAMP-project/docs-forum/blob/master/docs/d56_uc_validation_report.pdf



Table of Content

1. Executive Summary.....	4
2. Revision History.....	5
3. Objectives.....	5
4. Introduction.....	6
5. References.....	6
6. Validation Questions.....	7
7. Activeeon Use Case Validation.....	9
7.A. Use Case Description.....	9
7.B Validation Experimental Method.....	11
7.C Validation Results.....	17
8. Atos Use Case Validation.....	21
8.A. Use Case Description.....	22
8.B. Validation Experimental Method.....	26
8.C Validation Results.....	34
9. Tellu Use Case Validation.....	45
9.A. Use Case Description.....	45
9.B. Validation Experimental Method.....	47
9.C. Validation Results.....	52
10. XWiki Use Case Validation.....	60
10.A. Use Case Description.....	60
10.B. Validation Experimental Method	62
10.C. Validation Results	73
11. OW2 Use Case Validation.....	86
11.A. Use Case Description.....	86
11.B. Validation Experimental Method.....	88
11.C. Validation Results	91
12. Threats to Validity.....	97
13. Validation Summary.....	98
13.A Validation Questions Synthesis.....	98
13.B. KPI Synthesis	99
13.C. Quality Assessment.....	102
14. Conclusion.....	105

List of Tables

Table 1: Use Case Experimentations during Period 2.....	4
Table 2: Key Metrics of Activeeon's Use Case Target Software.....	9
Table 3: Tasks for Measuring Metrics for Activeeon Validation Experimentation.....	12
Table 4: PWS Catalog Project Scores.....	17
Table 5: Roadmap for DSpot within Activeeon's Use Case.....	19
Table 6: Roadmap for Descartes within Activeeon's Use Case.....	20
Table 7: Roadmap for CAMP within Activeeon's Use Case.....	21
Table 8: Roadmap for Botsing within Activeeon's Use Case.....	21
Table 9: SUPERSEDE IF Key Code Metrics.....	23
Table 10: Control Tasks for Atos Validation Experimentation.....	27
Table 11: Treatment Tasks for Atos Validation Experimentation.....	28
Table 12: Atos Experiments for KPIs. Sequence of Experimentation Tasks.....	32
Table 13: STAMP Github Repositories and Paths for Reporting Evaluation Experimentation Data.....	33
Table 14: Atos Code Base Repositories for Use Cases (SUPERSEDE IF and CityGO CityDash).....	34
Table 15: K01 Code Coverage Metrics for IF Use Case.....	34
Table 16: K02 Code Coverage Metrics for IF Use Case.....	35
Table 17: K03 Code Coverage Metrics for IF Use Case.....	35
Table 18: Stress Tests Statistics on CityDash for Different CAMP Generated Deployment Configuration.....	37
Table 19: K04 Increment in Invocation Traces Metric for CityDash Use Case.....	37
Table 20: K05 Number of Configuration Specific Bugs Metric for CityDash Use Case.....	38
Table 21: K06 Metrics for CityDash Use Case.....	38
Table 22: Common Runtime Exceptions in IF Use Case.....	38
Table 23: K08 Metrics for IF Use Case.....	39
Table 24: Quality Model Based Evaluation of Descartes Tool.....	39



Table 25: Quality Model Based Evaluation of DSpot Tool.....	41
Table 26: Quality Model Based Evaluation of CAMP Tool.....	41
Table 27: Quality Model Based Evaluation of Botsing Tool.....	42
Table 28: Key Metrics of Tellu's Use Case Target Software.....	46
Table 29: Tasks for Measuring Metrics in Tellu Validation Experimentation.....	48
Table 30: TelluLib Project Metrics.....	53
Table 31: Core Project Metrics.....	54
Table 32: FilterStore Project Metrics.....	54
Table 33: Controls and Treatments for KPIs for XWiki Software.....	66
Table 34: Validation Target Objects for XWiki Software.....	67
Table 35: Validation Tasks for XWiki Software.....	69
Table 36: Supported Configuration Types.....	70
Table 37: Validation/Measurements Methods for XWiki.....	73
Table 38: K01 Metric Measures for XWiki.....	74
Table 39: Coverage and Mutation Score Contributions by Descartes Strategy.....	75
Table 40: Coverage and Mutation Score Contributions by DSpot Strategy.....	76
Table 41: Flaky Test Measurements.....	77
Table 42: Configurations Tested.....	80
Table 43: Floating Configurations Implemented.....	80
Table 44: Coverage Improvement Thanks to Configuration Testing.....	80
Table 45: System-Specific Bugs Found Thanks to Configuration Testing.....	80
Table 46: Number of New Configurations Tested.....	81
Table 47: Percentage of Crash-Replicating Test Cases.....	82
Table 48: Key Metrics of OW2's Use Case Target Software.....	87
Table 49: OW2 Experimentation Environment.....	87
Table 50: OW2 Validation Treatments.....	89
Table 51: DSpot Experimentations on Sat4j.....	90
Table 52: OW2 K01 & K03 with DSpot (Control).....	92
Table 53: OW2 K01 & K03 with DSpot (Treatments).....	92
Table 54: DSpot Results on Sat4j.....	92
Table 55: OW2 K01 & K03 with Descartes (Control).....	93
Table 56: OW2 K01 & K03 with Descartes (Treatments).....	94
Table 57: Descartes Results on Sat4j.....	94
Table 58: OW2 Roadmap for VQ3 and VQ4.....	97
Table 59: K01 Progress Overview.....	99
Table 60: K02 Progress Overview.....	100
Table 61: K03 Progress Overview.....	100
Table 62: K04 Progress Overview.....	100
Table 63: K05 Progress Overview.....	101
Table 64: K06 Progress Overview.....	101
Table 65: K08 Progress Overview.....	101
Table 66: Global Quality Model Applied by the Use Cases.....	102

List of Figures

Figure 1: Proactive Workflow & Scheduling (PWS) Development Chain.....	10
Figure 2: PWS CAMP yaml Configuration Example.....	14
Figure 3: CityGO CityDash Key Code Metrics.....	23
Figure 4: Docker Compose Configuration for City Dash Postgresql and Apache Web Server: Performance Configuration.....	29
Figure 5: CityGO CityDash CI/CD Workflow in Jenkins.....	30
Figure 6: Performance Results of Stress Tests Applied by JMeter in Jenkins CI/CD to CityGO CityDash.....	30
Figure 7: Some Metrics for XWiki.....	61
Figure 8: Global Coverage Contributions Module by Module for XWiki.....	69
Figure 9: XWiki's Testing Framework Architecture.....	69
Figure 10: Sample Snapshots of Sat4j Reports Table.....	91
Figure 11: DSpot Quality Assessment Evolution.....	103
Figure 12: Descartes Quality Assessment Evolution.....	103
Figure 13: CAMP Quality Assessment Evolution.....	103
Figure 14: Botsing Quality Assessment Evolution.....	103

1. Executive Summary

The STAMP test amplification technologies developed by research partners are being validated by industry partners in use cases representative of different business-oriented implementations. Use cases are provided by following partners: two software vendors, Activeeon and XWiki, a systems integrator - ATOS, a cloud service provider - Tellu and an open source organisation - OW2.

All the STAMP tools have been experimented by several, if not all use case partners as shown in the table below. Use case providers have applied the STAMP tools, namely DSpot, Descartes, CAMP and Botsing, to their target software, according to the "Validation Roadmark and framework V1" described in deliverable D5.2, in ongoing interaction development partners.

	Descartes	DSpot	CAMP	Botsing
Activeeon	√	-	-	√
Atos	√	√	√	√
Tellu	√	√	√	√
XWiki	√	√	√	√
OW2	√	√	-	-

Table 1: Use Case Experimentations during Period 2

All the use case partners have experimented the STAMP tools carrying out real-life evaluations in their specific technical settings and from the perspectives of their own business processes. The contexts of the different use cases vary greatly and while this is a definite strength of the validation activity, all use case do not progress at the same pace due to the constraints of their specific business conditions.

The use case partners have defined a common approach for their validation activities. This approach is based on three pillars: a) a recognized experimentation methodology, b) a set of four Validation Questions jointly defined to guide the validation activities, c) a quality model to help assess the tools from a user-experience perspective. Use case have developed special efforts to measure quantitative results obtained with the STAMP tools. Moreover each use case conducted several iterations in their experiments so as to understand the fitness of the tools for their own business purpose and to explore how the tools can be fine-tuned to their needs.

Each use case is reported with the same structure as follows:

- **A- Description:** this part re-uses some elements already presented in D5.5, it introduces the context of the use case, the target software used for the experimentations, the technical environment, the benefits that are expected and the business relevance of the use case for the industrial partner.
- **B- Validation experimental method:** this part describes validation-related activities how the experiment is set up and conducted and how the baseline ("control" values) and how experimental data was sampled, collected during experimentation.
- **C- Validation results:** the third part presents the main validation findings, the results associated to the KPIs: variable measurements, the answers to Validation Questions and the quality assessment.
- **D- Next steps for validation:** the use case being work in progress, this last part outlines the validation activities for the next period.

After the individual description of each use case, the report provides a cross-use case summary of the results. In a nutshell the main findings of this period of testing the STAMP tools can be summarised as follows.

- **Descartes is highly praised by all use case partners.** A rather mature tool, Descartes is a good candidate to be integrated in the software development process. It is perceived, for instance, as a tool well

suited to be used by developers at the time of writing tests or, better, to be automatically launched to test newly written tests. A good exploitation strategy would include the promotion of Descartes plug-ins for popular IDEs such as Eclipse and IntelliJ IDEA.

- **All use case partners report improvements in their test suites and in the confidence of their own tests** after using the STAMP tools. The improvement is derived from the actual results provided by Descartes' ability to identify pseudo and partially tested methods and by CAMP facilitating the test of a greater number of execution configurations prior to production. This perception is supported by great expectations as well on DSpot automatic test generation and Botsing crash replications.

- **Development operations have clear feedback from the use cases.** With the tools being actually tested by industry partners with real life software, a threshold has been crossed in the development operations. This period has seen a significant increase in the interaction between use case partners and the tools development teams. Use case partners have been able to provide pragmatic feedback and share their needs and expectations based on business-driven experiments. Thanks to this interaction, all use case have a clear roadmap for the continuation and completion of their experiments.

2. Revision History

Date	Version	Author(s)	Comments
25-Sept-18	0.1	Jesús Gorroñoigoitia, ATOS	ToC
16-Nov-18	0.2	Mael Audren, Activeeon Jesús Gorroñoigoitia, ATOS Lars Thomas Boye, Tellu Vincent Massol, XWiki Assad Montasser, OW2 Cédric Thomas, OW2	First draft
27-Nov-18	0.3	Cédric Thomas, OW2 Benoit Baudry (KTH) Valentina Di Giacomo (Eng)	Review and clean-up

3. Objectives

This document reports on the validation activities of the STAMP tools performed by the use case as set out in the STAMP "Validation Roadmap and Framework". The objectives of the validation activities are as follows:

- To provide an assessment of the functionality and usability of the STAMP techniques and toolset.
- To provide an initial empirical assessment on the practical use of the current STAMP toolset.
- To involve real life developers in the development process of the STAMP toolset, with the aim to align its features and usage with their industrial needs.

The validation objectives are based on the project objectives set down in the Description of the Action. The primary validation objective is: Validate the relevance and effectiveness of amplification on 5 Use cases. This in turn refers to the fulfillment of Objectives 1, 2 and 3 elicited in the DoA. The fulfillment of the objectives is measured against technology Key Performance Indicators (KPIs) also provided in the Description of the

Action. The technology KPIs have been refined and detailed in deliverable D5.3.

4. Introduction

This document reports on the second period of the STAMP project use cases. The first period ran from M3 to M18, this second period, much shorter, runs from M19 to M24. The validation efforts are defined by work package 5 in tasks 5.3 to 5.7. The STAMP test amplification technologies developed by research partners are validated by industry partners in Use cases representative of different business-oriented implementations. These use cases form the backbone of WP5. The work package is also responsible for establishing the industrial requirements and metrics for validation, for defining the validation roadmap and framework and conducting the actual validation activities. Previous deliverable D5.1 and D5.2 determined the context in which the validation activities are conducted, deliverable D5.3 and D5.4 outlined the validation roadmap and framework. The validation activities conducted from month 1 to 18 were reported in deliverable D5.5; the present deliverable D5.6 provides updated results obtained during the M19-M24 period.

The use cases are conducted by five project partners, including two software vendors - Activeeon and XWiki, a systems integrator - ATOS, a cloud service provider - Tellu and an open source organisation - OW2. They constitute a representative sample of complementary industry perspectives and are all committed to provide enterprise-grade quality software. The evaluation conducted by industry partners provides direct feedback to the scientific and technical development conducted in WP1-WP4. The partners conducting validation activities follow different software engineering processes and different requirements and expectations about the STAMP outcomes. This diversity ensures a balanced coverage for the validation of the STAMP outcomes. However, in order to ensure consistency of the validation activities, a common quality model and validation framework is established.

The bulk of the deliverable is comprised of the use cases presented individually. While all use case are reported following the same structure, the diversity of the experimentations and of the authors make for an unavoidable diversity in both style and volume of the use case monographies. Several sections present synthetic results. The high-level structure of the report is thus as follows:

- presentation of the validation questions that serve as guidelines for the validation activities.
- presentation, use case by use case, of the context, the validation methods and the results obtained and the lessons derived from the hands-on validation activities.
- presentation of the threats to validity of these validation activities
- summary of the assessment..

5. References

This section provides a list of references cited in this document.

- D5.1: [Industrial requirements and metrics for validation V1](#)
- D5.2: [Validation roadmap and framework V1](#)
- D5.3: [Industrial requirements and metrics for validation V2](#)
- D5.4: [Validation roadmap and framework V2](#)
- D5.5: [Use Cases Validation Report V1](#)
- Fraser, G., & Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2), 8.

6. Validation Questions

This section outlines the validation objectives as already described in previous deliverable, we introduce the validation questions we have defined to guide the validation activity. Validation questions are like “research

questions” except more adapted to the practical context of use cases.

Validation objectives are specified in Chapter 5 of deliverable D5.4: “to conduct a fit-for-purpose evaluation that assesses the STAMP toolset and services utility in real life scenarios that requires test amplification” and “to assess whether the STAMP scientific and technical achievements satisfy the needs and expectations of users”. We are now reporting on phase 2 of the validation roadmap described in Chapter 8 of deliverable D5.4: this phase “focuses on validating the first stable STAMP toolset, focusing of their usability, effectiveness, efficiency and the perceived fit-for-a-purpose benefit in a variety of industrial real scenarios”.

Keeping in mind the above references for validation goals and objectives, we have defined a set of validation questions (VQs) for this phase of the validation. These questions elicit some hypotheses about the benefits of STAMP tools for the use cases and guide the validation activity. The VQs are all related to KPIs.

VQ1 - Can the STAMP tools assist software developers to cover areas of code that are not tested?

Testing as much as possible of a software behavior is of paramount importance. Yet, when a program is already well tested, the real challenge is to remove the amount of untested code¹. It is very challenging to write test cases that reach the remaining 10 or 20% of code that are not covered by existing tests. The software engineers that provide the STAMP use cases argue that the main goal of code coverage is to find untested code, that is where the real value is. The goal of this validation question is to assess the capacity of STAMP tools to assist developers to develop new test cases covering parts of the code that are not well tested.

Related KPIs:

- K01 - More execution paths: Code coverage is measured, including the contribution made by each STAMP tool on each use case. This is the primary metric for answering VQ1.
- K04 - More unique invocation traces: In addition to general code coverage, we look at to which extent configuration test amplification can help test more paths through the system under test. This metric is more use case dependent, with the relevance and methodology varying with use case.
- K08 - More crash replicating test cases: STAMP tools may be able to create tests which reproduce runtime crashes not detected beforehand by the existing test suites. These new tests can cover code paths not covered before.

This question relates to STAMP Objective 1: *Provide an approach to automatically amplify unit test cases when a change is introduced in a program.*

VQ2 - Can STAMP tools increase the level of confidence about test cases?

This question addresses two complementary aspects of confidence about test cases, which are both addressed by STAMP tools and KPIs. First, confidence relates to the issue of false positives, where tests fail without actual errors in the code. This leads to loss of confidence in tests, and may lead to test results being ignored. Second, confidence relates to the test verdict: to what extent can we be sure that the code executed by tests is indeed correct? This is a harder issue to address than false positives, since the tests do not indicate any problem (as an extreme case, it would indeed be possible to have full coverage of the code without actually testing anything, simply running the code without assessing it!).

Related KPIs:

- K02 - Less flaky tests: Flaky tests is a form of false positives, where the test sometimes fails and sometimes passes, without changes in the SUT configuration, internal or external (environment) state or test. Metrics related to flaky tests will be collected to help answer VQ2.
- K03 - Better test quality: The metric here is mutation score. This score tells us to which extent changes to the code is detected by tests. The STAMP tool Descartes finds pseudo-tested and partially tested methods - where removal of the method is not at all or only partially causing tests to fail, indicating that tests do not (fully) assert the correctness of the method. We will investigate to which extend knowledge of and fixing of such issues increase test confidence.

¹ <https://martinfowler.com/bliki/TestCoverage.html>



This question relates to STAMP Objective 1: *Provide an approach to automatically amplify unit test cases when a change is introduced in a program.*

VQ3 - Can STAMP tools increase developers confidence in running the SUT under various environments?

This validation question addresses dependency on the environment. To what degree will the tested software run correctly in various real deployments? The exact nature of the execution environment depends on the use case - on the nature and scope of the software and how broad or narrow the potential or intended set of target environments are. Yet, all software have external dependencies, whether it be Java VMs, operating systems, databases, networks, messaging protocols, etc., and must handle at least some variability.

Related KPIs:

- K04 - More unique invocation traces: We check if STAMP tools contribute to execute more paths through the SUT by varying configurations.
- K05 - System-specific bugs: We check if STAMP tools help find bugs which only occur on specific configurations.
- K06 - More Configuration/Faster Tests: We check if STAMP tools help run the SUT on more configurations.
- K08 - More crash replicating test cases: If configuration testing amplification produces new crashes, STAMP tools may also be able to create tests which reproduce such crashes.

This question relates to STAMP Objective 2. *Provide an approach to automatically generate, deploy and test large numbers of system configurations.*

VQ4 - Can STAMP tools speed up the test development process?

This question addresses the extent to which the STAMP tools can assist software developers in order to reduce the time they spend in developing and executing test cases. Here we assess the capacity of the STAMP tools to speed up the development of two specific types of test cases: configuration-dependent test cases and crash replicating test cases.

Related KPIs:

- K06 - More Configuration/Faster Tests: The objective is to be able to execute more tests than before per amount of time.
- K08 - More crash replicating test cases: Generation of crash replicating test cases will be evaluated against manual creation of such test cases.

This question relates to STAMP Objective 3. *Provide an approach to automatically amplify, optimize and analyze production logs in order to retrieve test cases that verify code changes against real world conditions.*

Each of the five use case partners use the STAMP tools on their use case(s), measuring the KPIs, in order to answer the validation questions for their use case(s). The different use cases have different secondary goals contributing to their methodology and answers. These are described in the use case descriptions. Based on the conclusions found in each use case, we will find general conclusions for the validation questions.

7. Activeeon Use Case Validation

Activeeon Use Case Highlights

- Descartes tool integration and results in ProActive Catalog project.
- Botsing integration with ProActive Scheduling project and stack trace analysis.
- Development of the Gradle plugin required for conducting the Botsing evaluation

7.A. Use Case Description

7.A.1 Target Software

The target software of the use case is ProActive Workflows & Scheduling (PWS), a software system that supports the execution of jobs and business applications, the monitoring of activity and the access to job results. Allowing to scale up or down applications adapting to the workload, PWS optimizes the match between availability and cost. It ensures more work done with fewer resources, managing heterogeneous platforms and multiple sites with advanced usage policies.

PWS is a distributed system built upon a microservice architecture pattern. PWS is developed and maintained by a team of 15 developers. The primary software language used to develop the backend is Java, and for the frontend, javascript. As a first goal we focus on two projects, Scheduling and Catalog.

In Table 2 we have **Scheduling** and **Catalog** projects, primary focus for the STAMP evaluation KPIs and validation questions. **Catalog** has a backend and frontend that enable to store entities used in PWS. Featuring a single module, it can be compiled in one go with a simple Gradle build. **Scheduling** provides the main core functionalities of PWS including the load balancer and a frontend to manage jobs and tasks. It features 8 modules, more than a thousand of unit tests. It also is the main focus for integration and system tests. Each module inside the **Scheduling** project builds separately. For those reasons, **Scheduling** is more complex but also can take the best outcome from STAMP. For the sake of simplicity, we focus on **Catalog** during the first phase. Its simpler structure and smaller test base will provide us a good challenge to put STAMP tools into practice and still produce relevant results.

	Scheduling	Catalog
Number of Java classes (Sonarqube)	1102	142
Lines of code (Sonarqube)	90,085	5866
Code Coverage	(23.6%)	64.8%
Unit tests (Sonarqube)	1307	262
Mutation Score (Descartes)	Not applied	35%

Table 2: Key Metrics of Activeeon's Use Case Target Software

7.A.2 Experimentation Environment

For continuous integration, we use Jenkins to launch the jobs that build and test PWS. 150 jobs are defined in the Activeeon Jenkins server in order to manage the whole product life cycle. 62 Jenkins jobs are gathered in one view called "The general ProActive monitor". If one job of the general ProActive monitor view fails, it becomes a priority for developers to fix it. Every workday a developer is chosen as a "Sheriff" and his role is to ensure that every job in this view is always successful. We use Gradle for build automation. In order to keep a high level of code writing quality, we use the Spotless Gradle plugin² to check and correct the formatting before building. We use SonarQube for continuous inspection of code quality. We also use the dependency management Gradle plugin³ to centralize dependencies in order to avoid conflicts between services.

Every day, the master branch is built with new features and bug fixes. Once the master branch is built, it is deployed on two servers. One server operates Ubuntu 16.04 and is used for automated system testing. The other one is called "TryDev", which is an Ubuntu 15.10 server, and is used by Activeeon to develop and test new features.

² <https://GitHub.com/diffplug/spotless>

³ <https://GitHub.com/spring-Gradle-plugins/dependency-management-plugin>

7.A.3 Expected Improvements

PWS is a generic system, used in many different domains. Consequently, it requires thousands of tests on different configurations. An official release is produced about once a month. A set of 355 manual tests are performed before the release. A majority of these tests are graphical user interface tests and are performed on the quality assurance platform. Some of those manual tests are configuration testing. It always takes several days to do all the manual tests.

ProActive is a software framework that has been developed for more than ten years and for years the test quality has not always been the priority. Because of tight schedules, companies tend to prioritize quick release over quality. By improving our test suite thanks to STAMP, we expect to fix the future and past technical debt in the test suites.

Moreover, by removing technical debt in the tests, we expect to fix and correct bugs that would have appeared in production. Descartes provides reports that show the issues that could be fixed in the tests. Those reports should be used to remove technical debt and increase code quality. Furthermore, by having those reports as comment in the pull request, a developer could use the suggestions to improve the code quality before merging. EvoCrash will help to reproduce customer issues and reduce investigation time. Also, the corresponding non-regression tests will be automatically produced.

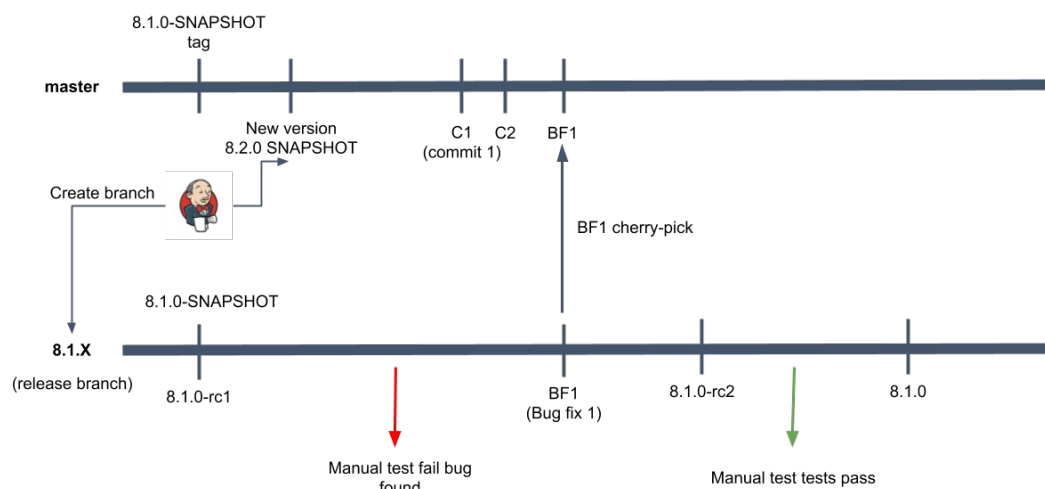


Figure 1: Proactive Workflow & Scheduling (PWS) Development Chain

7.A.4 Business Relevance

Because Activeeon is a software vendor leveraging an open source business model based on services, customer support is an important activity of Activeeon developers. Bugs and exceptions are reported regularly by our consumers. Identifying the origin and the exact contexts of these bugs is time-consuming and could induce delay in bug fixing due to exchanges with consumers. Quality assurance and customer support are core to Activeeon's business model. STAMP provides the tools that will help to reach the level of test required for a more stable product. Greater efficiency in testing and customer support would significantly improve the competitiveness and profitability of the company.

7.B Validation Experimental Method

7.B.1 Validation Treatments

Control and treatment tasks have been defined to measure all KPIs. The KPIs are described in D5.3, with tools and strategies for measurement. The following table gives tasks based on KPIs for Activeeon's validation experiments, and maps these to the STAMP tools to be validated. The resulting experiments are

described in the next sections.

ProActive Workflows and Scheduling (PWS) is organized in many projects, some are open-source and hosted on Github, others are closed-source and hosted on bitbucket. The main GitHub project for the ProActive Scheduler is called **Scheduling**⁴. Other projects such as the ProActive **Catalog**⁵ will be used during the validation.

DSpot and CAMP validation have not provided relevant result yet. We are still working on those tools to get results. As a consequence, we have not completed the validation result for both tools and focus more on the roadmap section.

Metric	KPIs	STAMP tool	Treatment for measuring
Code Coverage	K01	<ul style="list-style-type: none"> • DSpot • Descartes • Botsing 	<p>Code coverage is measured using Sonar inside the <i>Scheduling</i> project and using Clover inside the <i>Catalog</i> project.</p> <p>Clover is more accurate than Sonar because it can take into account integration tests and thus give a better indication on the total test coverage. Unfortunately, Clover was not used inside <i>Scheduling</i> because it does not currently support multi-module Gradle projects.</p> <p>A tool result is applied to the project one by one in order to avoid cumulative result.</p> <p>Each change to the project is measured to compute the tool result impact.</p>
Flaky Tests	K02	<ul style="list-style-type: none"> • Descartes • Camp 	<p>There is a baseline of flaky tests in the <i>Scheduling</i> project where they are all listed. Descartes and CAMP are used to stress out several execution paths.</p> <p>This would help to rise flaky tests since they usually fail due to changes in non deterministic way these tools are indirectly introducing such vulnerability.</p>
Mutation Score	K03	<ul style="list-style-type: none"> • Descartes 	<p>The mutation score is measured using PIT on the projects <i>Scheduling</i> and <i>Catalog</i>. The tools result is applied to the project one by one to avoid cumulative result. On each change the project mutation score is evaluated to measure the tool impact.</p>
Pseudo Tests Partial tested tests	K03	<ul style="list-style-type: none"> • Descartes 	<p>Count the issues reported by Descartes on the <i>Scheduling</i> and <i>Catalog</i> project. The relevant issues are fixed then the issues are counted once again.</p>
Execution Paths	K04	<ul style="list-style-type: none"> • CAMP 	<p>The first test coverage from Clover on the Catalog project is used as a baseline. By applying Clover on each CAMP configuration, it will generate a new test report that is aggregated by Clover. The generated report contains the code coverage among all configurations. By comparing with the baseline, it shows the coverage improvement introduced by CAMP.</p>

⁴ <https://GitHub.com/ow2-proactive/Scheduling>

⁵ <https://GitHub.com/ow2-proactive/Catalog>

configuration related bugs	K05	• CAMP	On the baseline, we have a green test suite on the Scheduling project. By applying CAMP on the <i>Scheduling</i> project and executing the test suite on several configurations, it highlights the bugs by making the test fail.
configurations tested	K06	• CAMP	Count the number of new configuration deployed by CAMP on the <i>Scheduling</i> project.
Time to deploy SUT in configuration	K06	• CAMP	For each configuration, count the time to deploy the <i>Scheduling</i> project.
crash replicating tests	K08	• Botsing	Count the number of crashes replicated by EvoCrash or Botsing from the project logs.
more production level tests	K09	• Botsing	Count the number of crashes replicated by Botsing that are introduced in the test suite as a non-regression test.

Table 3: Tasks for Measuring Metrics for Activeeon Validation Experimentation

7.B.2 Validation Target Objects and Tasks

Activeeon software is composed of several micro-services. Among those projects, Catalog⁶ and Scheduling⁷, were selected because they are very different (size, role, complexity) and open-source as stated on Section A. Working on open-source software is useful when evaluating tools because as an open-project share working examples with partners example and exact code used for testing without the need for NDAs.

In this chapter, we highlight the various type of tests being performed inside ProActive Workflow & Scheduling:

- **Unit tests:** low-level tests which validate each a small part of the code, meant to be stateless, and use mocks when interaction with other components is needed.
- **Integration tests:** perform testing scenarios on a single component or an assembly of several components. For example, integration tests of ProActive Resource Manager, integration tests of ProActive Scheduler connected with a Resource Manager, etc. Integration tests are run automatically inside each pull-request.
- **Regression tests:** similar to integration tests, but run only after a pull-request is merged (these are generally very long tests).
- **Performance tests:** perform a large number of concurrent or successive operations on the software, and monitor total time.
- **System tests:** similar to integration tests, but on the full ProActive Workflow & Scheduling release (all components), seen as a black-box.

We describe a few projects used:

The ProActive *Catalog* project is open-source software which acts as a database-backed object repository for the ProActive Workflow & Scheduling server. It can store various objects such as workflows, calendars, deployment infrastructure definitions, etc. It is developed using Java 8 and it uses the Spring framework. The Catalog is composed of 5866 lines of code in the last released version 8.3.3. It contains **184 unit tests** and **86 integration tests**. Those tests are separated and configured thanks to Gradle.

⁶ <https://github.com/ow2-proactive/Catalog>

⁷ <https://github.com/ow2-proactive/Scheduling>



The ProActive *Scheduling* project is also open-source software. It is composed of a scheduler and a resource manager that execute workflows on dedicated resources. It is set up as a Gradle multi-module project. The *Scheduling* project, core of the ProActive software suite and developed during more than 10 years, is the project with the highest complexity. Java 8 coding style support has been added one year ago, and the project is being refactored to make use of Java 8 syntax. *Scheduling* is composed of more than 90 000 lines of code, **1309 unit tests**, **389 integration tests** and **46 regression tests**. Gradle is used to configure modules that are built and the project tests.

The various projects such as *Catalog*, *Scheduling*, *job-planner*, script engines, etc. are assembled together when building a ProActive Workflow & Scheduling release (nightly, candidate, final, etc). The release is then deployed on a dedicated server against which **361 system tests** (black box) are run. System tests are using the REST API of the various micro-services exposed by the ProActive Workflow & Scheduling server.

DSpot and Descartes will be tested both on the *Scheduling* and *Catalog* projects. We decided to limit our scope at first on the *Catalog* project because of its simplicity compare to *Scheduling*. *Scheduling* will be used for more advanced testing in order to make the tools face bigger challenges. In order to be able to measure the impact of those tools, we configured Gradle to use Clover on both tools. Configuring Clover on *Catalog* required to update all the *Catalog* API because Clover does not support automatic Spring generation for API parameters. In order to handle the issue, we added explicitly the name of each parameters. In the *Scheduling* project, we searched for two weeks to make Clover plugin work with multi-module project. In the end, the idea was dropped due to the task complexity. Using Sonar and Jacoco is the alternative that was chosen to measure the tools' KPI on the *Scheduling* project.

CAMP is evaluated by deploying ProActive and executing the system-test on it with several configurations. Docker files and CAMP configuration files are developed to deploy such testing environments. A ProActive workflow was also developed in order to test CAMP usage with online testing.

Botsing is executed with the ProActive Workflow & Scheduling server logs. When the STAMP project started, the logs from all microservices were gathered in the same log file. We worked on harmonizing the logging process in order to separate the logs in different files. Having a specific log file per microservice/component allows to get individual microservice stack traces in an easier way. Initially, using EvoCrash in the project [EvoCrash-demo](#) was time consuming because it was not built with an automation tool and the configuration parameters were resolved at compilation time. We fixed the building issue by writing a Gradle build file which performs the build and automatically runs the EvoCrash-demo project. Then, we refactored the code to externalize the configuration parameters in an external file `config.properties`. This enables to load the parameters at runtime and avoid to recompile the code before each execution. In order to ease the use of EvoCrash from Gradle, we also developed a [Gradle plugin for EvoCrash](#). The plugin development and the migration from EvoCrash to Botsing is on-going.

7.B.3 Validation Method

Descartes

Test results need to be accepted by the developer team to be continuously supported and meaningful. In the past, we experienced bad test design that was often indicating false positives instead of helping to detect bugs. At that point, we knew that when a test failed, it was more likely because of a bad test design rather than the new piece of code failing a regression. This changed a lot in the last years where we introduced tests at several levels: unit tests, integration tests, system tests. While unit and integration tests are developers' tools, system tests run on a nightly basis trying to find bugs coders might have introduced when pushing the last changes.

This change in the test framework impacted our quality process, we managed to find hundreds of bugs before they reached production. However, by doing so, we also added time-consuming steps in our development chain. Currently, we lack a baseline to precisely quantify the outcome of this test framework improvement, but we sure share the impression as developers that our software is more reliable and clients are arguably coming forward with less bugs. Developers are finding faster ways to reproduce and adding

tests to avoid the same bug to appear in the future. Nevertheless, we are still searching for innovative ways of improving quality. STAMP brings us the discussion on how we can improve our test framework even more. Using automatic tools for testing is being done for a while and is part of the company culture. However testing the test set itself is innovative and definitely is a new factor to incorporate in our testing process.

CAMP

To explore configuration related bugs and execution path related KPIs, we will rely on the CAMP tool. As a software conceived with focus on configuration and execution path exploration, CAMP is a tool capable of testing several configurations that - inside our current process - we only tested manually before releasing or upon client's request. A simple example of a CAMP configuration is in Figure 2, where we can already explore a few JDK distributions and support for many databases. PWS depends on a database to run, and is shipped internally with the *hsql database*. Nevertheless, PWS can be configured to run with an external database such as: *mysql*, *mariadb*, *postgres*, *sql server*, and in a near future, *oracle*. This implies that upon a new release, we currently have to manually test PWS with all these databases to assess production quality. Integrating CAMP in our nightly-release testing process would enable to immediately spot regressions and compatibility issues when we bring modifications on the database or hibernate schemas. For example the creation of a new entity table in the Hibernate schema could work perfectly fine on MySQL and Mariadb, but will create issues on Postgres because the new table name is a Postgres reserved word. Examples of such potential issues are countless. Quantitative validation of what CAMP brings to our testing capabilities will be essentially measured by an increase in the number of different environments and configurations we are able to test. Qualitative validation will be related to an overall improvement in the product stability and also the ability to detect incompatibilities with several environments (for example a specific version of a database).

```
java:

openjdk: [openjdk8]

oraclejdk: [andreptb/oracle-java:8]

ibmjdk: [ibmjava:8-sdk]

db:

mysql: [mysql-server:5.5, mysql-server:5.6, mysql-server:5.7]

mariadb: [mariadb/server:10.3, mariadb/server:10.2, mariadb/server:10.1]

postgres: [postgres:9, postgres:10]

proactive: [proactive84mysql, proactive84mariadb, proactive84postgres]
```

Figure 2: PWS CAMP yaml Configuration Example

Designing a Camp-based testing environment for ProActive Workflow & Scheduling requires a quite important development and experimentation effort. Similarly to Gradle build files which can be very complex to write, writing Dockerfiles able to extract any PWS zip release, configure all microservices to attach to a Camp-provided database, do some configuration in the database itself, changing the PWS embedded Java Runtime Environment to another one provided by a Camp Docker container, start the scheduler server, run the system tests, make sure the test results are all available and labelled by each configuration tuple, etc...

EvoCrash/Botsing

Using EvoCrash and Botsing requires to gather logs that are used as an input. The *Scheduling* project is the core of ProActive Workflows & Scheduling. We gather the PWS server logs from several of our platforms and from exchanges with our clients. In order to analyse these logs with Botsing/Evocrash, we need to set up the analysis using the same version of the *Scheduling* project used at the moment of the crash. Going through

the Scheduling log to gather the relevant stack trace is important for gathering the inputs. Indeed, when an exception is thrown in the log, it is usually thrown several times, making it harder to find a new exception.

After gathering the stack traces that are used as input for Botsing/EvoCrash, we build the Scheduling project in order to get the codebase used as an input for the tool. Both software are configured differently. EvoCrash is configured through its config.properties file. Botsing parameters have to be provided by command line. As a consequence we wrote a simple script containing the command to execute with the parameters. After settings up the configuration file we execute Botsing.

7.B.4 Validation Data Collection and Measurement Method

Descartes

STAMP proposes with Descartes to run PIT tests for code coverage adding mutation coverage. Indeed, code coverage is a common sense approach to know whether tests are exercising the code properly. We have code coverage through Sonarqube however we fail to consider it in test design. As a matter of fact, running PIT/Descartes on the *Catalog* project (which models the entities in our solution) we found 314 code coverage issues. This brings to light that we must add code coverage as part of our company culture.

Mutation coverage is a less straightforward concept in comparison to traditional line coverage. Line coverage is the amount of lines of code that tests explore, 100 % coverage means all tests will, at some point, execute every single line of code. To explain mutation coverage let us stick to this example when we have 100% code coverage, if this is the case we might think our tests are good enough to detect problems anywhere in the code. However if tests do not have any assertions, the 100% coverage is less reliable as a test quality factor. One can say that these tests would still detect when exceptions are thrown, nevertheless this is a limited assertion⁸. The Descartes engine produces simple mutations of compiled code replacing complex methods by the faulty ones. For instance, always return true for a boolean function or use an empty method body for a void function. These methods that query bugs on purpose are called **mutations** and have as functionality to evaluate if tests as reliably breaking when the code is broken. If a mutation makes a test fail Descartes states it was successfully killed, and this is a good thing, meaning the test is strong enough to detect the artificially introduced bug. However, if after mutation the test does still pass, the mutation survived meaning the test fails to detect it. So these reports represent an automatic way to evaluate the quality of test framework.

Camp

To evaluate PWS software with Camp, we first need to provide a recipe for each different environment. This recipe corresponds to a ProActive distribution configured properly to work in this environment. Technically, as Camp is based on Docker, this will mean that we must generate a ProActive Docker image configured on a given environment. These Docker images should define build parameters to enable easy changes of minor variations. For example, a PWS Docker image configured to run with MySQL 5.5 should not require a major change to run with MySQL 5.6 (or no change at all). On the other hand, changing from a MySQL configuration to a Postgres configuration is a major change and require different Docker images.

We will concentrate on three variations: change of the java runtime environment used by the ProActive scheduler, change of the backend database software and finally change of the PWS server version itself.

- Change of the Java Runtime Environment (minor):

For the sake of usability, PWS is by default bundled with a 1.8 version of the Oracle JRE. Out of the bundle, the user has a start script that configures JAVA_HOME properly to use the embedded java distribution. To incorporate the capability of switching JRE distributions, we need to add support for an option to customize the JAVA_HOME environment variable on the proactive_server initialization script. The used JRE will be provided as a public Docker image (Docker Hub contains multiple versions of Oracle JRE, OpenJRE and IBM JRE)

We need the ProActive server to be able to detect automatically the java installation present in the Docker container. By doing so, we will be able to combine easily various JRE Docker images with the ProActive

⁸ Martin Fowler, last accessed in November 2018, <https://martinfowler.com/bliki/AssertionFreeTesting.html>

server.

- Change of the backend database vendor (major):

Similarly, to the embedded JRE, ProActive Workflows & Scheduling is released with an embedded database, HSQLDB, used as a backend for multiples microservices. In order to change one microservice database configuration, the system administrator must edit a database configuration file, providing for example the database url, username and password, the database schema used, etc. All of this needs to be automated in the context of a dockerized Camp recipe. This automation must be performed to change multiple configuration files inside the ProActive distribution, and also each database vendor requires specific configurations. Accordingly, a PWS Docker image must be built at least for each database vendor, e.g., PWS-mysql, PWS-postgres, etc.

Similarly with the JRE, publically available Docker images exist for various databases and databases versions. By associating the configured PWS Docker image with the corresponding database Docker image, we will have an up and running PWS server which can be tested.

- Change of the ProActive Server version (major):

Similarly to other public Docker images, PWS server images configured to run against a given database vendor will be versioned.

To configure CAMP we must describe the desired configurations. For instance, in **Figure 2** a yaml description featuring categories JDK distributions and databases show the goal of the exploration with CAMP in this UC. JDK distributions are worth testing because users may vary their preference. Moreover support should be straightforward among the 3 jdk distributions: OpenJDK, Oracle's JDK, and IBM's JDK. As outcome of exploring these we are likely to find more bugs and incompatibilities since we are using compilers and runtimes from 3 different vendors. Users of PWS also want to choose the database that matches their needs. Testing on several databases is currently manually done at the end of the development chain. CAMP hence is the key to assess product quality continuously from client's perspective.

CAMP requires a specific version of the Z3 solver that will provide the exploration of configuration following constraints. As an outcome we started dialog to improve the tool support for the Z3 solver. Downgrading the Z3 solver is a less convenient work around, yet possible. First because it requires tweaking and second because it avoids the convenience of installing with package managers available in most linux distributions such as apt-get for Debian/Ubuntu, yum for fedora, and such. Finally having to adopt a particular version of Z3 will eventually hinder the usability of the tool, only users eager and willing to tweak their systems will have the opportunity to add support. This is a major issue to spread the usage of CAMP and by consequence increase STAMP users in future exploitation plans.

EvoCrash/Botsing

We focus on two parameters in order to obtain different results for the tools. Our results are a combination of different logs with a variation of the stack trace frame level. We use seven Scheduling stack traces coming from client environment and test environment. Among those stack traces there are five custom exception and two existing exception from Java 8 library and JSON library.

For testing Botsing, we mainly focus on reusing successful test cases that worked with EvoCrash in order to compare the results.

7.C Validation Results

7.C.1 KPIs Report

Descartes

Using Descartes we evaluate line and mutation coverage several times, flakiness of tests K02 is related to these scores. For K03 we evaluate mutation score, pseudo-tested, and partially-tested. The plan is to evaluate how the project Catalog can be improved using these metrics to hence establish a roadmap to evaluate in depth other software modules of PWS. The table below show scores on Catalog project.

Type of test	Score	Description	Action
mutation coverage	36 %	Percentage of lines of code in which mutations were never explored by the tests.	Improve and add tests that cover related lines of code.
line coverage	43 %	Percentage of lines of code that without mutation are explored by the test set.	
pseudo-tested	38	Tests that pass for all related mutations.	Manually inspect each test case and evaluate survived mutations.
partially-tested	3	Tests that pass for at least one related mutation.	

Table 4: PWS Catalog Project Scores

In a quick evaluation we can already conclude that Catalog scores poorly in mutation coverage and line coverage. After in-depth investigation, we found a bug in our project thanks to Descartes. Indeed, the test was strong enough, but return values were never checked inside the tested method. The problem is probably because this method was mocked in an earlier development stage and never tested its return values when finishing the implementation. Further investigation showed that 36 of pseudo-tested plus 3 partially-tested are actually mutations of auto generated methods. In *Catalog* and several other PWS modules, we rely on external libraries and frameworks such as: Lombok, Spring, and GraphQL. These frameworks auto-generate boilerplate code like getters, setters, constructors, using java annotation. For instance, one can create a class and just using annotation add constructor or even GoF classes to implement the builder pattern. Mutations on these auto-generated methods are not detecting a problem in our tool but in the framework, and hence are false positives for us. This limitation is critical to run Descartes in other projects. As an outcome of this UC we filed an issue on Descartes project to take into account automatic method generation.

EvoCrash/Botsing

The result of Activeeon tests can be found on the <https://GitHub.com/STAMP-project/evocrash-usecases-output/tree/master/Activeeon/Evocrash/Scheduling> url.

It contains the results and analysis of the experimentation. We manage to generate a test that was able reproduce one of our stack traces. In order get a better understanding of EvoCrash output, we test that stack traces with several configurations. We modify the values of the parameter *frame level* going from 1 to 9. We observed three different kinds of result:

- Test are generated and can reproduce the stack trace at the requested level (Frame level 1,2 and 3)
- Test are generated but the test method is empty (Frame level 4 and 5)
- Test are not generated due to crash or timeout after 30 minutes running. (Frame level from 6 to 9)

Moreover by testing several time with the frame level 3, the generated tests are different, even if they all reproduce the stack trace.

By aggregating all the crash reproduction results we have 14% of well generated tests, 57% of empty tests generated and 29% of stack traces that don't generate tests. All the stack traces that do not generate tests are throwing existing exceptions, while 71% of stack traces that generate tests are made of custom

exception.

To conclude EvoCrash works better with a low frame level, this result is expected as it has lower complexity. Moreover, it seems to work better with custom exceptions than with existing exceptions.

7.C.2 Qualitative Evaluation and Recommendations

Descartes

Company test culture within agile teams is difficult to change because it requires to convince developers of the gain it has to offer. Testing seems often a boring task and is many times neglected. Descartes answers questions that go on this direction: Are my tests relevant? Will my tests break if a function stops working as expected? As a qualitative outcome we see these questions appear for the first time in our testing development retrospective meetings and we actually found a bug using this tool in a module. Moreover, we see with line coverage poor scores that unit tests are far from comprehensive and they are a candidate for a change as part of our company culture. So as a conclusion we see that Descartes enable to detect flaky tests and also provides a way to assess coverage on the traditional way with line coverage but also enlarge this concept on passing tests quality by mutation coverage.

Botsing

Botsing is the new release and version of EvoCrash. It came out with a several improvements that are improving its usability and accessibility. Botsing has less parameters than EvoCrash making the configuration easier and more understandable. Botsing is available on maven which making integration with plugins easier to develop. However some issues that exist in EvoCrash have not been fixed yet, the generated tests are too far from the human written tests which make it hard to integrate in the test suite. We are looking forward improvement in that context which would enable us to integrate Botsing generated tests in our test suite as non regression tests.

7.C.3 Answers to Validation Questions

VQ1 - Can the STAMP tools assist software developers to trigger areas of code that are not tested?

From our coverage scores on the Catalog project we can see 43% which is less than half of the code. Indeed before STAMP, developers were often bypassing the coverage check. Mutation coverage with Descartes gives a good idea of tests missing helping to bridge that gap. According to those reasons we can state that STAMP indeed helps to pinpoint areas of code that need testing and help to fast integrate this on the CI. For example, thanks to the STAMP tools we discover that a package of the Catalog was not tested. By adding integration-tests we improve the code coverage from 59% to 62%.

VQ2 - Can STAMP tools increase the level of confidence about test cases?

Currently we appraise test quality during code review tackled by a fellow developer. Sometimes a bug fix is critical and this review process may be hindered or even neglected. As a consequence, tests are often missing or have poor line coverage scores. This also entails that our current confidence in tests is low because we miss tools and knowledge of good practices for testing.

Adding Descartes to our CI we already envision two main advantages. First, stressing out tests using several executions of the same test while adding mutations enables to expose flaky tests and respond quickly to such kind of issues. Second, in addition to assessing coverage the traditional way, we get additional insight into test quality by evaluating the mutations from Descartes in our continuous delivery pipeline. With Descartes we can enhance our current process, which uniquely relies on human feedback. Instead, we can now rely on performance indicators such as partially-tested or pseudo-tested scores.

We are currently working on CAMP to unleash its full potential. However, we can already report from the back and forth discussions within our team that this tool is useful for some critical recurrent issues. So to give an idea of how developers are willing to integrate CAMP, in our last bi-weekly retrospective, this task was upvoted and developers agreed that exploring database configurations is a major concern that has degraded user experience in our last 2 releases. When implementing this form of test automation exploring different

environments we see a strong potential to increase the quality and confidence on our tests.

In this UC we see tests that are solely certified by humans and have limited configuration scenarios. With the help of STAMP tools Descartes and CAMP, we see a way of automatically integrating quality indicators into our CI. Within our development team there is consensus that STAMP is a powerful audit toolbox that raises discussions on quality and increases confidence in tests.

VQ3 - Can STAMP tools increase developers confidence in running the SUT under various environments?

We did not manage to get result from CAMP making it impossible to answer to the question.

VQ4 - Can STAMP tools speed up the test development process?

Botsing is able to generate stack traces that reproduce a crash that appears in a log file. We were already able to generate a test from logs provided by the client. However this test cannot be integrated in the test suite because it is too far from a hand written test. By improving the test quality to something that is human readable we would be able to integrate it in our test suite as a non-regression test. Generating non-regression test from a crash would help to do test-driven development. This would speed up the test development process, but also speed up the bug fixes development process.

7.C.4 Next Steps for Validation

DSPOT

We manage to run DSpot on the Catalog by using the option automatic-builder with the value GradleBuilder. The option enables to run the tool against a Gradle project. We have worked together with the DSpot developers in order to make it work with Gradle. Catalog contains 30 test classes and DSpot runs without error on 93% of them. DSpot runs on all the projects tests in 16.3 seconds. The Activeeon use case results can be found on GitHub at: <https://GitHub.com/STAMP-project/dspot-usecases-output/>

Thanks to the discussion and exchange with other use case partners, we know that DSpot usually does not provide results in 16.3 seconds. We can easily confirm that by checking the results on the previously quoted GitHub repository. For example, XWiki time execution goes from 15 minutes to 60 minutes. Our project analysis is at least $(15 \times 60) / 16.3 = 55$ times faster. As a consequence, there is probably an issue with our environment. We plan to fix the issue and measure again the result of DSpot against the Catalog.

Tasks	December				January				February			
Fix pitest error												
Generate tests with new amplifiers												
Integrate DSpot in the CI												

Table 5: Roadmap for DSpot within Activeeon's Use Case

Descartes

To the roadmap of Descartes we will focus on using it in several projects; we will also push to have the automatic generated methods ignored. A tentative roadmap of next 3 months is show in Table below.

Tasks	December				January				February			
add support on Scheduling												
detect auto generated methods												

explore Descartes report												
Quantify quality gain												
Delivering final results												

Table 6: Roadmap for Descartes within Activecon's Use Case

CAMP

CAMP is a major change in company culture. We currently support configuration testing through manual testing before releasing. This is a time consuming approach that is needed to assess the usability of software, giving insights for new features brings as drawback configuration specific bugs to appear just before production. Finding bugs just before production, implies finding them at the last point of the development chain. By consequence, several changes are candidates for introducing the bug, which hinders to pinpoint the person who could give a best response. It is common sense that a bug detected earlier reduces the investigation effort because developers are very likely still working on the changes that caused the issue. Our main goal with CAMP is to detect these bugs earlier and then respond more appropriately. Hence we plan to reduce the amount of critical bugs that are brought by clients after updating PWS. By this we conclude that STAMP satisfactorily enables to explore configuration environments that are not yet tested and part of our current company culture.

We are currently working on using CAMP, however the changes on our software to accommodate quick changes in configuration require some work: first, to implement changes in the configuration of the JDK and database that are as quick as setting an argument; secondly, create Docker containers that have pre-configured functional versions of the expected databases. We are currently working to provide these functionalities to enable us to use CAMP. Moreover, as we stated before, the Z3 solver compatibility issue needs to be mitigated through a more precise workaround documentation or even better by supporting the version that is automatically installed using package managers. As the writing of this report advances we reiterate a tentative roadmap in Table below.

Tasks	December				January				February			
workaround Z3 limitation												
proactive test suite for CAMP												
Docker proactive mysql												
Docker proactive mariadb												
Docker proactive postgres												
Explore db configuration												
Quantify quality gain												
Explore jdk distributions												
Delivering final results												

Table 7: Roadmap for CAMP within Activeeon's Use Case

Botsing

We first plan to execute the same validation on Botsing that we have already executed on EvoCrash. It will highlight the changes and improvement of that new version of EvoCrash. We also plan to finish the Gradle plugin which will make easier for a Gradle user to use EvoCrash in a Gradle environment. We aim to finish it before the next STAMP workshop in January in order to attract a maximum of users.

Tasks	December				January				February			
Fix Botsing Gradle plugin												
Apply Botsing on stacktraces												
Use Botsing on new stacktraces												

Table 8: Roadmap for Botsing within Activeeon's Use Case

8. Atos Use Case Validation

Atos Use Case Highlights

- Higher efficiency of the IF component test base to run larger portions of code (up to 10%) thanks to DSpot amplification
- Increase confident on IF test base quality thanks to Descartes analysis: significant increase of code coverage (up to 25%) and reduction of test quality issues (from 33 to 97%).
- Lesson learnt: test coverage of non-favorable IF execution behavior (exceptional branching) was ignored by test developers, but included by DSpot amplification
- -Botsing network security restrictions impedes the amplification of crash replicating tests for IF runtime exceptions
- CityGO CityDash optimal tuning under different workloads is now straightforward thanks to CAMP automatic generation of delivery configurations.
- CityGO CityDash delivery has been automated with CI/CD workflow and operational failures are minimized by configuration using CAMP amplification.

8.A. Use Case Description

8.A.1 Target Software

Atos Research and Innovation (ARI) department participates in R&D projects funded by different public funding programs, such as H2020, ECSEL, FI-PPP, EIT-Digital and others. Within these projects, ARI collaborates with research institutions and the academia on the development of innovative software technologies and tools. Internally, ARI Innovation Hub a) brings R&D project assets closer to the market and demonstrate the innovation capacity of Atos, b) communicates internally across all Atos Business Units and Global Business Units the ARI assets through innovation workshops, digital shows, etc, and c) incubates and favors the transference of ARI assets to foster new business for Atos.

In the context of STAMP, two R&D projects have been selected for experimentation with the STAMP test amplification toolset, namely SMART-FI and SUPERSEDE. In SMART-FI (<http://smart-fi.eu/>), Atos is developing CityGO (<http://www.city-go.eu/>). In SUPERSEDE (<https://www.supersede.eu/>), Atos developed DAPPLE. These assets are currently under evaluation and commercial promotion by the ARI Hub.

CityGO is FIWARE IoT-based platform that offers a Web Portal and Android App that citizens can use to obtain the best estimated transport route inside a city by combining several parameters such as user requirements, real time traffic data, location of vehicles, cars and bicycles parking availability and environmental data (weather forecast, social events in the city, etc.). The Web portal provides an informative dashboard that shows statistical data about the human and vehicle traffic inside the city.

CityGO was conceived to address some benefits for the citizens and the city:

- Traffic optimization inside the city
- Reducing the air pollution by encouraging people to use non motorized transportation.
- Low latencies and optimum workload for the transportation services
- Increasing the demand to use public transportation
- Introducing real time city data for the municipality to take actions based on the extreme situations, etc.

CityGO consists of:

- CityGO: a mobile application that indicates to the user what public transport options are available at any time for a particular route. For instance, it suggests options such as electric car sharing, buses, the nearest public bike rental station, available parking spaces, etc. Everything is managed in real time to obtain an optimal route based on data provided by the sensor network and open data from the city.
- CityDash: a Web-based dashboard for the city municipality control center, which allows civil servants to visualize all the data coming from the city sensors network to support everyday decision making and evidence-informed analysis to improve the traffic planning in the city in times of high tourist's flows, sport events, among others.

An instance of CityGO is currently operating on Malaga city (Spain), integrated with several sensors and data sources (traffic, weather forecast, social events, vehicle and bicycle stations etc..) provided by the Malaga Municipality.

As a solid product, the delivery of CityGO platform in other EU cities (e.g. Geneva, Nice, Toulouse or Portland) is under commercial action as these cities have showed their interest on this solution. Moreover, a commercial pilot is being developed in the EIT-Digital CEDUS project.

CityGO is implemented in different technologies: Python and Angular for the backend and Web Portal, and Java for the Android App. Adopted FIWARE backend services, such as Cygnus or Orion are developed in C++ and Java. Docker and Ansible containerized configurations for installing the CityGO platform are available.

SUPERSEDE develops a platform that assists software stakeholders (e.g. product owners, system architects and administrators, etc) on the software evolution and adaptation for their software systems, based on

gathered end-user feedback and system monitoring data. Concerning dynamic software adaptation (i.e. self-adaptation), SUPERSEDE develops an implementation of the MAPE-K control loop. Monitoring is supported by end-user feedback gathering and system monitoring techniques. Analysis uses Big Data techniques to cross and analyse data from multiple sources, in order to propagate observations on aggregated data. Planning implements decision making methods, based on Genetic Algorithms, that compute optimal configurations (as feature configurations) of the target system that will cope with runtime observations. Execution uses UML models@run.time that are kept in sync with the target adaptive system configuration. Optimal configurations are woven into these models using AOM techniques. Pluggable enactors interpret these UML models, generate and activate new target adaptive system configurations.

The SUPERSEDE platform is constituted as an aggregation of backend and front-end components that jointly inter-operate among them using an Integration Framework (IF) developed within the project. IF provides client-service communication among these components using the WSO2 Integration Platform (ESB, IS, DSS, MB). Docker-containerized configurations for installing the platform are available.

The SUPERSEDE backend components have been developed in different languages, mostly using Java, but also Ruby and Javascript (NodeJS). Front-end components have been developed with JS/HTML.

For this industrial validation process, Atos uses as validation target SUT the following software components, taken from the two projects described above:

SUPERSEDE IF

IF is a inter-service communication Java library that uses a WSO2 framework to deliver messages between SUPERSEDE front-end and backend components. The WSO2 framework consists of:

- An Enterprise Service Bus (ESB) that mediates and dispatches messages between peer-services in a micro-service architecture,
- An Identity Server (IS) that provides secure access to services based on user authentication and user access rights,
- An Message Broker (MB) that provides publish-subscribe broadcast communication,
- An Data Server Service (DSS) that exposes SUPERSEDE data-sources as services.

IF library offers a uniform and centralized Java API to the complete set of APIs exposed by the SUPERSEDE front-end and backend services. This library is used by all Java-based SUPERSEDE components to send messages to each other.

Table 9 describes some metrics about IF library source and test code base, computed with Clover library for Maven.

Component	SLOC	Classes	Unit tests	Coverage	Mutation Score
IF	25493	318	136	47,5%	39%

Table 9: SUPERSEDE IF Key Code Metrics

CityGO CityDash

CityGO CityDash is a Web application developed with HTML/Javascript/Css and Python/Django code.

Figure 3 shows statistics computed using the Metrics⁹ Python script on the source code of CityDash., summing up to 15473 SLOC.

⁹ <https://pypi.org/project/metrics/>

Metrics Summary:				
Files	Language	SLOC	Comment	McCabe
3	Bash	25	10	4
17	CSS+Lasso	2162	140	0
13	HTML	582	181	0
2	HTML+Django/Jinja	46	6	1
31	JavaScript	9272	2144	2636
2	JavaScript+Genshi Text	361	42	42
2	JavaScript+Lasso	2179	320	397
35	Python	838	157	16
1	Text only	0	0	0
1	Transact-SQL	8	0	0
107	Total	15473	3000	3096

Figure 3: CityGO CityDash Key Code Metrics

8.A.2 Experimentation Environment

Atos experiments for STAMP test amplification on the SUPERSEDE IF and the CityGO CityDash SUTs have been conducted in a dedicated server, with the following technical characteristics:

Intel(R) Xeon(R) CPU E5-2620@ 2.00GHz, 6 cores, 32 Gb RAM.

In this server we have also installed the complete SUPERSEDE platform (including the front-end and the backend services) and the CityGO CityDash. Both UC SUTs have been built and delivery onto the server by CI/CD workflows run in a Jenkins instance hosted in the same server.

SUPERSEDE IF test suite has been installed in the server from the GitHub sources. CityGO CityDash sources has also been installed, including the default Docker descriptors for their backend and front-end components as well as parameterized Docker Compose descriptors that set up the entire platform and launch it.

In the server several setups for STAMP tools have been installed and For all these tools, we have also installed CLI scripts to run the tools and collect results:

- DSpot v1.2.0 from GitHub sources.
- Descartes v1.2.4 from Maven repository.
- Botsing v1.0.0 from GitHub sources.
- CAMP v0.1.0 from GitHub sources.

8.A.3 Expected Improvements

The key motivating challenges for Atos to adopt the STAMP test amplification techniques were elaborated and justified in previous documents, D5.5 and D5.4.

The Atos validation objectives and expected improvements for this assessment period (M18-M24) are listed in the following, grouped by UC:

CityGO CityDash

CityGO CityDash will be the main UC for adopting and evaluating the **STAMP test configuration amplification techniques and tools**.

CityDash cannot be adopted for evaluating the STAMP technologies and tools for unit test amplification and online testing amplification, since they are only compatible with JVM compatible systems.

The primary objectives for adopting STAMP technologies and tools in the CityGO CityDash UC are the following during this evaluation period:

- **Deliver an optimal CityGO platform for the current runtime infrastructure conditions.** This objective implies, in turn, to find out an optimal CityGO configuration for some specific non-functional requirements such as performance, throughput, reliability, resources consumption, and others. The qualification of optimality may depend on the perception of CityGO App users and CityDash supervisors, but it can be quantified by means of stress tests. The fulfillment of this objective depends on the capability of the STAMP techniques and tools to assist the CityGO administrators on the generation and testing of new configurations that provide optimal parameterization of the CityGO backend services w.r.t. performance and resources usage.
- **Deliver CityGO CityDash onto different compatible runtime platforms provisioned by city administrators.** Atos aims to increase the baseline of CityDash runtime compatible platforms and the confidence that CityDash execution is reliable on them. CityGO has been developed and tested by targeting a concrete baseline platform (OS, datasources, Web containers, Python/Django, etc) selected by the development team. The compatibility can be quantified by means of sanity tests. However, Atos is committed to support the widest possible range of compatible baseline technologies, in the aim to deliver CityGO onto the existing platforms supplied by the adopting city administrations. Atos also needs to identify beforehand all potential issues caused when running CityGO in those compatible baseline platforms, in order to neutralize them or minimize their impact. Similarly to the previous objective, the fulfillment of this one depends on the capability of the STAMP techniques and tools to assist the CityGO administrators on the generation and testing of new configurations that diverge from the default baseline. Moreover, as this test-on-configuration process is traditionally high time consuming, Atos is expecting that STAMP tools can contribute to accelerate it.

SUPERSEDE IF:

SUPERSEDE IF will be adopted as the may UC for evaluating the **STAMP technologies and tools for unit test amplification and online testing amplification**,

The primary objective for adopting STAMP technologies and tools in the SUPERSEDE IF UC is the following during this evaluation period:

- **Ensure the runtime functional consistency of the SUPERSEDE backend platform.** SUPERSEDE IF constitutes the inter-service communication framework used by all backend services and also between the front-end and the backend. IF encompasses a complete test suite that constitutes the de-facto sanity-check for the entire backend. In other words, IF test suite plays the role of the SUPERSEDE system integration test suite. Given the importance of this role, the SUPERSEDE IF test suite needs to be improved in several dimensions:
 - Improving the ability of the IF test suite to simulate the complete space of communications between SUPERSEDE front-end and backend services. This sub-objective implies to explore the widest possible variety of API invocations and the variety of communication messages. Behind this objective, we identify the need to reduce the IF inter-service communication untested features.
 - Improve the ability of the IF test suite to detect deviations from the normal backend service behavior during its execution in the evolutive maintenance. For this objective, IF test suite is executed periodically in a pre-production environment, or upon any update in the backend or front-end services.
 - Improve the clean-up process after test execution, guaranteeing that no side-effects are left on the SUPERSEDE execution environment, remarkably its data sources.
 - Related to above sub-objectives, we are also aiming to increase our confidence on the IF test suite results and their validity to shed some light related to above sub-objectives.

8.A.4 Business relevance

Atos Research and Innovation (ARI) is the R&D hub for emerging technologies and a key reference for the whole Atos Group. Software Quality Assurance activities around software testing have been progressively incorporated in the context of both the Atos Group for commercial development projects and, although not as widely, the ARI department for R&D projects. STAMP gives ARI and Atos a great opportunity to promote software testing methodologies and technologies as an essential activity in the software development

lifecycle and the DevOps pipeline in the development of R&D and commercial projects. The adoption of testing best practices (such as Test Driven Development) and test amplification techniques (with STAMP), which eventually are promoted onto commercial IT projects, plays as competitive differentiator in winning commercial projects. With regards to this specific use case, STAMP can help facilitate the market positioning of CityGO and DAPPLE which are currently under evaluation for commercial promotion by the ARI hub.

8.B. Validation Experimental Method

8.B.1 Validation Treatments

Different validation control and treatment tasks are adopted by Atos in these validation experiments in order to compute the KPIs metrics that can assist us to answer to the validation questions formulated in section 6. Different processes for computing the associated KPI metrics have been proposed in D5.3. This section describes the control and treatment methods adopted by Atos.

Control tasks are those conducted to compute the baseline for KPIs. Table 10 describes the control tasks for computing baseline and control KPI metrics in SUPERSEDE IF and CityGO CityDash use cases¹⁰:

Note: see D5.3 for definitions of KPIs, metrics and details about assisting technology.

Metric	KPIs	Task Name	Task ID	Description	UseCase
Code Coverage	K01	Compute Code Coverage	CCC	Code Coverage is computed using Clover and Jacoco tools, independently. In both cases, their configuration and execution is managed through Maven.	IF
#Flaky Tests	K02	Identify Flaky Tests	IFT	Manual repeatedly execution of test suites. Test cases that eventually are failing and passing will be manually analysed in deep detail in a search for flaky test candidates. Those for which there is not found a reason (related to test environment or test quality) for their eventual failure will be accounted for flakiness.	IF
Mutation Score	K03	Compute Mutation Score	CMS	Mutation score is computed using PIT and STAMP Descartes tools, independently. In both cases, their configuration and execution is managed through Maven.	IF
#Pseudo Tests #Partial tested tests	K03	Compute Test Quality	CTQ	These metrics are computed using STAMP Descartes tool.	IF

¹⁰ See D5.3 for definitions of KPIs, metrics and details about assisting technology.

#Execution Paths	K04	Compute Execution Paths	CEP	Inter-service/Inter-component message exchange logs are injected in code. Bash scripts are created to query for accounting specific entries related to above inter-service communications in runtime logs. Stress tests and sanity tests are executed on amplified deployment configurations using CI.	CityDash
#configuration related bugs	K05	Detect configuration bugs	DCB	Stress tests and sanity tests are executed on amplified deployment configurations using CI. Runtime manual logs are inspected in the search for reported exceptions. Occurrences are manually investigated and reported. 5 different queries are sent to CityDash by 50 concurrent users created within a ramp time of 10s	CityDash
#configurations tested	K06	Compute number of configurations tested	CNCT	STAMP CAMP CI Executor is used to generate, instantiate and tests (using stress tests and sanity tests) the amplified configurations that it generates. CAMP reports the number of configuration tested. 5 different queries are sent to CityDash by 50 concurrent users created within a ramp time of 10s	CityDash
Time to deploy SUT in configuration	K06	Compute deployment time	CDT	STAMP CAMP CI Executor reports the deployment time. 5 different queries are sent to CityDash by 50 concurrent users created within a ramp time of 10s	CityDash
#crash replicating tests	K08	Compute number of crash replicating tests	CNCRT	Manual computation of the number of successful crash replicating tests generated by STAMP Botsing	IF
All	All	Setup	Setup	Preparation of SUT and STAMP treatments for experimentation. In both cases, setup is semi-automated, conducted manually using Maven and/or with CI/CD using Jenkins	IF CityDash

Table 10: Control Tasks for Atos Validation Experimentation

Table 11 describes the treatment tasks applied in SUPERSEDE IF and CityGO CityDash use cases:

Treatment	Treatment ID	Description	Tool (version)	KPIs	Use Case
Test Fixing (Descartes reports)	TF	Manual fixing of test suites (and code base) addressing the Descartes report about pseudo tested and partially tested methods. This treatment introduces the required changes in code (base, test) to remove these issues in new Descartes report.	Descartes (1.2.4)	K01 K03	IF
Test Amplification with DSpot	TADS	DSpot is executed with different configurations (i.e. Selectors, Amplifiers, etc) targeting different SUT test suites. Generated Amplified tests are added to the SUT test base.	DSpot (1.2.0)	K01 K03	IF
Test Amplification with Botsing	TAB	Botsing is executed with different configurations (i.e. framelevel, budget, etc) on selected SUT runtime exceptions. Generated crash replicating tests are added to the SUT test base.	Botsing (1.0.0)	K01 K03 K08	IF
Test Amplification with CAMP	TAC	SUT configurations are generated. Stress and sanity checks are executed on SUT deployed for several amplified deployment configurations.	CAMP (0.1.0)	K04 K05 K06	City Dash

Table 11: Treatment Tasks for Atos Validation Experimentation

8.B.2 Validation Target Objects and Tasks

In this industrial validation process, Atos uses as validation target SUT the following software entities: **SUPERSEDE IF** and **CityGO CityDash**, which have been described above in section 8.A.1. Several artefacts have been developed to support and assist the experimentation process:

Use case: *SUPERSEDE IF*

The building and delivery process of the IF component is managed by a CI/CD Jenkins job. Building is managed by Maven. Complementing this existing project management support other artifacts have been created to support the evaluation of the STAMP Test Amplification techniques:

- Gradle descriptor: required to evaluate the Gradle plugin for DSpot, Descartes and Botsing.
- Maven configurations for Descartes, but also for tools that compute some metrics. See pom files: pom_Descartes.xml, pom_pit.xml, pom_clover.xml, pom_jacoco.xml in the IF GitHub repository¹¹.
- Botsing setup:
 - required libraries to run IF have been manually collected from the Maven local repository.
 - Runtime stack traces have been collected, analysed and extracted from SUPERSEDE production server (platform.supersede.eu)
- scripts for configuring and executing STAMP tools in CLI, and for collecting and placing their results and execution logs into GitHub repositories. Other scripts for computing baseline KPIs have been developed. See run_dspot.sh, run_Descartes.sh, run_pit.sh, run_clover.sh, run_jacoco.sh in the IF GitHub repository.
- test suites have been prepared for experimentation: a restricted green test suite was configured, disabling those test cases with collateral undesired effects detected during the experimentation (i.e. those test cases that after amplification lead the SUT and its environment to an inconsistent/corrupted state).

11 <https://GitHub.com/supersede-project/integration/tree/stamp-treatment/IF/API/eu.supersede.if.api>

Use case: CityGO CityDash

Before STAMP the process for deploying CityGO CityDash was manual and based on scripts. In order to adopt STAMP techniques for deployment amplification in this use case we have developed the following artifacts:

- Docker based containers for CityGO CityDash components and dependencies
- Ansible based containers for CityGO CityDash components and dependencies
- Stress test (based on JMeter)
- Jenkins based CI/CD workflow

Docker based containers for CityGO CityDash components and dependencies

These artifacts are available in the Atos ARI GitLab repository (private access) at:

https://gitlab.atosresearch.eu/ari/stamp_Docker_citygoApp

The `/repo` folder contains the Docker images for

- the CityDash application (`/repo/Showcase`), which manage some dependencies such as Python and Apache web server, and deploys the CityDash web application.
- the database (`/repo/Postgresql`), which initiates the database with the CityDash schema
- Other backend services, such as MongoDB, and FIWARE Orion Context Broker and Cygnus.

This folder also contains the *DockerCompose* descriptor that launches the CityDash application and the other backend services included in above containers. This descriptor also configures the main components affecting the overall performance, namely the Apache web server and the Postgresql database (see Figure 4)

```

3  services:
4    db:
5      build: ./Postgres
6      container_name: my_postgres
7      networks:
8        - my-app-bridge
9      environment:
10       - max_connections=500
11       - shared_buffers=256
12       - port=5432
13       - POSTGRES_DB=citygo_malaga
14       - POSTGRES_USER=citygo
15       - POSTGRES_PASSWORD=5X6sdoq0!?az=v2aSX
16       - PGDATA=/var/lib/postgresql/data/pgdata
17     ports:
18       - "5432:5432"
19     expose:
20       - "5432"
21   web:
22     build: ./
23     container_name: my_web
24     networks:
25       - my-app-bridge
26     environment:
27       - StartServers=2
28       - MinSpareThreads=25
29       - MaxSpareThreads=75
30       - ThreadLimit=64
31       - ThreadsPerChild=25
32       - MaxRequestWorkers=150
33       - MaxConnectionsPerChild=0
34     volumes:
35       - /var/jenkins_home/workspace/atos_usecas
36     ports:
37       - "80:80"
38     expose:
39       - "80"
40     depends_on:
41       - db

```

Figure 4: Docker Compose Configuration for City Dash Postgresql and Apache Web Server: Performance Configuration

CityDash can be started from this repository with a single command:

`Docker-compose up -d`

Ansible based containers for CityGO CityDash components and dependencies

An Ansible based container configuration of CityDash has been created with the aim to evaluate in the future

if CAMP can be adopted for amplifying deployment configurations based on other technologies. These artifacts are located at this Atos ARI GitLab repository (private access) at:

https://gitlab.atosresearch.eu/ari/stamp_citygo_ansible

Stress test (based on JMeter)

Stress tests have been designed in order to identify CityDash deployment configurations that optimize the overall performance. These tests send a predefined number of concurrent requests to CityDash, simulating normal user operations from Android CityGO application. These tests have been designed in JMeter and are available at the folder `/stamp_Docker_citygoApp/Tests/version2` of the CityDash repository (private access) at:

https://gitlab.atosresearch.eu/ari/stamp_Docker_citygoApp

These stress tests can be executed by issuing the following command in an CLI:

```
./executeCamp.sh
```

Jenkins based CI/CD workflow

CityDash can be built, deployed in a given configuration and tested (using the aforementioned stress tests) using an automated CI/CD workflow implemented in Jenkins (see Figure 5). A Jenkins Docker container, which deploys internally the CityDash within another Docker container (Docker out of Docker) and tests it using JMeter, has been implemented.

The Jenkins instance is located in the `/Jenkins` folder of the CityDash repository (private access) at:

https://gitlab.atosresearch.eu/ari/stamp_Docker_citygoApp

It can be launched from the CLI issuing the command: `run.sh`

Then, it can be accessed in the url (restricted access through ssh tunnel) of the Atos server:

<http://supersede.es.atos.net:7777/>




All +						
S	W	Name ↓	Last Success	Last Failure	Last Duration	
		citygo	22 days - #11	22 days - #1	57 sec	

Figure 5: CityGO CityDash CI/CD Workflow in Jenkins

The Docker compose configuration that is instantiated and tested is located in this path:

```
./Jenkins/Docker-compose-mapped.yml
```

It instantiates the CityDash and backend services, including Postgresql, MongoDB, FIWARE Orion Context Broker and FIWARE Cygnus.

Tests that JMeter executes after Jenkins instantiates CityDash (see Figure 6) are located in the folder:

```
./Tests/citygo_tests/
```

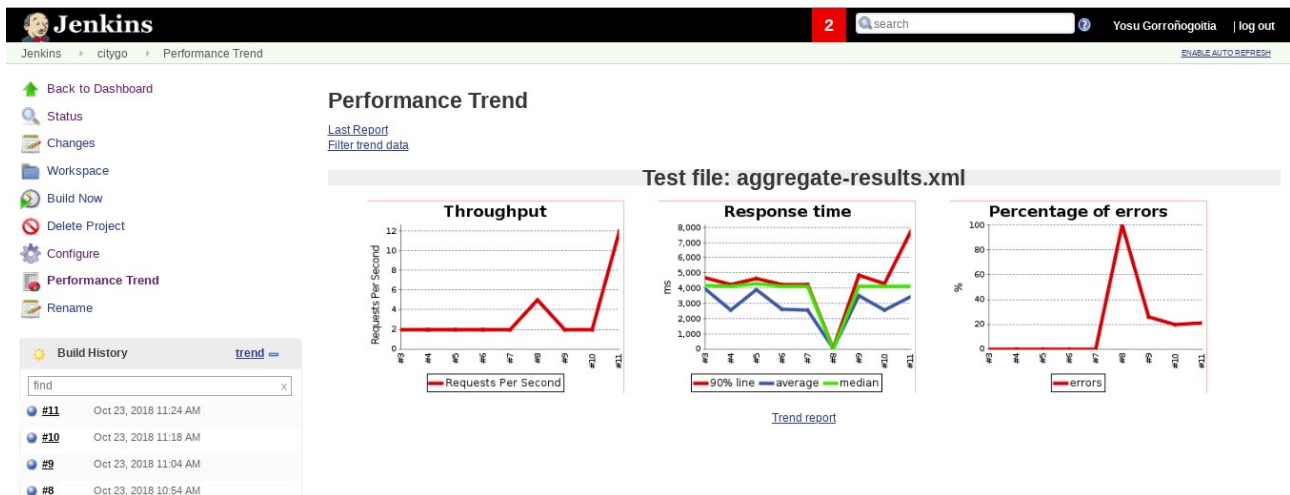


Figure 6: Performance Results of Stress Tests Applied by JMeter in Jenkins CI/CD to CityGO CityDash

CAMP image/container is obtained from DockerHub and instantiated as described in CAMP getting started#installation documentation¹². CAMP is launched for generating CityGO configuration as described in the CAMP CityGO Use Case¹³.

Generated configuration are then manually injected into the CI workflow through pushing them into the GitLab CityGO repository. Then, the CI process is manually triggered and stress tests executed. Test results are collected. The analysis of this results are performed by a script *reportTestResults.sh* located in the *stamp_Docker_citygoApp/Tests* folder.

8.B.3 Validation Method

The validation method adopted by Atos consists on a number of experiments, conducted independently, and suggested in D5.4 and D5.4 that combine different treatments (STAMP techniques and tools) and control (KPIs metric computation) tasks whose results will provide insights to investigate the effect of the STAMP technologies into the KPIs and to answer our validation questions. Table 12 defines the experiments conducted for each KPI. Tasks are identified by their identifiers (see tables 10 and 11 above).

Experiment	KPI	Use Case	Tasks								
Amplification of code coverage	K01	IF	Setup	CCC	CTQ	TF	CCC	TADS	CCC	TA B	CCC
Reduction of Flaky Tests	K02	IF	Setup	IFT							
- Amplification of mutation score - Reduce pseudo/partially tested methods	K03	IF	Setup	CMS CTQ	TF	CMS CTQ	TADS	CMS CTQ			

¹² <https://stamp-project.GitHub.io/camp/pages/setup.html>

¹³ <https://stamp-project.GitHub.io/camp/pages/citygo.html>

Amplification of #execution paths	K04	City Dash	Setup	CEP	TAC	CEP	
Amplification of detection of configuration related bugs	K05	City Dash	Setup	DCB	TAC	DCB	
- Amplification of #configurations tested - Reduction SUT deployment	K06	City Dash	Setup	CNCT CDT	TAC	CNCT CDT	
Amplification of the #crash replicating tests	K08	IF	Setup	CNCRT	TAB	CNCRT	

Table 12: Atos Experiments for KPIs. Sequence of Experimentation Tasks

The experiment for estimating the impact of STAMP technologies in the amplification of the code coverage computes the code coverage (CCC) before and after each STAMP treatment is applied. Atos applies the following treatments: a) manual Test Suite fixing (TF) for reducing Descartes detected pseudo/partially tested methods, b) test amplification with DSpot (TADS), c) test amplification with Botsing (TAB). Intermediate code coverage measurements will facilitate the analysis of the benefits of the different STAMP technologies on the amplification of code coverage.

Similarly works the experiment for amplifying the mutation score. Intermediate measurements for mutation score (CMS) and test quality (#pseudo/partially tested methods)(CTQ) are collected before and after treatments: a) manual Test Suite fixing, b) test amplification with DSpot. The former has a direct impact on test quality. The latter is supposed to increase the mutation score since it is expected to increase the test code coverage.

The experiment for estimating the amplification of the number of test execution paths alternates the measurement of these paths before and after the STAMP treatment is applied: amplification of tests configurations with CAMP. Similarly is conducted the experiment for estimating the amplification on the detection of configuration related bugs, and the one for estimating the amplification on the number of tested configurations and the reduction on the SUT deployment. In these cases, the metrics computed after applying the treatments are compared to the baseline control procedure, which uses manual or automated CI/CD.

Finally, the experiment for estimating the amplification of the number of crash replicating tests computes the number of those tests generated by the application of the STAMP treatment: test amplification with Botsing.

8.B.4 Validation Data Collection and Measurement Method

Data/Metrics resulting of the experiment tasks described above are collected by adopting different methods depending on the task nature:

- Data/Metrics produced as output by the execution of baseline (e.g. Clover, Jacoco, PIT, etc) or STAMP tools (Descartes, DSpot, CAMP, Botsing) are pushed onto the corresponding STAMP repositories located in GitHub for collecting industrial use case experiments (see Table 13):

Tool	Output Type	GitHub repository	Path
Clover	Coverage report (HTML)	Dspot-usecases-output ¹⁴	atos/supersede/IF/ CodeCoverage/Baseline/clover atos/supersede/IF/ CodeCoverage/Treatment/ clover
Jacoco	Coverage report (HTML)	dspot-usecases-output	atos/supersede/IF/ CodeCoverage/Baseline/ jacoco-ut atos/supersede/IF/ CodeCoverage/Treatment/ jacoco-ut
PIT	Mutation report (HTML)	Descartes-usecases-output ¹⁵	atos/supersede/if/mutation- score/baseline/pit/ atos/supersede/if/mutation- score/treatment/pit
DSpot	Test Amplification report Amplified tests Execution logs	dspot-usecases-output	atos/supersede/IF/DSpot
Descartes	Mutation report (HTML) Test issues report (HTML) Execution logs	Descartes-usecases-output	atos/supersede/if/mutation- score/baseline/Dcartes/ atos/supersede/if/mutation- score/treatment/Dcartes
CAMP	Deployment configuration Stress test results	confampl-usecases-output ¹⁶	atos/citygo
Botsing	Crash replicating tests Execution logs	evocrash-usecases-output ¹⁷	Atos/Supersede/IF/botsing

Table 13: STAMP Github Repositories and Paths for Reporting Evaluation Experimentation Data

The results of the different conducted experiments for same tasks (baseline, treatment) are placed within these repositories and paths into folders named with the experiment timestamp, using the following schema: YYYYMMDDhhmm

- Code/Test bases modified manually (in reaction to STAMP tools reports) or extended by amplified test generated as output by the STAMP tools (DSpot, Botsing) are pushed onto the corresponding Use Case repositories (see Table 14) located in GitHub and on Atos GitLab (restricted access):

Use Case	Repository	Branch	Path
Supersede IF	supersede-project/	stamp-baseline	IF/API/eu.supersede.if.api

¹⁴ <https://GitHub.com/STAMP-project/dspot-usecases-output>

¹⁵ <https://GitHub.com/STAMP-project/Dcartes-usecases-output>

¹⁶ <https://GitHub.com/STAMP-project/confampl-usecases-output>

¹⁷ <https://GitHub.com/STAMP-project/evocrash-usecases-output>

	integration ¹⁸	stamp-treatment	
CityGO CityDash	citygo stamp_Docker_citygoApp ¹⁹	Master	CityGO/ Citygo/www/ShowcaseServer/ demo_site

Table 14: Atos Code Base Repositories for Use Cases (SUPERSEDE IF and CityGO CityDash)

- KPI metrics not directly reported on the baseline/STAMP reports are computed by adopting the methods suggested in D5.3. Time computation is recorded by the scripts launching the baseline and STAMP tools and included in the execution logs that are pushed onto the usecases-output repositories in GitHub.

8.C Validation Results

8.C.1 KPIs report

These section reports and analyse the results obtained after applying the experiments, for each KPI, described, as a sequence of baseline and treatment tasks, in Table 12.

K01 - More execution paths

We apply the STAMP test amplification techniques to the IF component, following the procedure described in the row for K01 of Table 12 and Table 15 collects the code coverage measured with Clover and Jacoco (configured in Maven executions) before (baseline) and after applying different STAMP techniques (treatments)

Metric	Measure Tool	Base line	Treatment				Delta
			Descartes (Test manual fixing)	DSpot	Botsing	CAMP	
Code coverage	Clover	47.5%	46.3%	52%	No crash replicating test generated	N/A	9.47%
Code coverage	Jacoco	51%	50%	54%	No crash replicating test generated	N/A	5.88%

Table 15: K01 Code Coverage Metrics for IF Use Case

CAMP technique, described in D5.3 with a potential effect on K01 is not applicable here, since the deployment of the SUPERSEDE platform under different CAMP configurations have not been considered because its complexity. Additionally, it is not expected any CAMP application influence because the different deployment configurations do not modify the SUT code execution.

An analysis of results draws the following comments:

Code coverage computed by Clover and Jacoco are different, but the tendency observed for both measurements after applying the STAMP technics is similar;

¹⁸ <https://GitHub.com/supersede-project/integration>

¹⁹ https://gitlab.atosresearch.eu/ari/stamp_Docker_citygoApp

The refactoring of the IF test suites for addressing the Descartes report on pseudo/partially tested methods have a negative impact on code coverage, passing from 47.5% to 46.3% (-2.53% decrement). We cannot identify yet the reasons for this coverage reduction.

The test amplification by DSpot increases the code coverage in a significant 12,31% (from 46.3% to 52%). DSpot amplification was applied to the test base resulting from the Descartes-driven manual refactoring (see Tables 10 and 11 for a description of experimentation tasks applied). DSpot generated 26 test suites and 1127 new test cases that were incorporated to the IF test base, increasing the test cases to a total of 1294. There is a significant number of generated test cases that are redundant in the sense of testing similar code base APIs, but with different test inputs. Additionally, up to 433 new generated test cases were discarded, since they didn't pass successfully. This was caused because these test cases were expecting, when asserting the results, a specific snapshot of the SUPERSEDE datasource, which was valid by the time these test were created. In turn, this situation occurs because the generated tests are not restoring the data-sources to the initial state before test application, as the test manually created do. Once these failing test cases were discarded, the new test base (with 861 valid tests) was green and the code coverage could be computed.

The potential side effect of Botsing (former Evocrash) on increasing the code coverage could not be determined, since Botsing failed on generating crash replicating test cases for IF (see report on K08 in following paragraphs).

K02 - Less flaky tests

During the development of the SUPERSEDE platform in general and the IF library in particular, the development team faced runtime situations detected by randomly failing tests that were initially qualified as potential flaky tests, attending to the definition that these tests randomly passed or failed without modifications in the SUT code and/or the tests code. However, after a deeper analysis, the development team concluded that these failures, which are not intermittent but permanent as long as the runtime cause producing the test failure is not fixed, cannot be tagged as flaky. Among the causes of eventual test failures, the team detected the following reasons:

- Inadequate test setup: tests used (or referred to) input artefacts that got eventually unavailable or incompatible (after modifications);
- Test collateral effects: some tests modified the SUPERSEDE internal state (i.e. its internal storage) without recovering it after the test execution, what led following tests asserting the state to fail;
- Temporal unavailability of some SUPERSEDE backend services;
- Network failures.

IF test suites were manually improved to minimize the impact of test collateral effects, what reduced significantly the number of unexpected test failures. The other causes of failures cannot be totally discarded, and upon failing tests, they are immediately checked.

To conclude, at to this reporting period, **SUPERSEDE IF use case is not facing flaky tests.**

Metric	Measure Tool	Baseline
Number of flaky tests	Manual	0

Table 16: K02 Code Coverage Metrics for IF Use Case

K03 - Better test quality

We apply the STAMP test amplification techniques to the IF component, following the procedure described in the row for K03 of Table 12. Results on K03 metrics for control (i.e. baseline) and STAMP treatments (i.e. tool usage) are collected in Table 17. An analysis of results draws the following comments:

Mutation score was increased a **10.26%** (from 39% to 43%) when the test cases (and code base) are

manually refactored to address the Descartes issues report. These refactoring reduced the number of issues from 21 to 9 (in case of pseudo tested method) and from 21 to 3 (in case of partially tested issues).

Mutation score was increased a **13,95%** (from 43% to 49%) when tests amplified by DSpot (see section above on K01) were incorporated into the baseline test base.

The total increment on mutation score was 25,64% (from 39% to 43%).

Metric	Tool	Baseline	Treatment		Delta
			Descartes (Test manual fixing)	DSpot	
Mutation Score	PIT	27%	43%	49%	81.48%
	Descartes	39%	43%	49%	25.64%
#Pseudo tested methods	Descartes	21	9	14	33.33%
#Partial tested methods		21	3	2	97.49%

Table 17: K03 Code Coverage Metrics for IF Use Case

The manual refactoring of both code and test bases, in reaction to the Descartes report on pseudo/partial tested methods, significantly reduce the number of reported issues, passing from 21 to 9 the number of remaining pseudo tested methods (**57.14%**), and from 21 to 3 (**85.71%**) the number of partial tested ones. It took a morning (around 4 hours) a IF developer to complete this task, including the analysis of the Descartes report and the code and test bases, and the implementation of refactorings. **The Descartes issue report was perceived as quite useful and straightforward tool to improve the code base quality.**

Descartes reported an increment (of 4 issues) in the number of pseudo tested methods when we compare the Descartes analysis on the refactored test base with the similar analysis on the extended test base (i.e. after adding the DSpot amplified tests). A visual inspection of both reports does not draw clear conclusions. Some of the issues reported in the first report (before adding the DSpot amplified tests) vanished because added tests detected extreme mutation in those methods. Contrarily, new issues were reported in methods now tested with DSpot generated tests. It seems DSpot design some tests that are not extreme mutation safe. Despite this collateral effect in test quality, **DSpot largely improves the mutation score.**

CAMP related KPIs

We apply the CAMP configuration amplification techniques to the CityGO CityDash component, following the procedure described in the rows for K04, K05 and K06 of Table 12.

Table 18 collects the stress tests results obtained by running those tests in the CI for the 10 different deployment configurations for CityDash that were generated by CAMP.

- *#successful/failed requests* gives the number of successful/failed requests processed by CityDash during the stress test.
- For *failed requests* in parenthesis is given the number of unique stress tests failing.
- *#Internal requests* gives the number of additional requests internally propagated to internal components
- *#unique remote requests* gives the number of additional unique requests externally propagated to external services

Configuration	#successful requests	#failed requests	#Internal requests	#unique remote requests
1	250	0	26	89
2	249	1 (1)	29	83
3	250	0	24	90
4	240	10 (4)	28	90
5	250	0	28	82
6	250	0	20	83
7	250	0	23	88
8	249	1 (1)	23	79
9	250	0	23	80
10	247	5 (3)	18	90

Table 18: Stress Tests Statistics on CityDash for Different CAMP Generated Deployment Configuration

K04 - More unique invocation traces

CityDash deployment configurations generated by CAMP do not modify the component assembly of CityDash but the parameterization of CityDash Web container for optimal performance. Besides, CityDash is a monolithic system (in opposition to a microservice system) as defined in D5.3. For these reasons, we are not expecting that the different deployment configurations have any influence in the total number of execution paths transverse by the tests. However, from Table 18 we observe that the different CityDash configurations give varying results w.r.t the number of issued unique remote requests during the stress tests, which, in this use case, constitutes a measure of the *K04 number of invocation paths*. Different configurations offers results varying from 79 to 90 unique external service invocations (**13.92%** of variability). We conclude that **CAMP generates configurations that resulted in a higher number of unique invocations to external services**. Similarly, the different configurations influences the number of internal inter-component requests, ranging from 18 to 29 (**61.11%** of variability)

Metric	Baseline (suboptimal configuration)	Treatment (optimal configuration)	Delta
#nvocation paths (unique external service invocations)	79	90	13.92%
#nvocation paths (inter-component requests)	18	29	61.11%

Table 19: K04 Increment in Invocation Traces Metric for CityDash Use Case

K05 - System specific bugs

Similarly, we are not expecting that the different parameterization for performance of the Web container, computed by CAMP, may have any impact on the detection of configuration specific bugs, defined as

execution failures in code branches that are reported into the system runtime logs as exceptions. However, from the analysis of results in Table 18 we observe that for some configurations, the number of observed request failures -reported in the Web container logs- increases, compared to other configurations where no failures are detected. Therefore, if we include in the definition of configuration specific bugs the number of stress test failing (the number of failing requests) we can conclude that **the some underperforming configurations spot few errors that are not detected in other configurations** (which are optimal for the workload tested).

Metric	Maximum number of failing stress tests
#configuration specific bugs	4

Table 20: K05 Number of Configuration Specific Bugs Metric for CityDash Use Case

K06 - More Configuration/Faster tests

CAMP amplified up to a number of new 10 configurations. Before using CAMP only one CityDash default configuration was available. CAMP does not provide any additional support to automate the deployment of generated configurations apart from a script that launches the generation of images. As explained in CAMP documentation for CityGo use case²⁰, generated configuration can be deployed by issuing a Docker compose command. Similar approach is adopted in CityGO Jenkins CI/CD workflow. The K06 deployment time reported in Table 21 corresponds to the time measured in the CI/CD workflow.

Metric	Value
#new configurations tested	10
#deployment time	17 s

Table 21: K06 Metrics for CityDash Use Case

K08 - More crash replicating test cases

A manual analysis of SUPERSEDE production server (platform.supersede.eu) logs detected a total of 21 different runtime exception stack traces, producing a total of 7 different triggered exceptions (see Table 22). Out of these exception stack traces, 12 corresponds to the target IF component.

javax.management.InstanceAlreadyExistsException
org.springframework.web.client.HttpServerErrorException
org.springframework.web.client.HttpClientErrorException
java.lang.IllegalArgumentException
org.apache.axis2.AxisFault
java.lang.NullPointerException
java.lang.StringIndexOutOfBoundsException

Table 22: Common Runtime Exceptions in IF Use Case

The different experiments with Botsing on 3 of these IF related runtime exceptions have ended up without any crash replicating test generated. We chose 3 target exceptions because that are reproducible and for which we created test cases manually.

²⁰<https://stamp-project.GitHub.io/camp/pages/citygo.html>

A discussion with the Botsing development team provides the following insights about the current limitations of Botsing to replicate IF exception stack traces:

- Botsing relies on a Security Manager that imposes some limitations to the classes they can test, in order to avoid unwished events. It seems that most of IF proxy classes, which are the main entry point to the IF behavior causing reported exceptions, have the characteristics (e.g. *java.net.SocketPermissions*) of the target classes that the Botsing Security Manager is banning. The analysis done in Fraser, G., & Arcuri, A. (2014)²¹ should provide insights to progress on addressing this issue.
- Botsing has still not implemented a significant number of challenges identified in the large-scale evaluation conducted by its development team. It is likely that new releases of Botsing will be able to reproduce some of the IF runtime exceptions.

Metric	Value
#new crash replicating configurations	0

Table 23: K08 Metrics for IF Use Case

8.C.2 - Qualitative Evaluation and Recommendations

Descartes

Descartes is a very mature tool. Atos evaluation team has not faced any issue when it has applied Descartes to the IF use case during this evaluation period. **Operability is straightforward thanks to its Maven and Gradle interface and the visual support in the Eclipse IDE.**

Descartes configuration is also simple thanks to an available and very complete documentation.

Descartes issue report has proven very useful and easy to understand. The link between the detected pseudo tested and partially tested methods and the test cases enabled the development team to analyse and inject required refactorings in the code and test base in less than four hours. The detection of partially tested methods for those that returned boolean values upon success encouraged us to reconsider this strategy returning an enum of returned HTTP codes. The detection of pseudo tested methods related to the deletion of entities in the backend storage, spotted to us the lack of such verification, and in particular help us to identify a flaw in the test cleanup, since the entities created by test cases were not removed upon test completion in order to restore the storage consistency.

Performance has also been improved. Descartes completed its execution over the entire IF test suite in less than an hour, compared to more than two and half hours it took to complete in previous evaluation phase over the same test base. The qualitative evaluation of Descartes is summarized in the Table 24.

Descartes	Sub-Characteristics	Phase 1	Phase 2
Functional suitability	Functional Completeness	4	5
	Functional Correctness	4	4
	Functional Appropriateness	4	4
Compatibility	Co-existence	N/A	4
Perf. Efficiency	Time-behavior	3	4
Usability	Appropriateness recognisability	4	5
	Learnability	4	4
	Operability	3	4
Reliability	Maturity	4	5
Portability	Installability	4	5

Table 24: Quality Model Based Evaluation of Descartes Tool

²¹ Fraser, G., & Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using evosuite. ACM Transactions on Software Engineering and Methodology (TOSEM), 24(2), 8.

DSpot

The manual inspection of the tests amplified by DSpot provides insights for the Atos SUPERSEDE IF development team to improve the test base. In particular, **DSpot has largely increased the test base (up to 694 new valid test cases were added) on what concerns the assertions of the API guards under wrong API usage or provided input.** Many amplified tests deliberately inject invalid input values into the API and verify the throwing of expected exceptions. This robustness testing approach was not conceived before by IF developers; only the optimistic API behaviour was asserted.

DSpot has gained significant maturity in the last 6 months and is a much more solid than the version evaluated in D5.5. Although DSpot could be applied to most of the packages included into the IF API, some issues arose and were reported in its tracker²². In particular:

- A number of reported issues were caused because of wrong input values and/or format provided as DSpot parameters, which were parsed but not verified before proceeding with the amplification, leading to apparently unrelated crash exceptions. Post-mortem deeper analysis determined the original cause. It was suggested to improve the verification of inputs (and reporting to user if not passed) before proceeding with test amplification.
- For some IF target test cases, DSpot crashes raising memory exceptions (OutOfMemory, StackOverflow) that cannot be sorted out by increasing the JVM memory.
- Clover selector does not work for IF target test cases. Results have been obtained using the Jacoco selector.
- For other IF target test cases, DSpot terminates reporting a failed amplification, without exceptions triggered and without reporting additional failure information (or the causes the amplification failed), despite verbose mode is enabled.
- In other test cases, timeouts were reported. They were fixed by increasing the timeout parameter in DSpot input configuration.
- Upon a test failure, DSpot stopped execution, instead of proceeding to the next target tests. This behavior was reported and fixed by the development team, improving the usability of DSpot over a wide range of target test cases.

Despite these reported issues, DSpot could work for more the IF test cases, thanks to its development team, who fixed a significant number of them. Moreover, **other aspects have been significantly improved w.r.t the previous evaluation phase**, including:

- **performance:** in previous evaluation, DSpot executions took over more than 12 hours to complete with a single run. The new releases evaluated in this period run over the same IF target test cases for no more than 3 hours in the more time consuming case.
- **operability:** It offers more client interfaces to end users, including Eclipse IDE, Maven and Jenkins interfaces, and significant higher number of configuration parameters. Documentation has been largely improved.

The qualitative evaluation of DSpot is summarized in Table 25.

DSpot		Phase 1	Phase 2
Functional suitability	Functional Completeness	4	4
	Functional Correctness	4	4
	Functional Appropriateness	4	4
Compatibility	Co-existence	N/A	3
Perf. Efficiency	Time-behavior	2	4
Usability	Appropriateness recognisability	4	4
	Learnability	4	4
	Operability	2	4

22 https://github.com/STAMP-project/dspot/issues/created_by/jesus-gorronogoitia

Reliability	Maturity	3	4
Portability	Installability	4	5

Table 25: Quality Model Based Evaluation of DSpot Tool

CAMP

The amplification of deployment configurations is a key requirement for Atos CityGO use case. CAMP has a huge potential for Atos in addressing this requirement. **CAMP significantly simplifies the generation of new configurations for Docker container based deployment.** CAMP generates configurations that modify the parametrization for the CityGO deployment containers (e.g. Web Container, the database) that satisfy expressed constraints (among key parameters). **But it does not explore yet the parameters solution space, in seeking for optimal configurations with regards to a target optimization problem (i.e. performance), driven by an objective (or fitting) function model,** as suggested in the Atos CityGO use case. **In this sense, CAMP is still useful, but the exploration of the parameters solution space seeking for an optimal configuration remains a manual, tedious and time consuming task.**

Current CAMP technology seems to be more promising for the generation of deployment configurations that change their functional requirements, replacing some baseline dependencies with other functional equivalent ones. This scenario will be explored for CityGO CityDash in the next evaluation period, but the replacement of baseline technologies requires completely different deployment Docker descriptors and other artifacts, which should be generated manually. Therefore the potential of CAMP in this regards is unclear. For instance, replacing Apache Web container for NGINX may require completely different Docker descriptors. Similarly, the replacement of the database from PostgreSQL to MySQL (or another RDBMS engine) may require the reengineering of the database schema and their SQL queries.

CAMP management and generation of Docker configurations is fast and precise. CAMP now supports the instantiation of CityGO for a given amplified configuration, but this feature is already supported by our CI/CD process. Therefore, the potential of CAMP for CityGO instantiation still needs to be clarified.

CAMP documentation has been extensively improved. Current documentation is quite complete, easy to understand and useful. **CAMP adoption (installation and usage) is also straightforward.**

CAMP	Sub-Characteristics	Phase 1	Phase 2
Functional suitability	Functional Completeness	3	3
	Functional Correctness	2	3
	Functional Appropriateness	3	3
Compatibility	Co-existence	N/A	4
Perf. Efficiency	Time-behavior	4	5
Usability	Appropriateness recognisability	3	3
	Learnability	4	5
	Operability	4	5
Reliability	Maturity	2	3
Portability	Installability	5	5

Table 26: Quality Model Based Evaluation of CAMP Tool

Botsing

Botsing is a promising tool for Atos use cases. Its ability to generate crash replicating tests from reported exception stack traces is quite well appreciated by Atos developers. These tests could provide valuable insights leading to the understanding of the causes of the issue and speeding up the development of patches.

Botsing seems not compatible with the IF component use case, because of the network access restrictions imposed by its Security Manager. Further investigation with the development team is required to understand whether or not these restrictions can be relaxed. IF offers networks mediation between backend services. Most of its test cases are therefore system or integration tests. If Botsing restrictions cannot be relaxed, the IF use case will be out of scope.

Botsing performance could not be determined for a successful case. In all the experiments the search budget was exhausted. Increasing the budget did not lead to a successful test generation. Further experiments with new Botsing releases are required to determine its efficiency and performance.

Botsing documentation is correct but concise. It is lacking of examples that users could use to get familiarized with the tool. Botsing interface is straightforward, easy to understand and use by users.

Botsing	Sub-Characteristics	Period 1	Period 2
Functional suitability	Functional Completeness	N/A	3
	Functional Correctness	N/A	1
	Functional Appropriateness	N/A	3
Compatibility	Co-existence	N/A	4
Perf. Efficiency	Time-behavior	N/A	2
Usability	Appropriateness recognisability	N/A	4
	Learnability	N/A	3
	Operability	N/A	3
Reliability	Maturity	N/A	3
Portability	Installability	N/A	4

Table 27: Quality Model Based Evaluation of Botsing Tool

8.C.3 Answer to Validation Questions

This section provides Atos use case specific answers to the validation questions introduced in section 6.

VQ1 - Can STAMP technologies assist software developers to reach areas of code that are not tested?

Partially yes, they can.

In *IF component use case*, we have got a significant improvement, **9.47%**, on the number of tested execution paths, measured through the code coverage metric. This is achieved after applying two STAMP treatments: a) test and code base refactoring for addressing the Descartes test quality report, and b) the test base amplification with DSpot. On the contrary, we have not got any increment in the number of crash replicating tests generated by Botsing, due to the restrictions imposed by the Botsing security manager (see K08 section). Nonetheless, we are not expecting any significant contribution to increase the number of untested code lines as the number of crashing exceptions reported by runtime logs is relative small.

In our opinion, the most relevant STAMP technology for addressing this VQ1 is the combination of Descartes and DSpot, particularly the latter. Nonetheless, DSpot is still producing a significant high number of (quite similar) amplified test cases. Further investigation should address this issue in the following directions:

- reducing the number of generated tests cases that show high similarity in terms of the API methods ;that are included into the test cases with similar API choreography, and that are equally contributing to the increment on the code coverage;
- rank higher those generated test cases that contribute most to increase the code coverage.

For *CityGO CityDash use case*, however, as we argued above in K04 section. we cannot consider that the

increment we have got for the number of unique invocation traces in CityGO CityDash use case, namely 13.92% and 61.11% for unique external invocation traces and inter-component requests, is a measure of the number of new code lines or branches executed by existing tests, which is the target of the VQ1.

VQ2 - Can STAMP tools increase the level of confidence about test cases?

Definitely, yes

In *IF component use case*, we have got significant STAMP technique-driven assistance to increase the observe quality of our test cases in what concerns their test verdict confidence. We could increase the mutation score up to **25,64%** with the combined STAMP treatment for addressing the reported Descartes issues and amplifying the test base with DSpot. In addition, the number of method issues reported by Descartes (they give a direct metric on the quality of the test and code bases) decreased a **33.33%** and a **97.49%** for the number of pseudo and partially tested methods, respectively. Significantly enough was the DSpot management, in amplified tests, of the exceptional code behavior. Test management of unfavourable code behaviour is perceived by IF development team to largely increase their test confidence.

Our test confidence related to false positives seems to lay on the side of the test environment and test preparation rather than on the existence of flaky tests (which are absent in our IF use case). Therefore, we cannot draw any conclusion on the utility of the STAMP techniques to reduce false positives.

VQ3 - Can STAMP tools increase developers confidence in running the SUT under various environments?

Partially yes

CityGO CityDash use case holds a strong sensibility on its execution environment on what concerns its experience perceived by users, both through the CityDash portal and the Android App. In this context, STAMP CAMP tool is expected to offer great potential for the optimal configuration of the CityGO backend, increasing its team confidence during its delivery and maintained operation.

We have got promising results to increase our confidence in CityDash under different workloads requiring optimal configurations. First, we observe significant variability (up to **13.92%**) on the number of unique invocation traces, depending on the CityGO deployment configurations. This behaviour shows sufficient sensibility as to determine optimal deployment configurations for a given workload. Similarly, we found configurations that shows up to 4 configuration-specific-bugs (i.e. understood as the number of unique SUT requests facing runtime failures), so we can identify configuration that are reliable vs those that are not. Finally CAMP largely simplify the generation of more compatible configurations (up to 10 in our experiments) that assist developers to experiment them all in order to detect more invocation traces and bugs.

VQ4 - Can STAMP tools speed up the test development process?

Partially yes

In case of the *CityGO CityDash use case*, we can automate (via CI/CD Jenkins or CAMP) the instantiation and testing of new CityDash instances, much faster than manually. However, the kind of configurations CAMP generates are still suboptimal as they are not maximizing a given performance objective function/model. In that sense, there is not that much difference between adopting CAMP or a standard CI/CD workflow (see K06 section). Once CAMP can explore the configuration space in seeking for optimal configurations, it will outperform existing automated CI/CD workflows.

In case of the *IF use case*, we failed to generate crash replicating test cases. For this reason, the question whether Botsing can speed up the development of crash replicating test cases is still undecidable.

8.C.4- Next Steps for Validation

We conclude Atos UC validation report with a prospect of the next-term objectives and actions for evaluation.

Descartes

The IF test and code base will be iteratively improved to get rid of all those issues reported by Descartes in the last treatment experiment. The accompanying PIT mutation report will be further analysed in order to identify those code packages that are significantly uncovered by the tests. Based on this analysis additional test cases will be engineered for these target packages and both code coverage and mutation score computed again. Further code/test base refactorings will be conducted for reported issues. This iterative process will continue until reaching a relevant threshold in code quality, measured by the code coverage and code mutation metrics.

Experimentation with other Descartes clients, including Gradle, Jenkins and Eclipse IDE will be conducted. Descartes analysis will be included as part of the QA process in the IF CI/CD workflow.

DSpot

We have planned to conduct a manual analysis of test amplified by DSpot in order to:

- improve our understanding on the kind of tests and assertions injected by DSpot so we can incorporate them into our test base, although reducing significantly the total number of test cases;
- provide feedback to DSpot development team, helping them to reduce the number of redundant/overlapping generated tests.

We are also planning to conduct additional experiments on IF test base with different amplifiers, aiming at obtaining different kinds of tests. Previously failing experiments will be re-conducted with new amplifiers and selectors, using new DSpot releases. Particularly interested in the adoption of Clover selector to improve the branch coverage.

Additional existing DSpot clients, including Maven, Gradle, Jenkins and the Eclipse IDE will be adopted for conducting these new DSpot evaluation experiments. We will explore possible adaption of DSpot into our IF CI/CD workflow.

CAMP

The following main objectives and associated action plans are identified for CAMP for the next evaluation period:

- Evaluation of CAMP ability to explore the configuration space, seeking for optimal configuration w.r.t. a given non-functional property, such as performance, reliability, etc. Exploration is guided by means of objective/fitness function/model provided by the domain expert. Further discussion with CAMP team is required in order specify this scenario.
- Usage of CAMP for amplifying deployment configurations that diverge on the Web containers (Apache, NGINX, etc) and data sources (PostgreSQL, MySQL, etc) and OS (different linux distros).
- Adoption and evaluation of WP2 Web containers for testing CityGO CityDash using Selenium and JUnit 5, so test cases could be written in Java.

Botsing

In the following evaluation period, we will address with the Botsing team the management the configuration of the Botsing security manager restrictions. We will conduct experiments for those IF runtime exceptions for which Botsing could not generate crash replicating tests.

9. Tellu Use Case Validation

Tellu Use Case Highlights

- All four main STAMP tools has been tested on the TelluCloud use case.
- Descartes is working well on TelluCloud unit tests, and has led to fixes of many important issues of pseudo-testing, where the test was not really verifying all the functionality it was meant to verify.
- CAMP has made it significantly easier to bring up working versions of TelluCloud using various different configuration, and to run system tests on these configurations.
- TelluCloud stack traces from one month of multiple running instances of the service has been collected and analysed

9.A. Use Case Description

9.A.1 Target Software

(Note: This description is based on previous descriptions given in D5.4 and D5.5.)

Tellu provides **TelluCloud**, a cloud platform for collecting and processing data with a focus on IoT. Tellu currently operates two main commercial instances of the service, both of which are multi-tenant and host multiple service providers. One is running on a Norwegian hosting provider and one is running in Amazon's cloud infrastructure.

After refactoring from a more monolithic system, the platform now has a micro-service architecture. In total there are currently 12 microservices. All services are coded in Java with the exception of the web and main API, which are built on Grails/Groovy, and which are not included in the STAMP validation. The use case for STAMP includes the whole TelluCloud system, with the exception of the web/API components.

For unit test amplification we selected three key projects to focus on, which represent different levels in the source code hierarchy:

- **TelluLib**: In this library we keep code of a general nature, for use in any Tellu project. All TelluCloud modules have a dependency on this code. The code is well-suited to unit testing and had a good number of unit tests to begin with, running very quickly.
- **Core**: This library contains common TelluCloud service code, and is used by many of the micro-service projects. It contains the common domain model and several data access levels. It originated from the more monolithic main application of TelluCloud 2.x, with features gradually being refactored into service-specific projects when possible. This is still the largest of all TelluCloud projects (in lines of code). It has unit tests coming from 2.x, but some of these are actually integration tests which take some time to execute.
- **Filterstore**: This is one of the micro-services. It is arguably the most important and most complex of the services, having a central position of receiving normalised data from edges, doing much of the initial processing and distributing the data to a number of other services. It is strongly dependent on Core. It has few, high-level tests.

Here is an overview of the source code metrics for the three projects before running this phase of tool tests:

	TelluLib	Core	FilterStore	Total
Java classes	107	357	45	509
Lines of Code	7638	25112	4662	37412

Unit tests	75	85	4	164
Coverage	53.6%	29.4%	35.8%	41% *
Mutation score **	48%	14%	54%	

* Total coverage is computed by Clover after running the tests of all three projects. High-level tests in FilterStore invokes code in TelluLib and Core, bringing their total coverages up to 58% and 34.2% respectively.

** Mutation score computed by Descartes, details below.

Table 28: Key Metrics of Tellu's Use Case Target Software

9.A.2 Experimentation Environment.

The code is made up of around 30 separate Maven projects, each in its own GIT repo. The repos are hosted on GitHub, but with private access since TelluCloud is not open-source. Tellu uses Maven as build system and Jenkins for CI.

Tellu developers use different operating systems and IDEs on their developer machines. Both Windows, Mac and Linux is in regular use. Unit testing and experiments with Descartes, DSpot and Botsing will be done locally on developer machines with Windows 10 and Maven. CAMP experiments are done on the CI server.

9.A.3 Expected Improvements

Splitting up the system into micro-services has made the development and maintenance of each part more manageable. However, it has also introduced new challenges. The complexity of the system as a whole has increased, especially with respect to configuration and deployment, as there are many more parts to deploy, including queues and other infrastructure. It became very important that each micro-service has well-defined APIs and protocols. In addition to unit testing, testing of each micro-service and of the system as a whole became paramount.

Tellu's overall objective is to improve and automate testing and logging for TelluCloud 3.x, to ensure its correctness and robustness in a cost-effective manner so that Tellu can continue to develop and deploy new versions without introducing bugs which disrupt the services. Tellu's objectives in STAMP are described based on the three forms of amplification in the project.

For unit testing, Tellu's objectives are to increase the number of JUnit tests for the TelluCloud source code, increasing test coverage, as well as improving the quality of the tests. The first part is easy to quantify with tools measuring test coverage. For test quality, Tellu's initial objective was to learn about state of the art practices and tools and techniques such as mutation testing. With mutation testing Tellu gets a metric to help quantify test quality. The objective is to compute this metric on TelluCloud projects and improve it. This should be integrated into the build chain.

For runtime test amplification, Tellu's initial objective is to get tools and methodologies to test the TelluCloud services at runtime, to check that running services behave according to specifications. More specifically, it is an objective to improve the quality of logging done by the system, and to get intelligent monitoring of the logs, to get automatic discovery and analysis of problems and inconsistencies.

TelluCloud is made to be deployed in different configurations. For deployment and test, it is important to be able to deploy both micro-services and full systems in a quick and easy manner. For production, it is important for Tellu to be able to move to different forms of cloud hosting depending on technological developments, pricing and customer needs. Tellu foresees the need to continue to support at least two different production deployments. Some customers are best served with an instance running in Amazon's cloud infrastructure. But Tellu is also heavily involved in e-health in Norway, and these customers want the

data to be stored in Norway, requiring a more local solution. This makes configuration test amplification important for our case. The first objective here is to orchestrate and automate deployment of the system, to facilitate efficient and automated testing. The next objective is to amplify configurations and test inputs, generating new variations of configurations based on existing ones. Finally, there are objectives to run different types of tests on the system:

- Functional tests: Verify correct service and system outputs based on inputs.
- Quantify performance and scalability, to find optimal configurations (performance vs price) and verify correct distribution and load balancing.
- Stress testing, ensuring that the response times and throughput latencies are within specified limits.

Finally and common to all forms of testing amplification, is the objective to operationalize the tools and technologies.

9.A.4 Business Relevance

Tellu provides TelluCloud to service providers and other partners in different domains, for integration in their solutions. Our current commercial focus is e-health, communal care and personal safety and security domains. With the move to a micro-service architecture Tellu wants to release new versions of separate services often. Strong automated testing of both the service and the system is a prerequisite and Tellu's strategic goal is to move towards DevOps. Tellu's STAMP use case supports this goal.

9.B. Validation Experimental Method

9.B.1 Validation Treatments

Control and treatment tasks have been defined to measure all KPIs. The KPIs are described in D5.3, with tools and strategies for measurement. The following table gives tasks based on KPIs for Tellu's validation experiments, and maps these to the STAMP tools to be validated. The resulting experiments are described in the next sections.

Metric	KPIs	STAMP tool	Treatment for measuring
Code Coverage	K01	Descartes DSpot Botsing	Code Coverage is computed using Clover, on each of the three selected use case projects. <ul style="list-style-type: none"> • A common baseline is measured on each use case project before experiments. • A tool-specific result coverage is measured after concluding experiments with that tool, on each use case project modified by the tool. Each tool is applied to the same baseline, so effects are not cumulative. The relative change in coverage is computed. • Changes made by each tool are merged, and a cumulative coverage effect is measured for each use case project modified by multiple tools.
#Flaky Tests	K02	-	Measuring this metric depends on finding a test suit with flaky tests. <ul style="list-style-type: none"> • Count number of flaky tests, and the total number of tests in the suit. • Manually fix flaky tests, counting how many are successfully fixed. • Compute percentage of fixed tests.

Mutation Score	K03	Descartes	Mutation score is computed using PIT and STAMP Descartes tools, on each of the three selected use case projects. <ul style="list-style-type: none"> • Measure baseline mutation scores. • Fix issues reported by Descartes. • Measure resulting mutation scores.
#Pseudo Tests #Partial tested tests	K03	Descartes	Count the issues reported by Descartes on each use case project.
#Execution Paths	K04	CAMP	We measure execution paths using sysdig on the network connections between the Docker components of the system.
#configuration related bugs	K05	CAMP	We have reviewed the issues reported in the last year, and made an estimate on what were configuration related.
#configurations tested	K06	CAMP	Compute number of configurations tested.
Time to deploy SUT in configuration	K06	CAMP	STAMP CAMP CI Executor reports the deployment time.
#crash replicating tests	K08	Botsing	Crash replication is done with the Botsing tool, based on stack traces from deployed TelluCloud instances. <ul style="list-style-type: none"> • Count number of stack traces. • Attempt to replicate crashes with Botsing, counting the number of replicating tests.

Table 29: Tasks for Measuring Metrics in Tellu Validation Experimentation

9. B.2 Validation Target Objects and Tasks

The tasks of the experiments are organized by STAMP tool to evaluate. We see from the table above that all KPIs measure the effect of one specific STAMP tool, with two exceptions. Code coverage is measured for multiple tools, as well as overall. Flaky tests (K02) are not addressed by any of the main STAMP tools, and needs its own task. So on the top level we have the following main tasks:

- Overall code coverage
- Descartes testing and validation
- DSpot testing and validation
- CAMP testing and validation
- Botsing testing and validation
- Flaky test experiment

The tool-specific tasks involve the treatment for measuring the relevant KPIs and running the tool, possibly in various configurations. The typical methodology is to collect baseline KPI values before using the tool, for KPIs which may be affected by the tool. We then use the tool on the use case(s), logging our experiences and committing produced artifacts to the STAMP GitHub repositories for use case results. Once tool usage is finished, new KPI values are collected and changes relative to baseline values are computed. The sub-tasks and methodology details will be described in the following sections.

Regarding target objects, code coverage and testing of Descartes and DSpot target the three selected use case projects: TelluLib, Core and FilterStore. TelluLib and Core both have substantial amounts of code and tests, and the tasks will be carried out separately on both of these. The FilterStore project is high-level and does not have much unit tests of its own. It will also be considered in each task, but some tasks may be less

relevant here. CAMP, Botsing and flaky tests target the TelluCloud system as a whole, each in their own way.

9.B.3 Validation Method

Validation method - Overall code coverage

Test code coverage has been measured at previous points for the three use case projects, and is measured right before starting the tool-specific test tasks, to get a baseline for this round of experiments. Changes in coverage will be measured as part of the tool-specific tasks when the test suit has been modified by the tool. If the test suite of a use case project is modified by more than one tool, we will merge the modifications and compute the final test coverage.

We use Clover to compute the code coverage²³.

More specifically, we use the Eclipse plugin version of OpenClover 4.3.1, running it in Eclipse. It generates an HTML report, which we store in the GitHub repositories for use case output.

Test coverage is computed from running the test suit of the Maven project. This is predominantly unit tests.

Validation method - Descartes

The Descartes mutation engine for PIT is tested on the three use case projects. The goal of using this tool is to improve the mutation score, finding pseudo and partially tested methods and improving the tests where feasible. This is tracked by KPI 03. A secondary goal is to improve the test coverage (KPI 01).

Descartes produces an *Issues* report, listing pseudo and partially tested methods - methods which are invoked by tests but not failing tests when the method bodies are removed. We count partially-tested and pseudo-tested methods. We work on each issue, looking at the code and tests to understand why the mutation doesn't fail the test(s). We try to categorize each issue according to the following categories:

- Fixed (tag FIXED): The issue uncovered a weakness in our tests. We want the method to be tested, and so we improve an existing test or make a new test to make sure all mutations are caught.
- Won't fix now (tag WONTFIX): This is a method we do not have any intention to test at the present time, typically because it is not significant and because the work involved to fully test it is not justified. We may decide to extend our tests to include such methods in the future.
- Exclude (tag EXCLUDE): For methods which are not meant to have any impact on measurable functionality, and which should ideally be left out of the mutation score. The classic example is logging methods, which we have in TelluLib. Note that we don't imply it is wrong of Descartes to flag it as an issue, just that it is a mutant we will never intend to kill because it is not business critical.

For each of the three use case projects, we have the following procedure:

1. Record baseline metrics reported by Clover: LOC, number of unit tests, coverage.
2. Calculate baseline mutation score with PIT (default Gregor engine) - record mutation score and runtime.
3. Run PIT with Descartes. Record baseline Descartes mutation score and runtime.
4. Study *Issues* report produced by Descartes. Count pseudo-tested and partially-tested methods.
5. Analyse each issue to understand why removing the method body doesn't fail the test(s) listed in the report. If method functionality should be tested, attempt to fix by adding assertions or new test. Categorize each issue as FIXED, WONTFIX or EXCLUDE. Log time spent to analyse and fix issues.
6. Count the number of issues in each category.
7. If any changes were made to tests or code, we must run Descartes again, to verify fixes and check for new issues which may have been introduced. This means repeating steps 4-7, until there are no more issues to fix. In the analysis we must check if FIXED issues really are fixed, and check for new issues.
8. Record final numbers for Descartes mutation score, pseudo- and partially-tested methods and the

²³ <http://openclover.org/index>



three categories.

9. Calculate final mutation score with PIT (default Gregor engine) - record mutation score and runtime.
10. Record new metrics reported by Clover: LOC, number of unit tests, coverage.
11. Calculate relative change in mutation score and coverage.

For these main tests we use Descartes as a Maven plugin, configured as described in the GitHub quick start and adding METHODS and ISSUES to outputFormats to have complete reporting. This means using plugin org.pitest.pitest-maven version 1.4.0 with eu.stamp-project.Descartes version 1.2.4. We have configured this in the pom file of each project, in a profile named “stamp”. We run with the following command:

mvn org.pitest:pitest-maven:mutationCoverage -Pstamp

This produces the PIT report with coverage and mutation score, and the Descartes-specific Issues report. This is convenient to integrate into Tellu’s development environment.

In addition to the main test sequence on each use case project, we also test the Descartes plugin for Eclipse. This is a functional test to check if it works and its usability, and evaluate it as a way to use Descartes in the Tellu development environment.

Validation method - DSpot

DSpot is tested on the same three use case projects as Descartes. The goal of using this tool is to improve the test coverage (KPI 01), through generation of new tests. We test DSpot as a command line tool, first building it from source code, as this is the base version. Once this works well enough to produce new test coverage, we do additional tests of the Maven and Eclipse plugins, to verify that these work the same way and to evaluate these ways of running the tool.

DSpot has various configuration parameters to vary, and much of the challenge of using the tool effectively is to find a good configuration. Parameter values are partly set in a property file and partly provided on the command line. We divide our work into a number of experiments, where each experiment vary one specific aspect. The properties file will usually be the same, not just within one experiment but also across experiments. The main variability is expressed in the command line arguments. One experiment consist of multiple runs, typically modifying the value for one specific argument. Each experiment has a sub-folder in Tellu’s folder in the “dspot-usecases-output” GitHub repository, with descriptions of the experiment and logs of each run. Here are our plan for key arguments:

- Iterations: Our intention is to keep it at the default 3 for all experiments, unless very long or short runtimes should prompt us to try other values, and we have not found reason to change it.
- Test class: We try running the tool without limiting it to specific tests, but since this takes very long, most experiments are limited to a specific test class, chosen because interesting and fixable issues were found by it in the Descartes experiment.
- Test selectors (criterion): We test the different test selectors, doing experiments where this is the variable. The most interesting ones are the JacocoCoverageSelector, which selects tests which increase test coverage, and the PitMutantScoreSelector, which selects tests which kill more mutants. We run PitMutantScoreSelector with the Descartes engine. We are especially interested to see if any of the issue fixes we did manually in the Descartes experiment can be done for us by DSpot.
- Amplifiers: DSpot has a number of amplifiers to modify tests, and these can be combined. We will do experiments testing each amplifier in turn, as well experiments combining multiple amplifiers. As it is impossible (and unnecessary) to test every possible combination, we select some different combinations based on interest and advice from the DSpot developers.
- Test timeout: We find that setting a timeout lower than the default 10 seconds is useful to avoid unnecessarily long run times. This can be set lower as long as no test should take so long to complete.

Procedure for each experiment:

1. Selecting a use case project, test class and fixed DSpot arguments, make a folder with README.md to document the experiment, with a copy of the dspot.properties.
2. Run DSpot with a specific value for the variable argument.
3. Document the run in README.md, with argument value, runtime and result.
4. Save a copy of the entire console output of the run, as well as any DSpot report produced.

5. If tests were generated, examine these and measure coverage improvement.
6. Repeat steps 2-6 for each argument variation.

Validation method - CAMP

This experiment has focused on mutation testing deployment configurations. The CAMP tool allows us to mutate the way we deploy individual components as well as how they are deployed together.

Tellucloud is currently using Docker images which are deployed into a Kubernetes cluster running on AWS. The Docker images are built by Maven on our CI server and pushed to our DockerHub account. All the images are fairly simple, based on a base jdk image, adding in a jar or war archive and launching this. Configuration of each service is mainly done using configuration files which are made available to the Docker images using volume mounts and kubernetes config maps, however some services also require certain environment variables to be set.

At the moment our system tests only 1 configuration, which mimics the way we currently deploy the application to the AWS cloud. However as we are considering deploying the application to multiple cloud providers, a short term goal would be to allow easy testing of the system using several database backends for instance. In addition we would also like to test different versions of key components (MQ server, JDK etc) as a way of moving technology choices forward.

Validation method - Botsing

The STAMP tool Botsing is a Java framework for crash reproduction. The validation is focused on the KPI K08: More crash replicating test cases. We have three subtasks within the Botsing testing task:

1. Setting up and testing that the tool works in various forms. The first goal is to get the tool running in Tellu's development environment. The version available at the start of testing requires writing a bit of code to invoke it with the right arguments. We test running this as a test with Maven, and as a test from within the Eclipse IDE. We also test the Botsing support in the STAMP Eclipse plugin, which allows inputting the arguments in a wizard and saves us from writing code to run it.
2. Collecting stack traces from running TelluCloud instances. Stack traces are processed and stored to serve as input for Botsing. This is discussed further below.
3. Running Botsing and analysing stack traces to evaluate to which extent Botsing can recreate the TelluCloud traces. If new tests are produced, we measure the change in test coverage with Clover.

Tellu does not have any explicit crash replicating test cases. It is possible to find mention of crashes or exceptions in some Jira issues, but there has not been any formal way of reporting exceptions with stack traces, and no practice of writing tests based on exceptions. The initial number of crash replicating test cases is therefore zero.

Tellu uses the so-called ELK stack for log handling (Elasticsearch, Logstash, and Kibana), currently through the cloud service Logz.io. All logs from both production and staging TelluCloud instances are processed by this system. We have set up a query for all log messages with an exception, from all micro-services. We look at all such log messages over a timespan of exactly one month. The log messages are sent to the main developer conducting the experiment, who analyses the stack traces, filtering out duplicates and traces which are mostly similar and have the same root cause. The resulting stack traces are stored in the use case output repository on GitHub, with the following file naming scheme:

`<micro-service>_<instance>_<log level>_<date>`

where <date> is the date the error was first observed.

The goals for Tellu is to find to which extent crash replicating test cases are useful to the TelluCloud case, and to which extent the Botsing tool can automate the creation of such test cases. Analysing the collected stack traces, we try to determine which are relevant for crash replicating test cases, and run these through Botsing.

Validation method - Flaky tests

The method follows from the validation treatment described in B.1:

- Count number of flaky tests, and the total number of tests in the suit.
- Manually fix flaky tests, counting how many are successfully fixed.
- Compute percentage of fixed tests.

Tellu has looked for flaky tests in our test cases, manually comparing the results from multiple runs. None of the true unit tests are flaky. We identified flaky tests in a system test suite. It has six tests, running against a test deployment of TelluCloud. As this test uses a full system deployment and depends on infrastructure and database working as expected, there has been repeated instabilities with the tests. Running the test suite multiple times, we found that three of the six tests were flaky, failing only some of the time. The experiment consists of analysing the flaky tests to understand the issues, trying to fix the issues, and verifying any fixes.

9.B.4 Validation Data Collection and Measurement Method

How to measure data and which data to record follows from the method descriptions above. As TelluCloud is closed-source code, the code itself is not posted to repositories outside Tellu control. We collect reports and logging from the tools as relevant, and put this in the usecase output repositories set up by the project on GitHub for this purpose. We also put our own test reports there.

Descartes: <https://GitHub.com/STAMP-project/Dcartes-usecases-output>

Contains reports from PIT/Dcartes and Clover. We keep an issue log here, with the FIXED/WONTFIX/EXCLUDE category of each issue.

DSpot: <https://GitHub.com/STAMP-project/dspot-usecases-output>

The Tellu main folder contains one sub-folder for each experiment, with a README.md serving as overview of the experiment and logs and output stored for each run of the experiment.

CAMP: <https://GitHub.com/STAMP-project/confampl-usecases-output>

Contains the model and generated configurations from CAMP

Botsing: <https://GitHub.com/STAMP-project/evocrash-usecases-output>

Contains all production traces collected, and tests produced by Botsing on TelluCloud traces.

9.C. Validation Results

9.C.1 KPIs Report

Descartes K03 and K01

Presents results of running Descartes on the three use case projects.

Case project TelluLib

The table below shows project metrics collected throughout the STAMP project. The last two columns are for the current test iteration - before and after running the experiment.

	20.12.2017	24.04.2018	18.09.2018 Before test	20.09.2018 After test
NCLOC	7912	8196	7638	7638
Unit tests	49	75	75	76
Coverage	34%	51.9%	53.6%	54.4%

Pitest mutation score	29%	44%	44%	43%
Pitest runtime	113 sec	197 sec	173 sec	167 sec
Descartes mutation score	35%	46%	48%	50%
Descartes runtime	46 sec	78 sec	69 sec	67 sec

Table 30: TelluLib Project Metrics

Total issues: 34

- Pseudo-tested: 30
- Partially-tested: 4

Issue resolution:

- Fixed: 15
- Won't fix now: 5
- Exclude: 14

The analysis revealed a number of interesting weaknesses in the tests, with methods we thought were covered. These were usually fixed with a simple extension of the test. One new test was written. TelluLib defines its own logging system, and so we have logger methods and corresponding back-end methods which can be removed without affecting tests. All issues categorized as EXCLUDE belong to this package, except for one method which has a purely cosmetic effect. A few methods dealing with writing to a file is considered out of testing scope for now, and tagged WONTFIX. One issue we believed we could fix, but the second iteration of Descartes revealed that the mutants were still not killed. More attempts were made to fix it, but it turned out to be unfeasible with the current approaches, as the effect of the method in question were always implemented by other parts of the code as well. This issue was changed to WONTFIX.

Time spent on issues (analysis+fix): 8 hours.

Mutation score: from 48% to 50% → 4.2% increase.

Test coverage: non-covered code reduced from 46.4% to 45.6% → 1.7% improvement.

Case project Core

See table for project metrics. The last two columns are for the current test iteration - before and after running the experiment.

	25.04.2018	18.09.2018 Before test	03.10.2018 After test
NCLOC	26905	27018	27006
Unit tests	80	84	95
Coverage	26.5%	26.5%	26.9%
Pitest mutation score	0% (Fails)	13%	21%
Pitest runtime	(21.7 sec)	24 min	37 min
Descartes mutation score	17%	14%	16%

Descartes runtime	21 min	18 min	15:30 min
-------------------	--------	--------	-----------

Table 31: Core Project Metrics

Total issues: 125

- Pseudo-tested: 110
- Partially-tested: 15

Issue resolution:

- Fixed: 19
- Won't fix now: 102
- Exclude: 4

We fixed 19 out of 125 issues. For the majority of the fixes a new test case was needed. This might seem a small proportion, but it is important to notice that (i) we fixed all the methods we have aspirations to cover with our current tests, and most of these fixes represent an important increase in test quality; (ii) we analyzed all the 125 issues and know exactly why we decided to not fix the other 106 issues, as discussed below. The main problem in this project is not any remaining issues, but rather the low test coverage. The change in test coverage was minimal (although there was some improvement), as the new tests mainly execute the same code as existing tests.

This project had a large number of issues. Most of these issues relate to classes which implement the TelluCloud data model. TelluCloud has a large data model with many classes, and this is a substantial part of the Core project. Hibernate is used to persist this data, and the classes have Hibernate annotations. The classes typically have many data fields, with getters and setters which are called by Hibernate, when data is transferred between SQL and object form. There were few unit tests for these classes, so the test coverage usually comes from higher-level tests which invokes Hibernate. Much of the issue count turned out to be get-methods called by Hibernate (tests typically use an in-memory database). We have looked at all of them in the issue analysis. Most of them simply return a value, and these have been categorized as "Won't fix". We don't consider it useful to write unit tests for every getter. Some methods do some processing to be able to return a value, and for these we have written tests. What we see as useful in the longer term, is to test the persistence and loading of objects. These are not unit tests and have not been made in this experiment, but once we have such tests Descartes will be very useful to make sure all members are properly covered by them. Some set-methods on the data model classes were also pseudo-tested. Here we also added tests when we found any logic in the method, leaving the simplest ones pseudo-tested. So while we found some useful fixes for the data model classes, the majority of issues are things we don't intend to test at this time, and this accounts for most of the around 100 unfixed issues.

Of the rest, the class with the highest number of issues deals with retrieving properties which typically come from a property file. Four issues of partially-tested methods were tagged as EXCLUDE, as some of the values returned after mutation are legal in any case (changing the mutator to return different values might remove some of these from the issue list, but the important thing for us is that the partial testing is all we need). Some methods here should indeed be fully tested, and those tests were fixed. The most important issues in the project dealt with utility methods doing transformations of data, and these were all fixed. Then there were some methods related to persistence and file output, which we did not consider in our current testing scope. A few issues were troublesome to fix, as the system has a number of layers and it is not always easy to know how to fully test some private low-level method.

Time spent on issues (analysis+fix): 14 hours.

Mutation score: from 14% to 16% → 14% increase.

Test coverage: non-covered code reduced from 73.5% to 73.1% → 0.5% improvement.

Case project FilterStore

See table for project metrics. The last two columns are for the current test iteration - before and after running the experiment.

	25.04.2018	18.09.2018 Before test	21.09.2018 After test
NCLOC	4572	4662	4662
Unit tests	3	4	4
Coverage	35.8%	35.8%	35.8%
Pitest mutation score	40%	38%	38%
Pitest runtime	28 min	20:30 min	20:30 min
Descartes mutation score	52%	54%	54%
Descartes runtime	7:30 min	7:21 min	7:21 min

Table 32: FilterStore Project Metrics

Total issues: 4

- Pseudo-tested: 2
- Partially-tested: 2

Issue resolution:

- Won't fix now: 4

This project represents the highest level in the TelluCloud system architecture, implementing a micro-service. The high-level classes depend on the Core and TelluLib projects. There are also some low-level service-specific classes, but there are very few unit tests here. Almost all test coverage is provided by a single test, which is a component test rather than a true unit test. It runs with the unit tests, with everything running in a single Java VM, including in-memory database, and does assertions on message handling. We do not expect these assertions to kill all mutants. In addition to message output, the service makes changes to the database, and this is so far not verified by tests. Exceptions and other errors are typically caught and logged, and so does not propagate to the surface of the component test unless it affects message output.

Mutation testing is less relevant here than in the other projects, but it still provides useful insight, showing strengths and weaknesses of the component test. Only four issues were found, and two of these are partially tested. We were surprised by the low number, showing that even with the limited testing approach, the lines which are covered are actually covered properly. We analysed the four issues. None of them can be fixed within the current test approach. One could be caught by detailed validation of database changes and one with monitoring of the error log. Two actually deal with running a timer, and would need a much longer-running test to validate. So all issues were tagged WONTFIX. They are very useful as input to what types of testing we may want to do in the future.

Time spent on issues (analysis+fix): 1 hour.

DSpot K01

The DSpot experiments did not manage to produce any new tests for our use case projects. Experiments are documented in the GitHub use case output repository. The issues are described in the qualitative evaluation (C.2).

Test coverage: no change

CAMP K04, K05 and K06

We have created a Docker-compose configuration and a model of our system to be used by CAMP to generate different configurations. As we are only changing out interchangeable parts (Message Queue system, database server), we did not expect that our code would behave any different for the different configurations. If in the future we create multiple versions of our own services we expect this number to increase.

Execution paths (K04): No change

We found 10 issues discovered previously, which is attributed to specific configurations, based on review of issues reported into Jira. While testing the new configurations with CAMP found 2 issues that had to be fixed in order to run the system in the new configuration.

Existing configuration-related bugs: 10

New configuration-related bugs discovered (K05): 2

Percentage improvement in configuration-related bugs discovered (K05): 20%

CAMP has matured a lot in recent weeks, and is at the moment a fairly easy system to set up. It is now a lot easier to run the system tests as a developer, as there is only one command to get the system up in a given configuration (typically ~5 minutes as opposed to 3-4 hours to set up all components manually).

Number of new configurations tested (K06): 2

Baseline time to deploy TelluCloud (K06): 4 hours

Time to deploy software using CAMP (K06): 10 minutes

Botsing K08 and K01

We have monitored the stack traces logged by operational TelluCloud instances for one month. Log messages with stack traces have usually been on an order of magnitude of tens per day, but sometimes going into the hundreds per day. Most are recurring issues, and there proved to be a low number of distinct traces. Filtering out duplicate messages, we were left with **20 stack traces**. These are stored in the evocrash-usecases-output repository on GitHub, under Tellu/production_traces.

None of the exceptions represent a crash of a service. All exceptions are caught and logged by Tellu code, often on the warning level. So the code is more or less designed to handle all these exceptions. Another common factor is that the cause of the exception always has an external factor, either in input to the micro-service or in its connection to another component. The main types of exceptions follows from the nature of these micro-services - they process incoming messages, and most of them do database transactions as part of their processing. We have found the following types of errors:

- Database connection: A common root cause for a number of different exceptions was loss of connection with the database service.
- Invalid database data: We have seen duplicate database key issues.
- Database deadlock: A conflict between two concurrent database transactions.
- Missing file: Indicates an error with the deployment, where a needed file is not available to the service.
- Illegal input data: An exception is thrown when processing a message which is not according to specification. This is an intended behaviour, but can in some cases indicate a mismatch in a protocol between two micro-services.
- Too many requests: The edges which receive data from external sources limit the rate of requests from any single sender, and throw an exception to stop the message processing which exceed the limit. This is an intended behaviour.
- Missing service: Failure to send a message to a micro-service, because no instance of the service was running.
- Queue error: Messaging between micro-services are handled by queues. We have seen an error with the connection to the queue.

- Jetty web-server: Most micro-services run servlets with Jetty, and we see some exceptions within the Jetty framework.

None of the exceptions reveal errors with the TelluCloud code, and creating unit tests to recreate them is therefore of little value. They are mainly interesting from a system test perspective, replicating external conditions. We have no replicated tests, and therefore no impact on test coverage. We have further tested replicating an introduced error in the code, and discuss Botsing relevance for the TelluCloud use case in the qualitative evaluation below.

Flaky tests - K02

As described in the methodology, Tellu only have flaky tests in a system test suite with the following baseline metrics:

- Total tests: 6
- Flakey tests: 3

The flaky tests were analysed. The tests depend on messages sent to the system triggering specific rules. Closer inspection revealed that an additional rule was sometimes triggered, as an unforeseen consequence of message content. This flakiness was fixed by changing the message content. This gives the following result metrics:

- Number of flaky tests fixed during the period: 3
- Percentage of flaky tests fixed, compared to number of flaky tests: 100%
- Percentage of flaky tests fixed, compared to the total number of tests: 50%

The experiment was a successful exercise in that we were able to fix the flaky tests found, but the result is of little interest to the project, both because of the small numbers involved and since it was a fully manual process not involving tool support.

9.C.2 Qualitative Evaluation and Recommendations

Evaluation and recommendations are given for each tool.

Descartes

We have tested Descartes on our three use case projects. The tool worked well and did what was expected of it - to compute mutation scores and reveal methods which were not properly tested despite being covered by the test executions. The issue report it produces gives a good overview of the issues. The information is useful, and can be used to improve test quality. An important benefit of using Descartes is increased insight in the code. It has showed that our tests are not always testing all they were supposed to test. Fixing bad tests is very important, and fixing one issue can be vital, even if the change in mutation score is small. The distinction between partially-tested and pseudo-tested is usually not important in our cases, although there are special cases where partially-tested is good enough.

We see that far from all issues are fixable or relevant. Only manual analysis can make the distinction. In our experiments we used a lot of manual effort to analyse and fix tests. For each issue it is necessary to understand the workings of both the pseudo/partially tested method and the test. This typically involves implementation details, which can be difficult to follow if the tester did not write the code himself or if the code is old. At Tellu we believe the best way to use Descartes is when the test is made. After writing a test, the developer should check that it covers the intended code, and run Descartes to make sure it is covered properly, fixing any pseudo/partial testing. It is much harder to apply it retroactively to old tests. So at Tellu we will use Descartes as part of our development process, as a means to ensure test quality.

It would be useful to have tool support for comparing two coverage reports. When running Descartes again after fixing issues, there is a need to identify fixed, remaining and new issues compared to the last run, and doing this manually in the experiments took some effort and was error-prone. In Tellu's case it would also be useful to have tool support for tagging the issues according to our categories, and hide those we don't intend to fix yet.

We also tested the Eclipse plugin. We found some issues which needed sorting out by the developer. Once the issues had been worked out, it worked well. It makes it easy to run Descartes without any pre-existing knowledge. No configuration is needed unless you want to change the default set of mutators, and the PIT

report is automatically displayed in Eclipse. It is a good option for those who do not want Descartes in their regular Maven setup. At Tellu we want to use it with Maven, and the Eclipse plugin does not add anything for us.

DSpot

We encountered problems building and running current versions of DSpot on the developer machines used in the validation. DSpot is developed and primarily tested on Linux, while Tellu uses Windows 10 in the validation, as that is relevant in the Tellu case. DSpot runs Maven builds, so it depends on Maven and all its dependencies working as intended. We have not had any other trouble with the Maven build system on the test machines. DSpot does not build on the main developer machine used for STAMP, or on a secondary Windows 10 machine available to the tester in the office. This issue has not been resolved, despite many attempts in cooperation with DSpot developers. Running DSpot on these machines has also been unsuccessful, although there is no apparent link between the two problems. DSpot launches, but the process eventually stops with Maven not being able to compile Java sources, something which otherwise works fine on these machines. Again none of the attempts at fixing the issue has been successful, and the fact that DSpot also does not build on these machines makes it difficult to test possible changes to the DSpot source code.

We discovered that DSpot both builds and runs on a third tested Windows 10 system. This is the home computer of the Tellu developer testing DSpot. No significant difference between this and the other two Windows 10 systems have so far been found, related to Maven and Java, and we have focused our remaining efforts on running experiments. Investigation into the compatibility issues will be resumed after the experiments are done. The documented experiments have all been run on this home computer. Experiment details are found in the GitHub use case output repository, and issues with the tool are posted on the DSpot repository. We were not able to produce any test improvements. Here we summarise the main issues:

- The JacocoCoverageSelector does not work on the test system. Fixes have been attempted, but so far a successful solution has not been found.
- It is not possible to combine multiple amplifiers when running with the Windows command line. This means the intended experiments combining amplifiers could not be carried out.
- Many test runs ended with an IllegalArgumentException "character to be escaped is missing". This is probably also a Windows compatibility issue. Attempts to fix it has so far not been successful.

With the PitMutantScoreSelector using Descartes, DSpot was able to run to completion with some of the amplifiers, but these runs did not yield any new tests. To summarise, the experiments uncovered a number of Windows compatibility issues which is being worked on but not yet resolved. These issues meant that not all of the intended test plan could be followed.

CAMP

CAMP was initially very complicated to set up, but has matured considerably over the last months, and while the most recent version use completely different configuration files, setting these up is a lot simpler and less time consuming than it used to be. We are now able to get it to generate only valid configurations, and to vary those based on the model and the base Docker-compose file. The internal TelluCloud micro-services are not really interchangeable, so at the moment we are left with varying the infrastructure services, such as database, message queue etc.

While we did not uncover many issues by running the system with the new configurations, it is an interesting way to validate that the code is able to run using multiple versions of infrastructure services, and, if run as part of CI, to ensure that we do not have regressions related to specific versions of infrastructure services.

Botsing

Firstly, we were able to get Botsing running in Tellu's development environment. It was first run on a demonstration case from the EvoCrash-demo repository, which we have previously run with EvoCrash. It successfully replicated the crash of the XWIKI_13031 case, which we have previously replicated with

EvoCrash (reported in D5.5). Botsing successfully runs, invoked as a test running either from Maven or Eclipse.

As shown in the KPIs report, we collected TelluCloud stack traces, but none of these are relevant for replication. To still investigate use of Botsing on the TelluCloud case, we introduced an error in the code in the filterstore project, causing a `NullPointerException`, to see if this could be replicated with Botsing. It turned out Botsing was unable to replicate this crash, from the third or the second frame. It either stops with an exception, or it completes without being able to replicate the crash. We looked at the stack trace it was trying to reproduce to understand why. It is a typical stack trace for a TelluCloud micro-service, with the service trying to send an output as it is processing an incoming message. The problem is that it requires the incoming data to have an ID which matches a row in the test database. There is no chance that random mocking of input data is going to produce this match, and a genetic algorithm is of no help. This further strengthens the point identified in the earlier stack trace analysis - the dependence of the external system context and incoming messages as a key aspect of testing TelluCloud micro-services. Input data in the form of incoming messages and database rows are integral to the testing and need to match. The code itself, taken in isolation, is mostly simple and well-proven. The important challenges in the TelluCloud case lays in system testing. This does not mean that it won't be possible to reproduce any TelluCloud stack traces. We were able to reproduce one with EvoCrash in the previous test phase. But it means we will very seldom see such a stack trace logged by a running instance, and Botsing is of low usefulness in the TelluCloud case.

9.C.3 Answers to Validation Questions

We summarise our evaluation in reference to the validation questions VQ1-VQ4.

- **VQ1 - Can STAMP technologies assist software developers to reach areas of code that are not tested?** The KPIs indicate we have not yet had much success in this respect. Windows compatibility issues with DSpot means this tool has not yet contributed. Fixing issues raised by Descartes only contribute small improvements as a secondary effect of improving mutation score. We did not find useful stack traces for Botsing to reproduce. And for CAMP, as expected, we did not increase the code covered by tests, as the system behaves the same independent of which infrastructure services we use.
- **VQ2 - Can STAMP tools increase the level of confidence about test cases?** Descartes has proved very useful in both efficiently computing mutation scores and in finding pseudo-tested methods. KPI wise, the increase in mutation score (K03) is not very big, but we have found and corrected 34 cases of pseudo-tested methods in two of the use case projects, as well as understanding many more such issues, with the result being that we now know what is tested. Tellu sees Descartes as a useful addition to the DevOps toolbox, and will continue to use the tool. Flaky tests (K02) are not a big problem for Tellu, and the issues found have been fixed in the experiment.
- **VQ3 - Can STAMP tools increase developers confidence in running the SUT under various environments?** Yes, it is now fairly easy to bring up working version of TelluCloud using various different configuration, and to run system tests on these configurations.
- **VQ4 - Can STAMP tools speed up the test development process?** DSpot and Botsing are the tools which can generate test cases, but neither has produced test cases for the TelluCloud use case in this phase of validation. Generating a new set of configurations to run system tests are now quite easy thanks to CAMP. The main effort in this would be to create a working base configuration for the new component/service. However, as Tellucloud is using kubernetes for deployment, keeping the Docker-compose template and real deployment configuration in sync will be an extra effort.

9.C.4 Next steps for validation

Tellu will continue to test all the tools as new versions are released. The current focus is on DSpot, where we want to continue assisting on fixing Windows compatibility. We hope to be able to test all capabilities of the tool.

For CAMP we would like to integrate it within our CI process, and also increase the number of components we are varying. In addition it might be interesting to increase the test cases, and using the parameter variations in the dockerfile, try to find different edge cases.

For Botsing we plan to do an evaluation where we plant bugs at various places in the code, triggering exceptions and seeing which of these can be recreated with Botsing. We already know that Botsing in its current form is of little relevance for the TelluCloud use case, but we hope that collecting more data on where it works and not will be valuable input for the developers and also in how code complexity affects the feasibility of tools such as Botsing.

Tellu will continue to use Descartes as a developer tool when writing new tests.

10. XWiki Use Case Validation

XWiki Use Case Highlights

- Descartes in production and failing XWiki's build when mutation score decreases on modules (executed by CI)
- Developed Docker images, deployments, builds and setups for 10 different supported configurations under which to execute XWiki functional tests (and executed over 26 configurations)
- Configuration testing in place and committed in the XWiki project, with CI jobs developed and now testing the 10 configurations automatically every day.
- Regular global test coverage increase on XWiki even though it was already at a relatively high value when we started on STAMP (65%). Not easy to do on a large code base.

10.A. Use Case Description

The content below has been derived from D5.5 in order to make this section more readable. It contains updates to the use case in order to explain it better.

10.A.1 Target Software

XWiki is an advanced Enterprise open source wiki tool written in Java but it's also positioned as a generic web-development platform a [generic web-development platform](#) that can be used to develop any kind of web applications, especially if they relate to content. Its [features are numerous](#) and this makes it a large and complex platform.

XWiki currently boasts more than [700 extensions](#) providing additional features (Blog, Meeting Manager, LDAP integration, etc).

The whole codebase size varies between 6M lines of code as reported by OpenHub (see figure 7 and the [XWiki OpenHub page](#) for more metrics) and for the whole platform with lots of extensions included, down to between 300K ([SonarQube](#)) or 650K ([Clover](#)) for the Core product that is called XWiki Standard. For the STAMP project we've decided to focus on XWiki Standard.

Code

Lines of Code

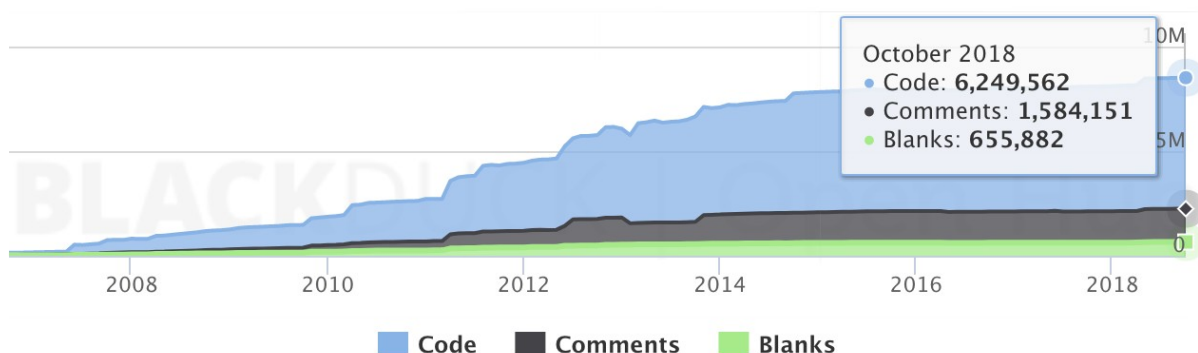


Figure 7: Some Metrics for XWiki

10.A.2 Experimentation Environment

The code for XWiki Standard is spread across 3 repositories in [GitHub](#):

- `xwiki-commons`: common modules that can be used outside of XWiki
- `xwiki-rendering`: the XWiki Rendering engine (i.e. libraries used to convert an input in a given syntax into another target syntax, e.g. from XWiki Markup to HTML)
- `xwiki-platform`: all the wiki features

This means that we're applying the tools developed by STAMP on the full XWiki Standard and that the KPI/metrics are measured on the 3 code repositories mentioned. We could have taken a smaller subset but we decided to take the whole product code base in order to have both a significant code base but even more importantly to be using real production code from various domains: Database access, Web UI with Javascript/CSS/HTML, etc. We wanted to make sure that the STAMP-developed tools could work for those various use cases/needs. It's also important to note that XWiki is in the process of migrating all its JUnit4 tests to JUnit5 and thus it's important that the STAMP-developed tools work on JUnit5.

In addition, we also wanted to be able to apply STAMP on XWiki for a real usage and be able to commit the results, by having the XWiki development community vote on using them and integrate them in the XWiki development process.

10.A.3 Expected Improvements

The following use cases are of interest to the XWiki project:

- **Improving Flaky Test handling**: Several times per week, [XWiki's Continuous Integration server](#) (CI) sends false positives to the XWiki open source project's notifications mailing list. This list is used by XWiki developers working on the project. When they receive such emails, they usually stop what they're doing to analyze the problem, spending several minutes or hours to finally discover that the problem is actually either a build environment problem or a flickering test. It decreases the value given to the CI, and, over time, developers stop caring about build failures coming from the CI till the day of the release and then all integration problems need to be fixed, this delaying the release a lot, and negating the exact purpose of a CI tool which is to discover integration problems as they

happen.

- **Improving test coverage and test accuracy:** XWiki already has a relatively high level of test coverage (around 65%+ at the start of the STAMP project, with 9K+ tests). Test coverage is not an absolute measure of the quality and getting to 100% is not the goal but increasing it to around 80% would help a lot. Especially in parts of the code where this coverage is low currently. Now test coverage doesn't guarantee the quality of the test (a test with no asserts will generate coverage but won't prove much). Thus it's also important to evaluate the quality of the tests. This is where mutation testing can help out. In addition it would be awesome to have tests automatically generated and that would increase automatically the test coverage/accuracy.
- **Configuration testing:** XWiki is web application deployed as a WAR which can be installed on any Servlet Container and running on any Database. It can also run in all browsers. Thus it's very important that the XWiki tests are executed in various environments to prove that the [subset of supported environments](#) work, and that no regression are brought to any of those environments when new code is added.
- **Reduce time to debug runtime issues:** When XWiki users have a problem in production, they report it through the [XWiki issue tracker](#). They are asked to include the stack trace in the issue report. It would be nice and interesting to make it simpler to reproduce the issue by having some tool generate automatically regression tests. Botsing is a first step in this direction as it's currently able to generate tests, which, when executed generate the stack trace. Thus the XWiki project is keen to test it and is looking forward to Botsing (or some other tool) being able to go step further and generate regression tests.

10.A.4 Business Relevance

XWiki SAS is an open source software vendor closely defined by its values and the relations it maintains with the open source community around the XWiki project. STAMP can help XWiki's business in different ways. For example, every day the XWiki community submits bugs and requirements. Bugs suggest we still need more and improved tests and requirements that we need an efficient developing process. STAMP fits in XWiki's production organisation. STAMP can help improve developers' productivity by reducing the time spent on false positives or the time spent to reproduce bugs related to a given configuration or to reproduce a production crash.

STAMP can be leveraged in the relationship with the community. We want to apply STAMP on XWiki for a real usage and be able to commit the results, by having the XWiki development community vote on using them and integrate them in the XWiki development process.

Last but not least, improvements in configuration testing can make XWiki's delivery process simpler and more efficient by enabling the packaging and distribution of XWiki as official Docker images.

10.B. Validation Experimental Method

10.B.1 Validation Treatments

The following table contains XWiki's control (a.k.a baseline) and treatment tasks for all KPIs and metrics. As a general rule and when not mentioned, XWiki is following the strategies defined in D5.3.

KPI	Metric	Control and Treatment
K01	Global coverage	<p>Control: XWiki is using the Global Coverage, as measured with Clover, from 2016-12-20, i.e. 65.29% and comparing the new values with it.</p> <p>Treatment: Use Clover and a specific XWiki-written Jenkins pipeline job to measure the global coverage.</p>

	Coverage from Descartes	<p>Control: The following points were taken into consideration:</p> <ul style="list-style-type: none"> The coverage contributed by each tool was not measured at the start of STAMP and it was decided to measure it only when D5.3 was released. In addition it's very hard to measure the coverage contribution of a single fix suggested by Descartes to the overall coverage. Several times per day, we have code and tests being changed and changing the global coverage so we would need to recompute the full global coverage after each fix suggested by Descartes. Since that would take over 6 hours for each fix (that's the time for the XWiki CI job to measure the global coverage), we decided it was not realistic and we've decided to measure Descartes contribution module per module and sum up the individual coverage percentage added for each module as a measure of the contribution. <p>Thus the baseline will be per module and will correspond to the value of coverage, measured with Jacoco, before applying the fixes suggested by Descartes. Note that we use Jacoco since it's already integrated in XWiki's build (with a strategy of failing the build when the coverage goes down).</p> <p>Treatment: Measure the local coverage with Jacoco at the module level.</p>
	Coverage from DSpot	<p>Control + treatment: Same as coverage from Descartes above. Execute DSpot from the command line (future: integrate the Maven plugin in XWiki's build).</p>
	Coverage from CAMP	<p>Control: Similar to the coverage from Descartes above, the coverage contributed by each tool was not measured at the start of STAMP so there no older baseline value to compare with. The idea is to compute the global coverage of XWiki, using Clover, and without executing the Docker-based functional tests (since prior to STAMP we didn't have such tests and they were introduced for K04, K05 & K06) and use the value as the baseline.</p> <p>Treatment: Measure the global coverage including Docker-based tests, using Clover and XWiki's Jenkins CI job.</p>
	Coverage from Evocrash/Botsing	<p>Control + treatment: N/A because at this stage Botsing generates tests to reproduce crashes but not regression tests. Thus currently Botsing doesn't contribute to the coverage. In the future, with K09, Botsing should generate regression tests and when this happens we would be able to measure its contribution to K01.</p>
K02	Number of flaky tests identified and handled	<p>Control: Number of flaky tests recorded in JIRA (using the custom "flickering" label in JIRA) at the beginning of STAMP, on 2016-12-31. JQL query: "Flickering Test" is not empty and created < 2016-12-31</p> <p>Treatment: Number of flaky tests record in JIRA at moment of measure, using the same JQL query as for the control but adjusting the date to be the one at the measure date. For example: "Flickering Test" is not empty and created < 2018-10-02</p>

	Number of flaky tests fixed	<p>Control: Number of flaky tests fixed in JIRA (using the custom “flickering” label in JIRA) at the beginning of STAMP, on 2016-12-31. JQL query: "Flickering Test" is not empty and created < 2016-12-31 and status = Closed</p> <p>Treatment: Number of flaky tests fixed in JIRA at moment of measure, using the same JQL query as for the control but adjusting the date to be the one at the measure date. For example: "Flickering Test" is not empty and created < 2018-10-02 and status = Closed</p>
K03	Mutation score thanks to Descartes	<p>Control: This metric was introduced at the time of D5.3 and is new and thus there's no older baseline. In addition we have exactly the same issue as with coverage above to get a global value. Thus the baseline will be measured module per module before applying any Descartes-suggested fixes. Measure using Descartes's Maven plugin integrated in XWiki's build and use the mutation score reported by Descartes.</p> <p>Treatment: Measure the Descartes mutation score after fixing the issues suggested by Descartes, module per module. Add up the score improvement from the base lines from each module.</p>
	Mutation score thanks to DSpot	<p>Control + treatment: Same as mutation score from Descartes above. Execute DSpot from the command line (future: integrate the Maven plugin in XWiki's build).</p>
K04	More unique invocation traces (microservices) or coverage increase (monolithic)	<p>Control: This is new metric from D5.3 (used to be measured only for microservices prior to D5.3). The baseline will be the same one as the one measured for K01/"Coverage from CAMP" (see above).</p> <p>Treatment: Compute the increased global coverage thanks to the Docker-based functional tests as described in K01/"Coverage from CAMP" above.</p>

K05	Number of new configuration-related bugs discovered	<p>Control: We can't compare with the total number of configuration-related issues since the beginning of XWiki's history (14 years ago). Thus we decided to semi-manually count the number of configuration-related bugs entered in XWiki's issue tracker (JIRA) from 2017-01-01 to 2018-06-31 (approx. 1.5 years) to get a baseline value of how many bugs are normally discovered by the XWiki community in 1.5 years of time. Since we started being able to measure configuration-related bugs with STAMP tools only 1.5 years into the project and since STAMP last 3 years, this means we'll also have 1.5 years of measure and thus we'll be able to compare how many new bugs are discovered thanks to STAMP tools. The following JIRA JQL query is used: <code>labels in (chrome, ie, firefox, mysql, Oracle, PostgreSQL, mssql, WebSphere, ie11, ie10, ie8, ie9, edge, java9, java10, java11, hsqldb) and category = 10000 and createdAt > 2016-12-31 and createdAt < 2018-08-31 and type = bug</code>. Value is 56.</p> <p>Treatment: Starting from 1.5 years into the project, we introduced a custom JIRA field named "ConfigurationTesting" in XWiki's JIRA so that whenever a new bug is found thanks to STAMP it's labelled. Thus the number of new bugs discovered can be found using the following JIRA query (example measure from 2018-10-22): <code>labels = ConfigurationTesting and createdAt < 2018-10-23</code>. Bugs will be found by executing XWiki Docker-based functional tests on various configurations either manually or inside XWiki's build/CI.</p>
K06	Number of new configurations tested	<p>Control: Prior to STAMP we had only 1 configuration being tested automatically. Namely it was XWiki running on HSQLDB, inside a custom Jetty packaging, on Java 8.</p> <p>Treatment: Implement a testing framework based on CAMP/TestContainers and convert XWiki's functional tests to use it. Define supported configurations and add them to XWiki's CI in pipeline jobs. Then count the number of tested configurations (either manually or in the CI).</p>
	Time to deploy software vs time before STAMP	<p>Control: Measure the average time for someone to install XWiki based on the manual install in Servlet container + Database before, as described in XWiki's documentation, using the WAR installation method. This was manually measured to be an average of 2 hours.</p> <p>Treatment: Build Docker images of XWiki and publish them as official XWiki Docker images on Dockerhub. Then use them to manually measure the time to deploy XWiki with them by following the documentation.</p>

K08	% of crash replicating test cases	<p>Control: Measure the number of production crash issues reported and fixed since the beginning of STAMP, without Evocrash/Botsing, and which have test cases for them. This is retrieved from XWiki's JIRA with the following JQL (at the date of measure, here follows an example measured o 2018-10-05): Tests in (Integration,Unit) AND description ~ ".java:" AND description ~ 'Exception' AND category = 10000 AND createdAt >= 2016-12-01 and createdAt <= 2018-10-05 and resolution not in ("Cannot Reproduce", Duplicate, Inactive, Incomplete, Invalid, "Won't Fix") ORDER BY createdAt DESC. Then manually remove false positives (i.e. issues not related to production crashes). Note that we can have this JQL query because we have a custom field in XWiki's JIRA to indicate if an issue has an automated tests associated with it, using the "Tests" custom field.</p> <p>Treatment: Run Evocrash/Botsing on issues in XWiki's JIRA having stacktraces and when Evocrash can reproduce the crash, add a custom label named "evocrash" in XWiki's JIRA. Then count the number of issues with label wit the following JIRA JQL (example of measure o 2018-10-0): labels = evocrash and createdAt <= 2018-10-05 and createdAt >= 2016-12-01</p>
-----	-----------------------------------	--

Table 33: Controls and Treatments for KPIs for XWiki Software

Important note: The XWiki open source project does not aim at supporting any possible configuration (there is a fixed list) and thus mutating configurations automatically with CAMP is not a valid use case for the XWiki project. Instead the idea is to loop through all the supported configuration and execute functional tests on them, and measure the metrics for KPIs K04, K05 and K06.

10.B.2 Validation Target Objects and Tasks

The following table lists the XWiki software systems/components that receive the validation control (baseline) and treatment (STAMP).

KPI	Metric	Target objects
K01	Global coverage	Full XWiki Standard (xwiki-commons, xwiki-rendering and xwiki-platform GitHub repository sources).
	Coverage from Descartes	A subset of XWiki modules from XWiki Standard where Descartes will have raised some "pseudo tested" and "partially tested" issues, and for modules where Descartes is succeeding in finding tests (there are cases where it doesn't recognize tests). Those modules will have their tests manually fixed to increased the Descartes mutation score (and incidentally potentially increase the coverage too).
	Coverage from DSpot	A subset of XWiki modules from XWiki Standard where DSpot will have successfully generated either new assertions or new tests (there are cases where DSpot fails to identify tests or generate new assertions or tests). Depending on the selector used, the new assertions/tests will increase coverage, mutation score or both.
	Coverage from	A subset of XWiki modules from XWiki Standard that contain functional tests

	CAMP	(i.e. inside xwiki-platform) and that will have been ported/converted to the new Docker-based testing framework (see below).
	Coverage from Botsing	No target defined for this period, see table in section B.1 for the explanation.
K02	(All metrics)	Full XWiki Standard (xwiki-commons, xwiki-rendering and xwiki-platform GitHub repository sources).
K03	(All metrics)	Same targets than for K01 metrics above.
K04	(All metrics)	Same target than for K01/Coverage from CAMP above.
K05	(All metrics)	Same target than for K01/Coverage from CAMP above.
K06	(All metrics)	Same target than for K01/Coverage from CAMP above.
K08	(All metrics)	Full XWiki Standard (xwiki-commons, xwiki-rendering and xwiki-platform GitHub repository sources). In practice it depends on existing XWiki JIRA issues having stacktrace and the ability of Evocrash/Botsing to replicate the stack traces.

Table 34: Validation Target Objects for XWiki Software

The following table lists validation tasks (experiments) that we conducted to be able to apply both controls and treatments.

KPI	Preparation tasks implemented
All	<ul style="list-style-type: none"> • Discussed in the XWiki open source community about having developers work regularly on test quality by using the various STAMP tools. Defined a TFD (Test Fixing Day), every Tuesday, where volunteers work in improving tests. • As a prerequisite for STAMP the XWiki project has converted all its CI jobs into a reusable Jenkins pipeline job, which was a huge undertaking. This enabled us to implement custom logic in the jobs to include strategies for all KPIs (flickers, Descartes execution, clover coverage computations, etc).
K01	<ul style="list-style-type: none"> • Created a Jenkins pipeline script to be able to compute the full coverage of XWiki Standard, across several GitHub repositories and thus across several Maven reactor builds, using Clover. • Defined and voted (in the XWiki open source community) a new global coverage strategy in XWiki (to ensure global coverage doesn't go down). In summary this strategy says that if the contribution of some module to the global coverage goes down, then those modules should fail the job. <ul style="list-style-type: none"> ◦ Initial proposal ◦ Revised proposal • Implemented this strategy in the Jenkins pipeline script. <ul style="list-style-type: none"> ◦ Still in progress for implementing the strategy fully ◦ See figure XXX below to see how the generated report looks like
K02	<ul style="list-style-type: none"> • Discussed and voted (in the XWiki open source community) to apply the strategy to recognise & handle flickers • Added new custom field in XWiki's JIRA (https://jira.xwiki.org) to recognise issues related

	<p>to flickers and to record flickering tests.</p> <ul style="list-style-type: none"> Implemented (and later improved) a Jenkins pipeline script to automatically flag flickers in the Jenkins reports and prevent false positive emails when there are only flickers.
K03	<p>For Descartes:</p> <ul style="list-style-type: none"> Voted and implemented (in the XWiki open source community) a strategy to fail the build if the Descartes mutation score goes down. Implemented as part of the XWiki build on all repos (commons, rendering, platform). It'd automatically executed and enforced in 3 Jenkins pipeline jobs preventing software release if failing. <p>For DSpot:</p> <ul style="list-style-type: none"> Discussed (in the XWiki open source community) and implemented modifications of XWiki's build to be able to commit DSpot-generated tests in XWiki sources (in src/test/dspot, next to src/test/java to not mix the two), and have Maven run them (using the Build Helper plugin)
K04/ K05/ K06	<ul style="list-style-type: none"> Created Docker images for XWiki and deployed them as official DockerHub images Developed and tested various approaches for integrating multiple configuration testing into XWiki's development process: <ul style="list-style-type: none"> Docker on CI experiments Docker in maven using Fabric8 POC developed Docker inside Java test classes, using Selenium Jupiter POC developed and working Docker inside Java test classes, using CAMP/TestContainers developed and working version. Selected the CAMP/TestContainers-based approach and created a testing framework for XWiki, based on it (see the documentation for using it), with automatic deployment and configuration. This allows to debug XWiki running in any configuration, right from inside your IDE. <ul style="list-style-type: none"> Takes about 3 minutes to build an XWiki WAR, deploy it and start all containers, on any configuration. See Figure XXX below to see the architecture for this testing framework that integrates with JUnit5 This testing framework is executor for configurations for CAMP. CAMP has 2 parts: the mutation part which produces Jenkinsfile files or Compose files and the execution part. What the XWiki experiment is providing is a new way to execute CAMP by running it inside functional Java tests from inside your IDE (or from your build or from your CI). This work was developed as part of WP2. Discussed and voted (in the XWiki open source community) the supported configurations for XWiki for databases, Servlet engines and Browsers, and their versions. Started implementing testing for the defined configurations in the Docker-based testing framework, see the table XXX below to see which configurations are currently supported. Discussed, voted and implemented several Jenkins pipeline jobs were created to automatically execute XWiki functional tests on various configurations <ul style="list-style-type: none"> A job to execute the latest version of supported configurations on all the Docker-based functional tests. Another job to execute all the supported configuration but on smoke tests only (i.e. a subset of of all Docker-based functional tests) A last job to execute currently unsupported configurations (and possibly not working) but that we may want to support in the near future. Note that in order to be able to execute Docker-based tests on XWiki's CI (Jenkins) running at https://ci.xwiki.org we had to introduce a new Jenkins agent using a completely different architecture (since it wasn't possible to execute Docker inside agents running on the vserver technology). This will need to be scaled up in the future.

K08	<ul style="list-style-type: none"> • Discussed with the XWiki community the opportunity to use EvoCrash/Botsing on XWiki. • Agreed on a new custom label named “evocrash” inside https://jira.xwiki.org to recognize issues with stack traces that can be reproduced by EvoCrash/Botsing
-----	---

Table 35: Validation Tasks for XWiki Software

The following figure shows a report sent by mail and generated by XWiki’s Clover pipeline job, highlighting the contribution of each XWiki module to the global coverage.

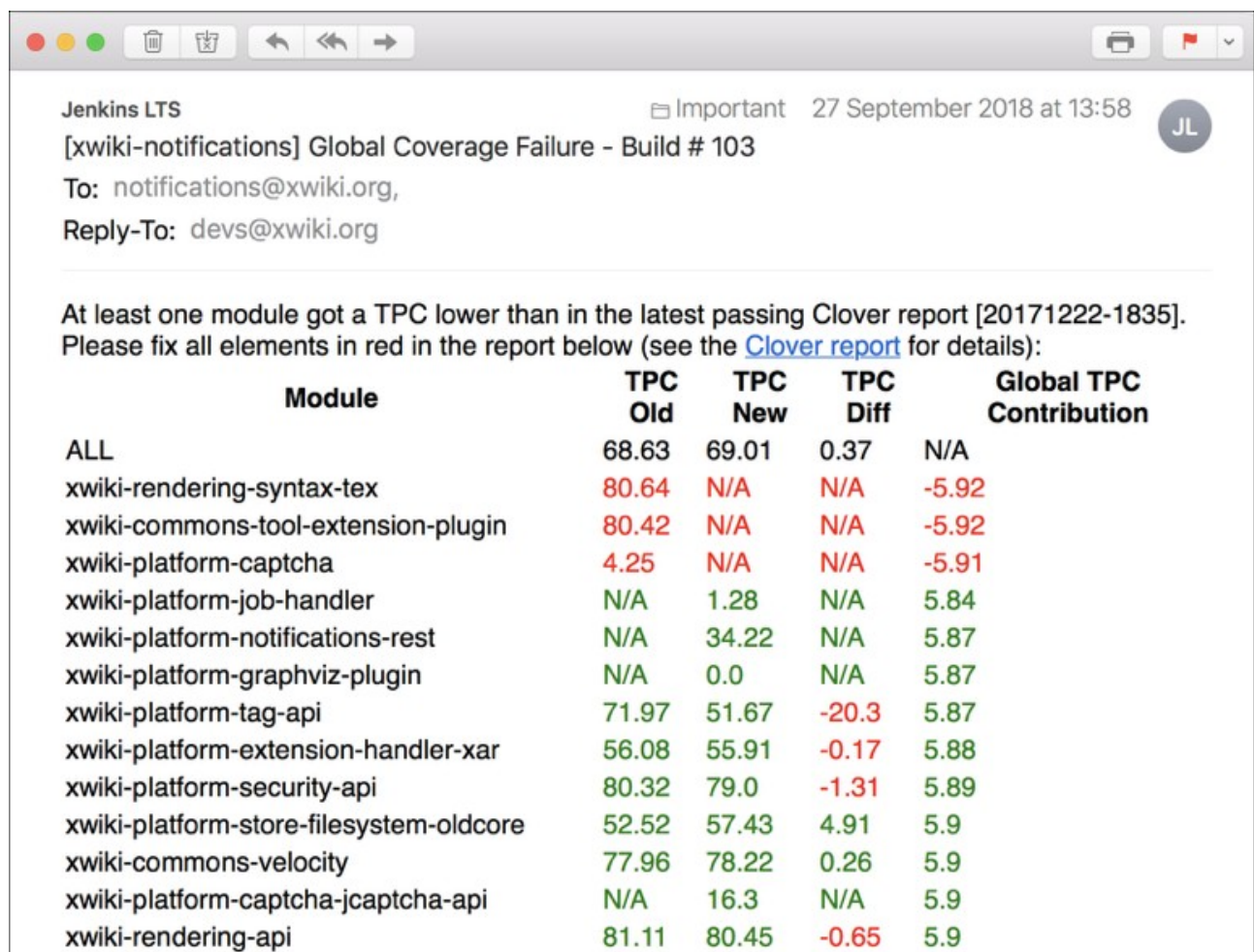


Figure 8: Global Coverage Contributions Module by Module for XWiki

Architecture for the CAMP/TestContainer-based testing framework developed by XWiki for testing various configurations inside JUnit5-based tests.

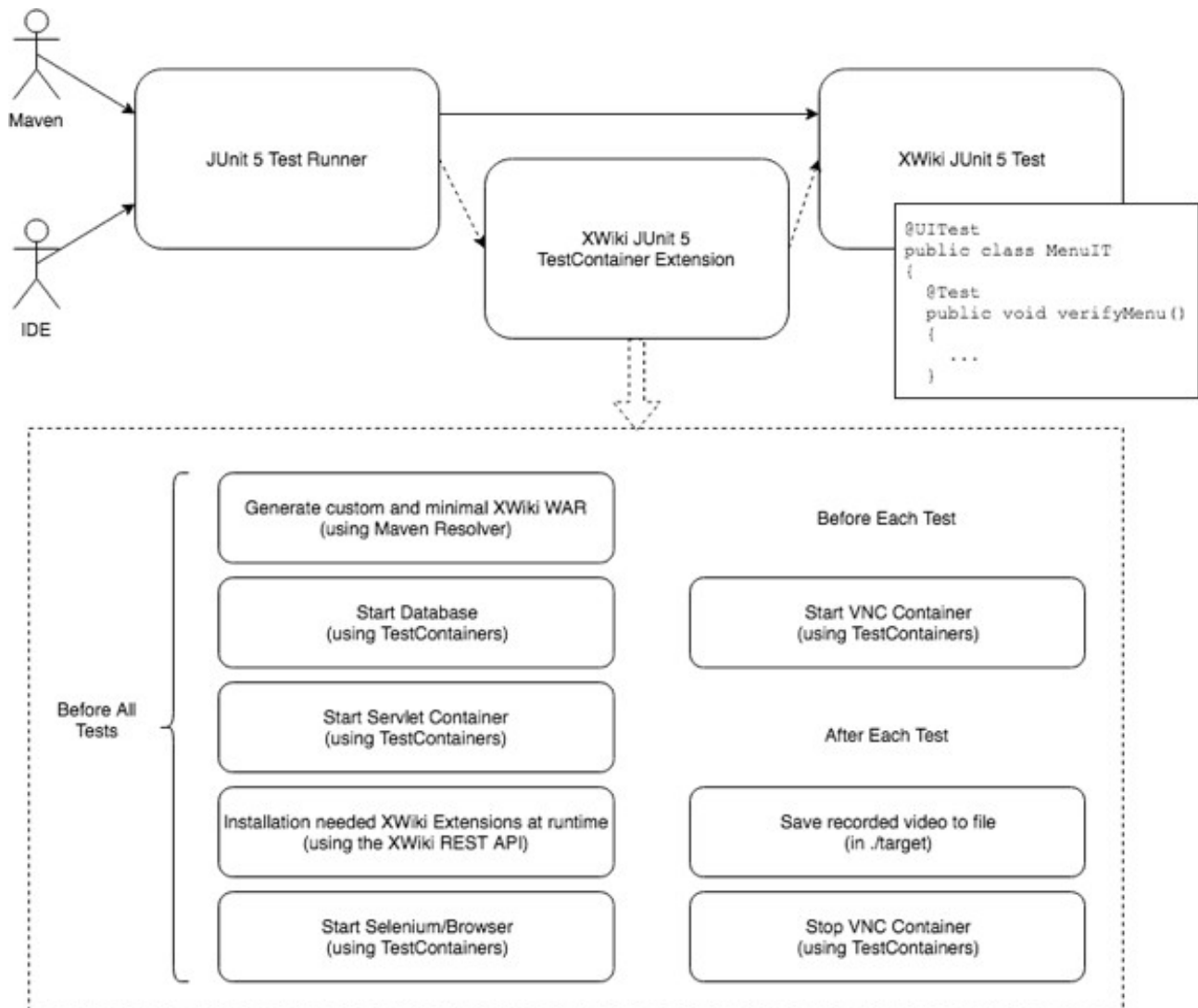


Figure 9: XWiki's Testing Framework Architecture

The following table lists the currently supported configurations in the CAMP/TestContainer-based testing framework. All versions for databases, Servlet containers and browsers can be specified in the framework. In addition any JDBC driver version can also be specified.

Database Type	Servlet Container Type	Browser Type
MySQL	Tomcat	Firefox
PostgreSQL	Jetty	Chrome
HSQLDB	Jetty Standalone (XWik custom packaging)	

Table 36: Supported Configuration Types

Compared to the officially supported list we're currently missing IE, Edge and Safari browsers and the Oracle database. For the Browsers the issue is that there's no Docker container for linux for them and they require

windows machines. For the Oracle database, Oracle doesn't distribute any image and forbids distribution. We will thus need to build our own image and host it in a custom Docker repository.

10.B.3 Validation Method

The XWiki project is following the validation/measurements methods described in D5.3 and which have also been described in the validation treatments in section B1 above.

Additional information is provided in the following table, which represent the set of actions done when measuring the KPI metrics at a given point in time.

KPI	Metric	Method/Process
K01	Global coverage	The XWiki Clover pipeline job executes every week by default and stores the results at http://maven.xwiki.org/site/clover/ . Thus all that's needed for a measure point is to get the global coverage value from the report matching the closest the measure date wanted.
K01/ K03	Coverage from Descartes + Mutation score thanks to Descartes	Process: <ul style="list-style-type: none"> • Pick a module where to execute Descartes • Measure the current coverage in that module using Maven and the command line: <code>mvn clean install -Pquality -Dxwiki.jacoco.instructionRatio=1.00</code> • Measure the current mutation score in that module using Maven: <code>mvn clean install -Pquality -Dxwiki.pitest.skip=false -Dxwiki.pitest.mutationThreshold=100</code> • Record both the current coverage and the current mutation score. • Check <code>target/pit-reports/<date>/issues/index.html</code> and verify if there are "pseudo tested" or "partially tested" methods listed. • Fix the tests so that Descartes doesn't report any pseudo tested or partially tested methods anymore. You run Descartes again with: <code>mvn clean install -Pquality -Dxwiki.pitest.skip=false</code> • Update the jacoco threshold and the mutation scores in the pom.xml file. • Log the results by sending a Pull Request on https://GitHub.com/STAMP-project/Descartes-usecases-output/tree/master/xwiki using the format already defined there.
K01/ K03	Coverage from DSpot + Mutation score thanks to DSpot	Process: <ul style="list-style-type: none"> • Build DSpot from sources (it's only recently that DSpot started to issue releases, before that we had to build it from sources to get the latest fixes/features). • Pick a module where to execute DSpot • Measure and record the current coverage + current mutation score. See line above for the process for doing that. • Execute DSpot, by using the Descartes selector and try with various amplifiers and combinations of amplifiers until you get generated tests or exhaust your testing time budget. • Example command line without any amplifier selected (will only add assertions): <code>java -jar /<path to dspot executable>/dspot-<version>-SNAPSHOT-jar-with-dependencies.jar --path-to-properties dspot.properties --Descartes --verbose --generate-new-test-class --with-comment</code> • Add amplifiers with <code>--amplifiers</code>. • When new tests are generated, commit them in <code>src/test/dspot</code>.

		<ul style="list-style-type: none"> Modify the module's pom.xml file so that the new tests are executed alongside the manually written tests. Add: <pre> <build> <plugins> <!-- Add test source root for executing DSpot-generated tests --> <plugin> <groupId>org.codehaus.mojo</groupId> <artifactId>build-helper-maven-plugin </artifactId> </plugin> </plugins> </build> </pre> Log the results by sending a Pull Request on https://GitHub.com/STAMP-project/dspot-usecases-output/tree/master/xwiki using the format already defined there.
K01	Coverage from CAMP	<p>Process:</p> <ul style="list-style-type: none"> Manually setup Jenkins on your local machine (using the Docker image to make it easy) Create a job in your local Jenkins and copy in it the Clover pipeline described in section B2. Modify it to add the "Docker" Maven profile (–Pdocker) so that it executes the Docker-based functional tests. Run it and gather the global coverage from the generated HTML report. Compare the value with the one from the matching report (without Docker-based tests) from http://maven.xwiki.org/site/clover/ The difference will correspond to the increase of coverage thanks to the configuration testing done. Log the results by sending a Pull Request on https://GitHub.com/STAMP-project/confampl-usecases-output
K01	Coverage from Botsing	N/A, see explanations in section B1.
K02	(all metrics)	<p>Process:</p> <ul style="list-style-type: none"> When noticing a new flaky test (this is done by looking at XWiki's CI and noticing a test failing and succeeding with no change in the sources), create a JIRA issue at https://jira.xwiki.org and fill the "Flickering Test" custom field using the format: <FQN of Test Class>#<flickering test method>, ... Example: org.xwiki.test.ui.extension.ExtensionIT#testUpgrade,t estUninstall,testUpgradeWithMergeConflict,testShowDet ails,testInstall,testDowngrade When fixing a flickering test, simply close the JIRA issue.
K04	(all metrics)	Same process as for K01/Coverage from CAMP above.
K05	(all metrics)	<p>Process:</p> <ul style="list-style-type: none"> When a test fails on XWiki's CI in one of the pipeline jobs running the Docker-based tests, analyze the problem and create a JIRA issue to record it. Label the issue with the label corresponding to the configuration at fault. For example, use "mysql" for a problem related to MySQL, "ie" for a problem related to Internet Explorer, etc.

K06	Number of new configurations tested	Process: <ul style="list-style-type: none"> Add new configurations in the XWiki testing framework located at https://github.com/xwiki/xwiki-platform/tree/master/xwiki-platform-core/xwiki-platform-test/xwiki-platform-test-Docker
	Time to deploy software vs time before STAMP	Process (to be done once): <ul style="list-style-type: none"> Deploy XWiki following the documentation and using the WAR installation. See section B1 for more details. Measure the time it takes. Deploy XWiki again but this time using the Docker images. See section B1 for more details. Measure the time it takes.
K08	(all metrics)	Process: <ul style="list-style-type: none"> Search in XWiki's issue tracker (https://jira.xwiki.org) for issues with stack traces. Extract the stack trace and run EvoCrash/Botsing to try to generate a test that reproduces the stack trace. Try with several frame values. Example command line: <code>java -cp libs/evocrash-<version>.jar:. Run 3</code> Note: Checkout the XWiki code corresponding to the source code having the problem and also make sure to use the XWiki JARs corresponding to that version. Log the results by sending a Pull Request on https://github.com/STAMP-project/evocrash-usecases-output/tree/master/XWIKI using the format already defined there.

Table 37: Validation/Measurements Methods for XWiki

10.B.4 Validation data collection and Measurement method

See table in section B3 which contains the processes and measurement methods.

10.C. Validation Results

10.C.1 KPIs report

K01 - More Execution Paths

The synthetic results for coverage are shown in table XXX.

Metric	Measure Date	Value
Global coverage	2016-12-20	65.29%
	2017-11-09	68.59%
	2018-10-02	68.67%
	2018-11-20	69.52%
Coverage thanks to Descartes	2018-10-02	37% total increase on modules having fixes done thanks to Descartes

Coverage thanks to DSpot	2018-10-02	0.9% increase on modules where DSpot generated tests
Coverage thanks to CAMP	2018-10-02	0% increase
Coverage thanks to Botsing	2018-10-02	N/A

Table 38: K01 Metric Measures for XWiki

Global Coverage

It should be noted that in previous reports XWiki was showing a global coverage of 73.2%. However we've realized that this included coverage on some tests, which are misleading since we're not interested in this coverage and also this [can't be compared between Clover reports](#). Thus we developed some custom Jenkins pipeline script to exclude tests and compute the global coverage from raw Clover data. For example, the XWiki report corresponding to the measure of 2018-10-02 is available at <http://maven.xwiki.org/site/clover/20181002/XWikiReport-20181002-1444.html> (and the Clover report itself for this date is available at <http://maven.xwiki.org/site/clover/20181002/clover-commons+rendering+platform-20181002-1444/dashboard.html>).

By looking at the global coverage values from the table above, we can see that global coverage is progressing regularly and it has increased by 4.24% since the beginning of STAMP which we consider as being a very nice achievement since XWiki had a relatively high coverage at the start of STAMP.

The KPI target is 40% reduction of non-covered code, which corresponds to $(100\% - 65.29\%) * 40/100 = 13.88\%$ increase of global coverage. Thus the objective for XWiki is to reach $65.29\% + 13.88\% = 79.17\%$ by the end of STAMP.

As of today we've reduced the non-covered code by $(100 - 65.29) - (100 - 69.52) = 4.23\%$ (out of 13.88%) and thus we're at 30.47% of the target.

It should also be noted that while it's nice to have the global coverage increased, it's even more important that it doesn't go down in the future and that it goes up steadily. We achieve this on XWiki by having implemented a 2 level strategy:

- At the Maven build level, we fail the build whenever a module sees its coverage decrease.
- At the CI level, we have a Clover job that verifies if the contribution of each single module to the global coverage is positive. If not we report a failure. There are several situations where local coverage can increase and global one decrease (removed modules that had a high coverage, new module that have a coverage lower than other modules, code from local modules tested indirectly from functional tests and refactorings causing the paths to not be executed anymore, etc).

We believe that putting this in place during STAMP was the only way to ensure that XWiki's coverage could continue to go up regularly.

Coverage thanks to Descartes

The following table lists measures corresponding to results that can be found at <https://GitHub.com/STAMP-project/Descartes-usecases-output/tree/master/xwiki> and that were performed till the 2018-10-03. Note that we only reported places where Descartes was able to locate tests and for modules where we made manual fixes to the test to fix the issues raised by Descartes. In addition, all issues we found by running Descartes on XWiki modules were reported in the [Descartes issue tracker](#).

Date	Module	Coverage increase (%)	Mutation score increase (%)	Time spent (mn)
2018-10-02	xwiki-rendering-transformation-macro	4	Not computed	30

2018-10-02	xwiki-commons-velocity	2	Not computed	30
2018-10-02	xwiki-commons-xml	1	1	60
2018-10-02	xwiki-commons-logging-api	1	7	15
2018-10-02	xwiki-commons-component-api	0	4	30
2018-10-02	xwiki-commons-extension-api	3	11	360
2018-10-02	xwiki-commons-filter-api	2	4	120
2018-10-02	xwiki-platform-observation-remote	1	8	120
2018-10-02	xwiki-platform-model	0	2	120
2018-10-02	xwiki-platform-lesscss-default	1	1	30
2018-10-02	xwiki-platform-activeinstalls-client-api	0	10	30
2018-10-02	xwiki-platform-resource-default	0	8	60
2018-10-02	xwiki-platform-url-api	1	7	30
2018-10-02	xwiki-platform-refactoring-api	0	9	465
2018-10-02	xwiki-platform-xar-model	17	24	180
2018-10-02	xwiki-platform-rendering-configuration-default	0	6	10
2018-10-02	xwiki-platform-rendering-xwiki	2	1	20
2018-10-02	xwiki-platform-index-tree-api	2	2	225
2018-10-03	xwiki-platform-store-filesystem	3	7	120
2018-10-03	xwiki-platform-store-filesystem-oldcore	0	4	60
TOTAL		40%	116%	2115 mn

Table 39: Coverage and Mutation Score Contributions by Descartes Strategy

The data show that by fixing issues raised by Descartes we were able to increase the coverage (40% overall increase). We can't easily compute the contribution of Descartes to the global coverage but it would probably be relatively small in the XWiki case. But it is important, as it adds up with other strategies to increase the global coverage for XWiki and it is part of what allowed XWiki's coverage by 4.24% over the period.

Coverage thanks to DSpot

The following table lists measures corresponding to results that can be found at <https://GitHub.com/STAMP-project/dspot-usecases-output/tree/master/xwiki> and that were performed till the 2018-10-29. Note that all issues we found by running DSpot on XWiki modules were reported in the [DSpot issue tracker](#).

Date	Module	Coverage increase (%)	Mutation score increase (%)	Number of generated tests	Time spent (mn)
2018-10-02	commons-cache-infinispan	0	0	0	15
2018-10-02	commons-component-api	0	0	0	15
2018-10-02	commons-component-default	0.216	0	2	60

2018-10-02	commons-component-observation	0	0	0	15
2018-10-02	commons-context	0	0	1	60
2018-10-02	commons-crypto-cipher	0.711	0	1	60
2018-10-16	commons-crypto-common	0	0	1	60
2018-10-16	commons-crypto-pkix	0.21	0	2	60
2018-10-16	commons-crypto-password	0	0	3	60
2018-10-18	commons-component-default	0	0	0	120
2018-10-18	commons-crypto-pkix	0	0	0	15
2018-10-29	commons-component-api	0	0	0	300
TOTAL		1.137%	0%	10	840 mn

Table 40: Coverage and Mutation Score Contributions by DSpot Strategy

It's important to note that DSpot didn't generate new tests (or added assertions to existing tests) in the majority of cases. There were several reasons for this:

- Originally we executed DSpot with the default configuration which only adds assertions and doesn't include any amplifier. Still, even with the default configuration, DSpot should be able to generate assertions in existing tests (which is what we got in the results reported in the table above).
- XWiki has been converting its unit tests to JUnit5 since the beginning of STAMP and till now DSpot was not supporting JUnit5 (only JUnit4). Thus it's logical that it couldn't generate tests for a lot of XWiki modules.
- XWiki is also using custom JUnit Test Suites and those are also not currently recognized by Descartes and DSpot.
- When we tried using several amplifiers, the tests took very long (over 4 hours in the tests we did) and we had to stop them.
- Various bugs prevented the execution of DSpot until recently, for example it wasn't working when using the Descartes selector (was only working with PIT/Gregor).

We reported all those problems and most of them were already addressed and fixed by the DSpot development team (except for the performance, see the next section). We expect that the tests we will do in the next period will yield better metric measures. At the moment the 1.137% increase of coverage is quite low versus the time it took to execute DSpot (840mn).

Now we're still happy that DSpot was able to generate automatically new assertions for XWiki. The real interest of DSpot is that it doesn't require human intervention to generate new tests (as opposed to Descartes for example). Thus, if we're able to make DSpot run faster in the future, the idea will be to integrate it into our CI to automatically whenever code and tests are committed and to commit the generated tests. We're ready to support this on XWiki since, for all the successfully generated tests from the table above, we've committed them in the XWiki source tree under a `src/test/dspot` directory and they're executed by the Maven Surefire plugin.

Coverage thanks to CAMP

Please refer to the section below about KPI K04 to see the list of all configurations that we tested XWiki on. When we measured the additional coverage contributed by executing the functional tests on the new configurations we got a 0% increase. We believe the main reason is related to the way we code XWiki. In our code we use technologies that abstract out the environment/configuration. For example:

- We use an ORM (Hibernate) framework to communicate with databases. So even though we tested on several databases, this didn't result in more code being covered.
- We use the Java Servlet API to have a generic API to abstract out the interactions with the Servlet Container/Application Server. Thus, even though we tested on several Servlet Containers, this didn't result in more code being covered.

Knowing our code there are few places where coverage could be increased by configuration testing in the future but we think the results will be low and that's normal and good (if it wasn't, it would mean the XWiki architecture would not that good and that the code would littered with special cases to handle various environments!). Here are a few places where future progress could lead to additional coverage being brought by configuration testing:

- Having functional tests testing the creation of new subwikis since the code to create new Schema/Databases is specific to the database used (since the version of Hibernate we use doesn't abstract out this part).
- Having functional tests testing LibreOffice integration. If we consider that having LibreOffice installed or not is part of the configuration and that it's not integrated in the baseline measures then testing automatically the integration will result in more coverage.

Our recommendation for the XWiki use case would be drop this metric of coverage thanks to CAMP since it doesn't really make sense because of our architecture.

Coverage thanks to Botsing

See section B1 for an explanation why this is out of scope in this period.

K02 - More flaky tests identified and handled

Results are shown in the following table:

Metric	Date	Value
Number of flaky tests identified and handled	2016-12-31	5 (source) (baseline)
Number of flaky tests fixed	2016-12-31	1 (source) (baseline)
Percentage of flaky tests fixed compared to the total number of tests	2016-12-31 (baseline)	0.01003814495% (9962 tests on 2016-12-20)
Number of flaky tests identified and handled	2018-08-20	15 (source)
Number of flaky tests fixed	2018-08-20	3 (source)
Percentage of flaky tests fixed compared to the total number of tests	2018-08-20	N/A. We had some issues in our Clover pipeline and it did not print the number of tests. ²⁴
Number of flaky tests identified and handled	2018-10-02	17 (source)
Number of flaky tests fixed	2018-10-02	4 (source)
Percentage of flaky tests fixed compared to the total number of tests	2018-10-02	0.03698566805 (10815 tests on 2018-10-20)
Improvement	2018-10-02	+368.45%

Table 41: Flaky Test Measurements

The KPI target for K02 is an increase of 20% of flaky tests fixed compared to the total number of tests. Thus we are well ahead of that number with 368.45% so far.

²⁴ See for example <http://maven.xwiki.org/site/clover/20180528/clover-commons+rendering+platform-20180528-2133/dashboard.html>

Some explanations:

- Before 2016 the XWiki project was not recording flaky tests in its issue tracker, which accounts for the low number at the end of 2016. This explains why the percentage increase is so important. Thus it's more important to look at the number of flaky tests identified and handles and at the number of flaky tests fixed.
- XWiki flaky tests are flickering due to automated UI tests (e.g. javascript executing and assertions being done before the HTML DOM elements have updated). They are extremely difficult to fix and the flickers reported in the XWiki issue tracker are tests that couldn't be fixed over 1 or 2 days. Thus temporary flicker tests are not counted since they didn't have time to be considered flickers for XWiki.

What's really good and useful for the XWiki project is that by focusing on flickers thanks to STAMP, we were able to set up a strategy to handle flicker and to avoid sending false positives build breaking emails to developers. More specifically flickers are now recorded in the XWiki issue tracker and the CI Jenkins pipeline script avoid sending emails when only flickering tests are failing the build. This has two effects:

- Less work for developers and happier developers
- More trust in the CI
- Faster releases of XWiki (before that, at release time, we had to spend substantial time to analyze failing tests to filter out flickering ones)

K03 - Better test quality

The results for computing the mutation score metric (measured by Descartes) are available in the tables for KPI K01 above. The reason is because we measured both coverage and mutation scores together when testing Descartes and DSpot.

To summarize, we got a +112% increase of mutation score thanks to Descartes over the period. But that's the sum of all the increases for each modules where we made improvements to the tests, as suggested by Descartes. It's hard to understand how much global mutation score increase that would represent over the whole XWiki code base since we don't have a way to measure that value easily. To be more precise, when we tried to measure the mutation score for the whole of XWiki using the PITmp plugin, it took a very long time just for xwiki-rendering (over 7 hours) and an extrapolation on the whole code base leads to times over several weeks (if it can succeed and not stop because of lack of memory), which is not practical. It's even less practical since we would need to execute this every time we fix tests thanks to Descartes or DSpot to exclude potential changes due to daily work of XWiki developers.

The objective for K03 is 20% increase of mutation scores. What we see is that Descartes is increasing mutation scores by 1 to 24%, with an average of 6.5% per module, so still far from the objective. This low value (but nevertheless positive) could be explained by XWiki tests being already of good quality (the average mutation score for modules where we applied Descartes is over 60%), so there's not much margin for improvement.

Regarding DSpot, we didn't get any mutation score increase. That is most certainly due to the reasons explained above in K01.

K04 - More unique invocation traces

The following 26 configurations were tested automatically during the period, using the CAMP/TestContainer-based testing framework developed by XWiki for STAMP:

Date	DB Type	DB Version	JDBC Version	Servlet engine Type	Servlet engine Version	Browser Type	Browser Version
2018-10-22	MySQL	5.7.22	5.1.45	Tomcat	8.5.32	Firefox	60.0.2
2018-10-22	MySQL	5.7.22	5.1.45	Tomcat	8.5.32	Chrome	67.0.3396.87

2018-11-03	MySQL	5.7.22	5.1.45	Jetty Standalone	9.4.8.v20171121	Firefox	60.0.2
2018-11-03	MySQL	5.7.22	5.1.45	Jetty Standalone	9.4.8.v20171121	Chrome	67.0.3396.87
2018-11-03	PostgreSQL	9.6.8	42.1.4	Tomcat	8.5.32	Firefox	60.0.2
2018-11-03	PostgreSQL	9.6.8	42.1.4	Tomcat	8.5.32	Chrome	67.0.3396.87
2018-11-03	PostgreSQL	9.6.8	42.1.4	Jetty Standalone	9.4.8.v20171121	Chrome	67.0.3396.87
2018-11-03	PostgreSQL	9.6.8	42.1.4	Jetty Standalone	9.4.8.v20171121	Firefox	60.0.2
2018-11-03	HSQLDB	2.4.1	2.4.1	Tomcat	8.5.32	Chrome	67.0.3396.87
2018-11-03	HSQLDB	2.4.1	2.4.1	Tomcat	8.5.32	Firefox	60.0.2
2018-11-03	HSQLDB	2.4.1	2.4.1	Jetty Standalone	9.4.8.v20171121	Chrome	67.0.3396.87
2018-11-03	HSQLDB	2.4.1	2.4.1	Jetty Standalone	9.4.8.v20171121	Firefox	60.0.2
2018-11-03	MySQL	5.7.22	5.1.45	Jetty	9.4.12	Firefox	60.0.2
2018-11-03	MySQL	5.7.22	5.1.45	Tomcat	9.0.12	Firefox	60.0.2
2018-11-05	HSQLDB	2.4.1	2.4.1	Jetty Standalone	9.4.8.v20171121	Firefox	63.0
2018-11-05	HSQLDB	2.4.1	2.4.1	Jetty Standalone	9.4.8.v20171121	Chrome	70.0.3538.77
2018-11-05	MySQL	5.7.22	5.1.45	Tomcat	8.5.32	Chrome	70.0.3538.77
2018-11-05	PostgreSQL	9.6.8	42.1.4	Jetty Standalone	9.4.8.v20171121	Chrome	70.0.3538.77
2018-11-05	HSQLDB	2.4.1	2.4.1	Tomcat	8.5.32	Firefox	63.0
2018-11-05	HSQLDB	2.4.1	2.4.1	Tomcat	8.5.34	Firefox	63.0
2018-11-06	HSQLDB	2.4.1	2.4.1	Jetty	9.4.12.v20180830	Firefox	63.0
2018-11-11	MySQL	5.7.24	5.1.45	Tomcat	8.5.35	Chrome	70.0.3538.77
2018-11-11	PostgreSQL	9.6.11	42.1.4	Jetty	9.4.12.v20180830	Chrome	70.0.3538.77
2018-11-12	PostgreSQL	11.1	42.1.4	Jetty	9.4.12.v20180830	Chrome	70.0.3538.77
2018-11-12	PostgreSQL	11.1	42.2.5	Jetty	9.4.12.v20180830	Chrome	70.0.3538.77

2018-11-13	PostgreSQL	11.1	42.2.5	Jetty	9.2.26	Chrome	70.0.3538.77
------------	------------	------	--------	-------	--------	--------	--------------

Table 42: Configurations Tested

More generically the following configurations were defined as supported by the XWiki open source project and implemented (the exact versions used are the latest available ones and this is why we call these floating configurations):

DB Type	DB Version	JDBC Version	Servlet engine Type	Servlet engine Version	Browser Type	Browser Version
MySQL	5.7.x	5.1.45	Tomcat	8.5.x	Chrome	Selenium
MySQL	5.7.x	5.1.45	Tomcat	8.x	Chrome	Selenium
MySQL	5.5.x	5.1.45	Tomcat	8.x	Chrome	Selenium
PostgreSQL	11.x	42.2.5	Jetty	9.2.x	Chrome	Selenium
PostgreSQL	11.x	42.2.5	Jetty	9.x	Chrome	Selenium
PostgreSQL	9.4.x	42.2.5	Jetty	9.x	Chrome	Selenium
PostgreSQL	9.6.x	42.2.5	Jetty	9.x	Chrome	Selenium
HSQLDB	Version from XWiki	Version from XWiki	Jetty Standalone	Version from XWiki	Firefox	Selenium

Table 43: Floating Configurations Implemented

The XWiki project has hundreds of functional tests (and about 10K tests in total) and the functional tests so far were all executed on a single configuration (HSQLDB database and Jetty Standalone, on Firefox). During the period, we focused on implementing all the configurations listed and converted one functional test of XWiki (MenuIT) to execute on all these configurations.

The results from these two tables are used for measuring KPIs K04, K05 & K06.

Regarding K04, and as discussed in K01/"Coverage thanks to CAMP" above, we didn't get any additional coverage coming from the test executed on various configurations, and this was expected. See above for explanations. We are thus very far from being able to reach the 20% of coverage improvement thanks to configuration testing.

Metric	Measure date	Value
More unique invocation traces (microservices) or coverage increase (monolithic)	2018-10-17	0%

Table 44: Coverage Improvement Thanks to Configuration Testing

K05 - System-specific bugs

Results are shown in the following table.

Metric	Measure date	Value
--------	--------------	-------

Percentage improvement vs baseline	2016-12-20	CAMP/TestContainers not ready yet
Percentage improvement vs baseline	2017-11-09	CAMP/TestContainers not ready yet
Number of new configuration-related bugs discovered	2018-10-22	2 (source)
Percentage improvement vs baseline	2018-10-22	$2 \times 100 / 56 = 3.571428571$
Number of existing configuration-related bugs from 1/1/2017 to 31/6/2018 (1.5 years). This is approx the time in STAMP since the first usable version of CAMP/TestContainers and the end of the project.	2016-12-20	56 (source)
Number of new configuration-related bugs discovered	2018-11-06	4 (source)
Percentage improvement vs baseline	2018-11-06	$4 \times 100 / 56 = 7.142857143$

Table 45: System-Specific Bugs Found Thanks to Configuration Testing

The target for this KPI is to increase by 30% the new configuration-related bugs discovered. As of 2018-11-06, we're at 7.14%. This is really promising as we were only able to start running tests on all the supported configurations quite late since it required the framework to be developed and stable enough and its usage only really started in November 2018. In addition we currently only have a single functional test that's been migrated to this testing framework (MenuIT) and it's now going to be relatively easy to migrate a lot more tests and thus have better results.

Also note that the reason for the relatively low 7.14% value is because XWiki has a large community and lots of system-specific bugs are discovered by the community and thus the baseline value of 56 configuration-related bugs found in a 1.5 years is a large number, and increasing it by 30% is challenging.

K06 - More configurations / Faster tests

Results are shown in the following table.

Metric	Measure Date	Value
Number of configurations tested	2016-12-20	1
Time to deploy software vs time before STAMP	2016-12-20	2 hours
Time to deploy software vs time before STAMP	2017-11-09	5 minutes
Number of new configurations tested	2018-10-22	2
Number of new configurations tested	2018-11-06	21
Number of new configurations tested	2018-11-13	26

Table 46: Number of New Configurations Tested

The target for the number of new configuration was 50% more configurations but since the XWiki project was testing automatically on only one before STAMP, we now 26 times more configuration, i.e 2600% increase. What makes more sense is to look at the 8 floating configurations (see table above in K04), which is substantial and which is going to bring huge value for the XWiki project as they'll ensure that XWiki works on any new versions of these configurations.

Regarding the time to deploy the progress is also very significant: $(120 - 5) * 100/120 = 96\%$ of time reduction.

K08 - More crash-replicating tests

Results are shown in the following table.

Metrics	Measure Date	Value
Number of fixed production crash issues not reproduced thanks to EvoCrash and which have test cases, since the beginning of STAMP	2018-10-05	5 (source)
Number of issues reproduced by Evocrash	2018-10-05	3 (source)
% of crash replicating test cases	2018-10-05	$((5+3)/5 - 1) * 100 = 60\%$

Table 47: Percentage of Crash-Replicating Test Cases

The target for K08 is to increase the number of crash replicating test cases by at least 70%. We're currently at 60%. One reason is because we don't have many production crashes issues that were fixed with tests on the XWiki projects. Usually that's because these tests are hard to make and depend on a specific configuration. Now that we're adding configuration testing in XWiki, it's possible that it'll be easier to write tests for a given configuration and thus that we may have more of these issues with tests proving the fix.

10.C.2 Qualitative evaluation and Recommendations

Descartes

We got an average of 2% coverage increase on XWiki modules where we fixed the "pseudo-tested" and "partially-tested" issues reported by Descartes (average coverage on these module was 62% before applying Descartes). We also got an average of 6.45% mutation score increase on these same modules (average mutation score on these modules was 60% before applying Descartes). This is encouraging but it remains in the low range (target objective is 20% for the mutation score increase). On the other hand, XWiki modules already have a relatively high coverage and high mutation scores (60%+) which could affect the effectiveness of Descartes.

In general, Descartes is working well and is easy to use for us thanks to the Maven plugin. Here are some ideas to improve it:

- Make it recognize more JUnit tests. Right now it [doesn't recognize well custom test suites](#) and thus a lot of XWiki tests are not recognized (such as parametrized tests using the [XWiki Rendering testing framework](#)) and thus no mutation score is computed for them and no issues are raised with indications on how to improve their quality. The same seems to be true for [Abstract tests](#) (i.e. test classes extending abstract classes).
- We are experiencing some [mutation score stability issues](#) and it's critical to XWiki that the mutation score remains stable. The reason is that we fail the build when the mutation score decreases and thus we should only fail when quality is lowered and not send false positives since this will lead to stopping the execution of Descartes as part of our build.
- When our build fails because of lowered mutation score, it would help a lot if we could easily find out

what was changed from before from the point of view of Descartes, i.e. have some [Descartes report diff](#) to pinpoint the problem.

- It would also be interesting for Descartes to improve the reports it generates by [indicating the code directly exercised by tests](#). Tests can exercise some code indirectly (i.e. not test this code explicitly) and Descartes can then report that some mutants were not killed. However, in practice, this code is not tested and it's the same situation as not having tests at all as far as this code is concerned. The really interesting cases are when unkillable mutants belong to methods that are directly called from the test methods. It would be nice to be able to show this clearly in the report to make it simpler to find hot spots to fix.

DSpot

Here are some ideas to improve DSpot:

- We have had issues running DSpot to generate new tests (we could generate new assertions in existing tests). When we tried using several amplifiers to generate new tests, it took hours and we had to stop the execution (we stopped after 4 hours). Thus it would be great if DSpot could find one or several amplifiers that bring the most value and that execute fast and have them used by default.
- XWiki tests are being migrated every day to JUnit5. DSpot was not supporting JUnit5 and thus there were less and less tests where we could run DSpot on. This was [raised](#) and is a work being addressed and we should be able to test this in the near future.
- Since DSpot takes very long to execute, an idea is to have a Jenkins plugin (or better a pipeline script) to automatically [call DSpot on the diff from a commit](#). Use case we'd like to implement on XWiki:
 - A developer commits code
 - An XWiki CI job is triggered and inside this job DSpot is called with the diff info from the commit
 - The pipeline script commits the generated new tests in XWiki's SCM
 - XWiki's Maven build is modified so that amplified tests are executed as part of XWiki's build as standard JUnit tests

Generally speaking it's quite hard to use DSpot today and it's very time consuming. It's a tool with a huge promise (automatically generating tests). We need to find ways to speed up its execution and to run it automatically so that there's no manual intervention from developers. This is when it'll deliver its full potential.

CAMP/TestContainers

During WP2 work, XWiki developed an add-on for CAMP that we called CAMP/TestContainers. XWiki supports a fixed set of configurations and the original CAMP only makes sense when you have a very large number of configurations that you want to test and experiment on (on multiple axis). Thus, XWiki spent a very large amount of time developing a Docker-based testing framework to use to test various configurations (which were used to measure KPIs 04, 05 and 06). This CAMP/TestContainers-based testing framework has been hugely successful so far in the little timespan we've been able to test it (a working version was finished about a month ago). It has allowed us to test 25 configurations for XWiki and to find some issues.

Here are some ideas to improve it in the future:

- Provide support for more configurations.
 - Add support for Oracle database (not easy since Oracle doesn't distribute an image nor provide easy access to its JDBC drivers).
 - Buy and configure CI agents on Windows so that we can add support for testing XWiki on IE, Edge and Safari browsers.
 - Add support for running LibreOffice to test office import scenarios
- Migrate XWiki's hundreds of configuration tests to this new framework.
- Work on improving the execution performance. Currently starting all the needed Docker containers and deploying XWiki in them takes about 3 minutes. We need to try to get this down to 2 minutes, which is about the time it was taking with our previous approach (which was supporting only 1 configuration and not using Docker containers).

Botsing

We have been running Evocrash on XWiki and haven't started testing Botsing yet (was released a month ago), which we'll do in the next period. During our testing of Evocrash we analyzed the crash-reproducing tests to gauge how useful they were and we found out that:

- Evocrash is mostly useful as a code comprehension tools, allowing developers new to the code to understand it and understand how a given crash situation can occur.
- The generated tests usually cannot be committed as is as regression tests, simply by negating the assertions. The reason is that most of the time the problem happens higher in the call chain and you'd write more explicit tests by testing conditions at a different level, closer to the user use case.

In the next period it would be great if Botsing could focus on generating regression tests instead of crash-reproducing tests, i.e. tests that can committed in the code base.

Also we find that it would provide even more value if Botsing could be [integrated with JIRA/GitHub](#) using the following scenario:

- A user logs an issue in the XWiki issue tracker (JIRA) with a stack trace
- Automatically, a process is started (JIRA integration) which executes Botsing on the extracted stack trace
- If successful to reproduce the crash, a comment with the test is added to the JIRA issue and a GitHub PR is opened with a regression test.

10.C.3 Answer to Validation Questions

VQ1 - Can the STAMP tools assist software developers to trigger areas of code that are not tested?

Descartes and DSpot's primary goals are to increase the quality of the tests and not to increase the code coverage. However, the experiments show that Descartes increased by an average of 2% the code coverage on modules where it was applied and that DSpot increased by an average 1.1% the code coverage on modules where it was applied. It should be noted that since we run DSpot with the Descartes selector, their effects are probably not cumulative (but we cannot conclude on this since in our tests we didn't get coverage increase on the same modules where Descartes and DSpot were executed. This could actually be a sign that they're exclusive).

As of today the CAMP/TestContainers has failed to increase the coverage for XWiki and it was expected. It should increase coverage a bit in the future when we add more diverse configurations but the increase will remain small. It should be noted that it's not the main goal of CAMP/TestContainers to increase code coverage; they are meant to reduce configuration-related bugs.

Regarding Botsing, right now it's not generating any regression tests and thus has no impact on code coverage. In the future, when it's able to generate regression tests, it'll certainly contribute to increase code coverage since it'll test branches that were not tested before.

VQ2 - Can STAMP tools increase the level of confidence about test cases?

Since we started to identify and handle flaky tests, it's much simpler to perform the following tasks:

- Plan the fixing of flaky tests since they're now recorded in our issue tracker (JIRA). Note that they're difficult to fix but at least we know them and they regularly come to mind and over time it's possible to fix them.
- Perform XWiki releases. Before the identification of flaky tests it was taking a long time for the Release Manager to identify failing tests that were "normal"/"expected". It is now a no-brainer.

So the flaky tests strategy definitely helps increase the level of confidence of the non-flaky tests and more generally in the CI.

Regarding Descartes/DSpot increasing the mutation score, the answer is less obvious. They do increase the mutation score but it's hard to gauge how much they increase the level of confidence of developers about the

tests that were modified thanks to them. In a lot of cases, developers were aware of missing assertions or branches that were not tested and they had voluntarily decided to not test them for time constraints, preferring the addition of other tests for features that were not tested at all. We believe Descartes/DSpot are especially good when you already have a fully covered code base with no more obvious tests to add. Another approach for using Descartes would be to integrate it as a developer tool when using his/her IDE, while writing tests. If Descartes was executed automatically it could provide interesting insight at that time. This is something we need to experience. One issue being that the XWiki developers using various IDEs, including IntelliJ IDEA for which there is no Descartes plugin at the moment. However some others using Eclipse and we could start by testing that in the next period.

However, what is the most important for XWiki is that the quality of new code is not less than the quality of existing code and this is where our strategy of failing the build when new introduced code has a lower mutation score is interesting. After about 3 months of having this in place, we still cannot conclude about its effectiveness since we're having mutation score stability issue and the strategy is thus not working for the moment. We hope to be able to stabilize it in the next period.

VQ3 - Can STAMP tools increase developers confidence in running the SUT under various environments?

This is the area that we find the most successful for XWiki. We are now able to run XWiki in more than ten different supported configurations and this helps a lot flush out environment issues and increases the quality of the XWiki software for our users. It also helps prepare the future by knowing if XWiki has problems deploying in future configurations that we know we will want to support.

VQ4 - Can STAMP tools speed up the test development process?

As for VQ3, the biggest win is the huge reduction of time it takes for developers to test XWiki in different configurations. They used to have to install various databases, browsers, Servlet containers on their machines and it was difficult, sometimes even not possible and they had to resort to installing on some remote servers, which made code debugging very difficult to achieve. They can now do so under 5 minutes for any supported configuration.

DSpot yields great expectations (automatically generating tests) but has yet to prove its worth. At the moment, it's a bit hard to use and doesn't generate many tests. It also takes too long to execute so that we can conclude that it is helping developers in the test development process.

Botsing is an interesting tool to help understand the failing code but it will be most effective when it can generate automatically regression tests and when it is integrated with JIRA/GitHub.

10.C.4 Next Steps for Validation

In the next period, the XWiki main goals will be:

- Play more with the DSpot amplifiers to be able to generate more tests and find "good" combinations that can be used throughout the XWiki project. Also test on JUnit5 and integrate in the CI by using a diff approach.
- Work with Descartes developers to stabilize the mutation score and support recognizing more tests.
- Add some exotic configurations for the usage of CAMP/TestContainers in XWiki to try to get some additional coverage.
- Test Botsing (successor of Evocrash) and try to get a Botsing integration with JIRA and GitHub working to have an even more useful workflow.

11. OW2 Use Case Validation

OW2 Use Case Highlights

- Additional projects recruited from community because of tricky negotiations to involve project leaders whose in-depth cooperation is required. Results currently available for the Sat4j project.
- While collected metrics do not reflect it, we verified with project leader Daniel Le Berre that STAMP tools, especially Descartes, resulted in improvements in the Sat4j test suite.
- Software with modules and using a lot the strategy design pattern such as Sat4j is a challenge for test amplification because only one of the available strategies is chosen at each runtime.
- Extensive interaction with the development teams, reporting and helping fix tools shortcomings and recommending improvements such as adding assertions testing null to Dspot.

11.A. Use Case Description

11.A.1 Target Software

The OW2 use case aims at experimenting STAMP over a sample of four projects representative of the community code base. The sample initially foreseen included Joram (middleware), Lutece (content management) and Sat4j and ASM (both software engineering). OW2 being an open source community, the execution of the use case depends on the cooperation of independent project leaders. As the use case unfolded it became clear that project leaders had changing priorities. This appeared as a major difficulty and as projects have been approached, the sample had to be adjusted. To be on the safe side and ensure four projects are actually included in the use case; the selection is now over-sampled to six projects consisting of the Sat4j software engineering project, the AuthzForce middleware, Lutece content management, application platforms DocDoku and Bonita, and the SeedStack framework. While these projects have been approached and agreed to take part in the use case, work is slow because project leaders are not employees of OW2, they cooperate only on a best effort basis.

Sat4j is the target software for the OW2 use case described in this report. We tested DSpot and Descartes on Sat4j project with the help of the project leader and the developers. OW2 plans to experiment with Botsing and CAMP, the roadmap and target software for these future validation activities are discussed in section C.3.

Sat4j is a Java library for solving boolean satisfaction and optimization problems. It can solve SAT, MAXSAT, Pseudo-Boolean, Minimally Unsatisfiable Subset (MUS) problems. Being written in Java, the promise is not to be the fastest one to solve those problems (a SAT solver in Java is about 3.25 times slower than its counterpart in C++), but to be full featured, robust, user friendly, and to follow Java design guidelines and code conventions. Sat4j is used by millions of people across the globe, and has been downloaded more than 200,000 times since January 2006, ranking it 15th among the OW2's most downloaded projects.

The project is a challenge for STAMP. Indeed, Sat4j is a modular, configurable library of tools. It is available as a Java library or as a set of command line tools. The command line interface is hardly tested, because it relies on features such as system exit code which are incompatible with automated tests. Furthermore, the library uses a lot the strategy design pattern: as such, at runtime, only one of the implementations available for each strategy is chosen. It is not possible to cover all the code in one run. This explains why the 2K+ unit tests only cover 40% of the code.

The project was originally a monolithic library. With its adoption in Eclipse, the library has been refactored into several bundles (core, pseudo, maxsat, sat), two of them being shipped with Eclipse (core and pseudo). Those two bundles do not have any external dependencies. All external dependencies are now found in the other bundles. All bundles depend on the core one. As a consequence, the core bundle contains a fair amount of classes mainly useful in other bundles but located in this bundle to be easily shared. OW2 decides to focus its attention on the core bundle for the validation campaign reported here. The particularity of that bundle is another hurdle to amplify its tests.

	Sat4j (global)	Sat4j (core module)
Java classes (Sonarqube)	485	232
Lines of code (Sonarqube)	44k	18k
Code Coverage (Jacoco)	40.6%	49%
Unit tests (Sonarqube)	2839	1015
Mutation Score (Descartes)	N/A	30%

Table 48: Key Metrics of OW2's Use Case Target Software

11.A.2 Experimentation environment

Sat4j uses both Ant and Maven for project management. Ant is used to build the command line tools. Maven is used for continuous integration and code analysis. This project uses OW2 Gitlab service for continuous integration. Until now the use case did not focus on integrating STAMP tools into the CI, this will be done during the next period with the help of project leaders.

The technical environment of the use case includes one local configuration and one remote virtual machine (VM).

	RAM	CPU	Frequency (per core)	Storage
Local configuration	8 Gb	i5-5200U × 4	2.2 GHz	256 Go SSD
VM configuration	16 Gb	Vcores * 4	3.1 GHz	100 Go SSD Raid

Table 49: OW2 Experimentation Environment

11.A.3 Expected Improvements

Expectations in this use case are from two point of views: from the perspective of OW2 and from that of the SAT4J project leader. For OW2, the objective is to evaluate the feasibility to offer additional QA tools to the dozens of open source projects forming the OW2 community code base. STAMP will enhance OW2's quality and market readiness assessment²⁵ at OW2. The objective is to improve the extent to which the community projects are tested in order to demonstrate their superior quality and attention to market readiness. Technically the core focus is to improve code coverage. Meanwhile Daniel Le Berre is the SAT4J project leader and a teacher. His intention is to use STAMP tools to show students how strong tests can still be enhanced, i.e., to demonstrate there is always room for improvement. As a researcher and creator of SAT4J, he is interested in STAMP tools to enhance the code coverage. As stated above, SAT4J structure makes it hard to cover all cases. This is why, as shown in the table above, code coverage is quite low, especially compared to other projects such as XWiki. Currently, new test cases are regularly reported to Daniel Le Berre by users each time they face a problem. The expectation is to reduce this workload by using STAMP tools to enhance code coverage.

²⁵<https://oscar.ow2.org/view/wiki/>

11.A.4 Business Relevance

As a global open source, non-profit organization, OW2's mission is to foster a code base of open source code infrastructure software and to grow a community of open source code developers. OW2 has set two objectives to help fulfill this mission. The first one is to help project leaders deliver quality software. We do that by providing them on the OW2 technical infrastructure a portfolio of QA and governance resources. Making available quality software keeps the OW2 code base attractive. STAMP tools shall be added to the QA tools offered to the projects. The second objective is to make OW2 project acceptable by mainstream decision makers who are not familiar with open source software and need to be convinced of the quality of open source components. We are developing a methodology to position OW2 projects along a hierarchy of «Market Readiness Levels²⁶» and we are working on updating our assessment models so that those projects that use STAMP will earn credits that will improve their market readiness ratings as a recognition of higher attention to software quality.

11.B. Validation Experimental Method

During this period, we focused on experimentations regarding validation question 1 (VQ1) and validation question 2 (VQ2). In this section, we present the strategies we used to answer those questions.

11.B.1 Validation Treatments

In order to measure KPIs related to VQ1 and VQ2, we need to define the initial metric (the control) and the final metric once we have used the tool (the treatment). We present in the table below our strategies regarding K01, K02 and K03. During this period, since we did not work with CAMP and Botsing, we have not been able to collect KPIs related to those tools, K04 and K08. This will be done in the next period, as detailed in section C3.

Code Coverage	K01	<ul style="list-style-type: none"> • DSpot • Descartes 	<ul style="list-style-type: none"> • The control metric is the code coverage calculated by Jacoco on Sat4j core module before using any tool. • There are two treatments: <ul style="list-style-type: none"> - one consists in executing DSpot with various parameters and then integrate new generated amplified tests in the existing test suite and run Jacoco to get the new code coverage metric. - the other consists in executing Descartes, analyse the results and improve manually the existing test suite by adding new tests, then run Jacoco to get the new code coverage metric.
Flaky tests	K02		<ul style="list-style-type: none"> • The control metric is the number of tests behaving flaky in the beginning of the period. • The treatment consists in manually fixing those tests and evaluating if the flakiness is still present.

²⁶ <https://oscar.ow2.org/view/MRL/>

Mutation Score	K03	<ul style="list-style-type: none"> • DSpot • Descartes 	<ul style="list-style-type: none"> • The control metric is the mutation score calculated by Descartes on SAT4J core module before using any tool. • There are two treatments: <ul style="list-style-type: none"> - one consists in executing DSpot with various parameters and then integrate the new generated amplified tests in the existing test suite and run Descartes to get the new mutation score metric. - the other consists in executing Descartes, analyse results and improve manually the existing test suite by adding new tests, then run again Descartes to get the new mutation score metric.
----------------	-----	--	---

Table 50: OW2 Validation Treatments

11.B.2 Validation Target Objects and Tasks

In this section, we present our methodology for each KPI in order to achieve our goals regarding VQ1 and VQ2.

K01

To improve code coverage, we focused in this period on two strategies:

- Use DSpot to generate new amplified tests. In order to comply the requirements, during our experimentations with DSpot, we used each time the test criterion 'JacocoCoverageSelector'. This criterion ensures us that DSpot keeps in output only generated tests that improve code coverage. DSpot offers different amplifiers, we experimented them and analysed their impacts on code coverage.
- Use Descartes to detect weakly tested parts in the source code. This is actually a consequence of the work on VQ2 and K03. The main goal of Descartes is to spot new pseudo-tested methods. By working on improving the test suite, there is also an implication on code coverage.

K02

To improve detection of flakiness, we worked on this strategy:

- identify failed builds in the CI (<https://gitlab.ow2.org/sat4j/sat4j/pipelines>)
- detect cases where a flaky test is the source of the failure.
- manually fix the test and commit.
- check if the flakiness is still present.

K03

To improve mutation score, we focused in this period on two strategies:

- Use DSpot to generate new amplified tests. In order to comply the requirements, during our experimentations with DSpot, we used each time the test criterion 'PitMutantScoreSelector'. This criterion ensures us that DSpot will give us in output only generated tests that improve mutation score. Our experimentations consisted in targeting different parts of the test suite and analysing the impact on mutation score calculated by Descartes.
- Use Descartes to detect weak test cases in the existing test suite. With each iteration of Descartes, our experimentation consisted in analysing the reports, improving the test suite either by adding new test cases or new test classes, then running Descartes again and comparing mutation scores.

11.B.3 Validation Method

In this section, we present the different configurations experimented with each tool.

DSpot

All experimentations on SAT4J project are published on Github:

<https://GitHub.com/STAMP-project/dspot-usecases-output/tree/master/OW2/Sat4j>

To make this report self-contained, we report the main table below:

Date	Test criteria	Amplifiers	Class files	Test cases
20181029	PitMutantSelector	-	10	12
20181030-1140	PitMutantSelector	-	1	1
20181030-1321	JacocoCoverageSelector	-	1	1
20181030-1515	JacocoCoverageSelector	-	4	4
20181031-1100	JacocoCoverageSelector	NumberLiteralAmplifier: MethodAdd:MethodRe move:MethodGenerato rAmplifier:NullifierAmpli fier	***	***
20181031-1600	PitMutantSelector	-	1	1
20181101-0945	JacocoCoverageSelector	NumberLiteralAmplifier: MethodAdd:MethodRe move	***	***
20181102-0920	JacocoCoverageSelector	MethodRemove	6	241
20181102-0930	PitMutantSelector	-	2	2
20181102-0950	JacocoCoverageSelector	NumberLiteralAmplifier: MethodAdd	13	1564
20181116-1430	JacocoCoverageSelector	MethodRemove	5	242

*Note: *** those experimentations failed due to an error in the execution, this will be discussed further in the document.*

Table 51: DSpot Experimentations on Sat4j

All experimentations were made with the latest DSpot release compiled source. This explains why in our experimentations we have sometimes the same configuration at different times. Each experimentation folder in the Github repository contains DSpot output (*dpost-out*) and Jacoco reports (*jacoco*).

Descartes

We used the maven plugin for our experimentations. All experimentations on the SAT4J project are published on Github:

<https://GitHub.com/STAMP-project/Descartes-usecases-output/tree/master/OW2/Sat4i>

In this repository, table, *Sat4j_reports.ods* lists all methods detected by Descartes for each execution. We keep track of changes on each iteration. While not meant to be read, the snapshot below outlines the color system used to visually spot the new mutants (green color), the ones killed by next iterations (red color) and the new mutants killed by next iterations (orange color). The actual table related to this snapshot is here:

https://GitHub.com/STAMP-project/Descartes-usecases-output/blob/master/OW2/Sat4i/Sat4i_reports.ods

15/09/18

15/09/18

org.s4fs.core.ASVofFactory.isoverlapped()	org.s4fs.core.ASVofFactory.isoverlapped()	
org.s4fs.core.ConstraintGroup.getConstraint()	org.s4fs.core.ConstraintGroup.getConstraint()	
org.s4fs.core.Vec.isEmpty()	org.s4fs.core.Vec.isEmpty()	
org.s4fs.core.VecInt.shrink()	org.s4fs.core.VecInt.shrink()	
org.s4fs.core.VecInt.sortInplace()	org.s4fs.core.VecInt.sortInplace()	
org.s4fs.core.VecInt.unsafeGetU()	org.s4fs.core.VecInt.unsafeGetU()	
org.s4fs.mimsl.constraints.cad.MarketCard.normalized()	org.s4fs.mimsl.constraints.cad.MarketCard.normalized()	
org.s4fs.mimsl.constraints.cad.MarketCard.removing(s4fs.specs.UnlPropagatorListener)	org.s4fs.mimsl.constraints.cad.MarketCard.removing(s4fs.specs.UnlPropagatorListener)	
org.s4fs.mimsl.constraints.cad.MarketCard.compilePropagating(s4fs.specs.UnlPropagatorListener)	org.s4fs.mimsl.constraints.cad.MarketCard.compilePropagating(s4fs.specs.UnlPropagatorListener)	
org.s4fs.mimsl.constraints.cad.MarketCard.leavesatisfying(s4fs.mimsl.core.Lits, org.s4fs.specs.IVecInt)	org.s4fs.mimsl.constraints.cad.MarketCard.leavesatisfying(s4fs.mimsl.core.Lits, org.s4fs.specs.IVecInt)	
org.s4fs.mimsl.constraints.cad.MarketCard.normalized()	org.s4fs.mimsl.constraints.cad.MarketCard.normalized()	
org.s4fs.mimsl.constraints.cad.MarketCard.removing(s4fs.specs.UnlPropagatorListener)	org.s4fs.mimsl.constraints.cad.MarketCard.removing(s4fs.specs.UnlPropagatorListener)	
org.s4fs.mimsl.constraints.cad.BinaryClause.equals(s4fs.lang.Object)	org.s4fs.mimsl.constraints.cad.BinaryClause.equals(s4fs.lang.Object)	
org.s4fs.mimsl.constraints.cad.BinaryClause.isSatisfied()	org.s4fs.mimsl.constraints.cad.BinaryClause.isSatisfied()	
org.s4fs.mimsl.constraints.cad.BinaryClause.activateChangeListener()	org.s4fs.mimsl.constraints.cad.BinaryClause.activateChangeListener()	
org.s4fs.mimsl.constraints.cad.BinaryClause.propagating(s4fs.specs.UnlPropagatorListener, int)	org.s4fs.mimsl.constraints.cad.BinaryClause.propagating(s4fs.specs.UnlPropagatorListener, int)	
org.s4fs.mimsl.constraints.cad.BinaryClause.isConsistent()	org.s4fs.mimsl.constraints.cad.BinaryClause.isConsistent()	
org.s4fs.mimsl.constraints.cad.HTClause.propagating(s4fs.specs.UnlPropagatorListener, int)	org.s4fs.mimsl.constraints.cad.HTClause.propagating(s4fs.specs.UnlPropagatorListener, int)	
org.s4fs.mimsl.constraints.cad.LearnBinaryClause.isActiveByDouble()	org.s4fs.mimsl.constraints.cad.LearnBinaryClause.isActiveByDouble()	
org.s4fs.mimsl.constraints.cad.LearnHTClause.isActiveByDouble()	org.s4fs.mimsl.constraints.cad.LearnHTClause.isActiveByDouble()	
org.s4fs.mimsl.constraints.cad.Lits.valueIsSatisfying()	org.s4fs.mimsl.constraints.cad.Lits.valueIsSatisfying()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.branchNewClause(org.s4fs.specs.UnlPropagatorListener, org.s4fs.mimsl.core.Lits, org.s4fs.specs.IVecInt)	org.s4fs.mimsl.constraints.cad.OriginalHTClause.branchNewClause(org.s4fs.specs.UnlPropagatorListener, org.s4fs.mimsl.core.Lits, org.s4fs.specs.IVecInt)	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.propagating(s4fs.specs.UnlPropagatorListener, int)	org.s4fs.mimsl.constraints.cad.OriginalHTClause.propagating(s4fs.specs.UnlPropagatorListener, int)	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.isSatisfied()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.isSatisfied()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.register()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.register()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.propagating(s4fs.specs.UnlPropagatorListener, int)	org.s4fs.mimsl.constraints.cad.OriginalHTClause.propagating(s4fs.specs.UnlPropagatorListener, int)	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	
org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	org.s4fs.mimsl.constraints.cad.OriginalHTClause.getU()	

Figure 10: Sample Snapshots of Sat4j Reports Table

11.B.4 Validation data collection and Measurement method

In this section, we present the method implemented with each tool to collect the metrics.

DSpot

The output of DSpot was each time added to the initial test suite. In order to preserve integrity toward the control metric, we did all our experimentations on a different local branch, whether on local configuration or remote VM, the strategy was the same. Each time, we were adding the output to initial test suite and not incrementing the test suite. We did this operation manually.

Descartes

Each iteration consisted in analysing the reports and choosing the mutation we want to resolve. This could not have been done without the help of Sat4j leader. Each time, the result of his work resulted in an update of the existing test suite. In the KPI table discussed below, there is a reference to each commit following those updates.

For those experimentations, the control is not fixed like in DSpot experimentations: we compared each mutation score to the previous iteration. This showed us the evolution of the mutation score over time thanks to Descartes tool.

11.C. Validation Results

In this section, we present the metrics computed during this period and our conclusions regarding each tool.

11.C.1 KPIs report

The following tables present KPIs computed from our experimentations metrics. We present for each tool our analysis following this period experimentations.

DSPot:

- Initial control:

Date	Code coverage	Lines covered	Mutation Score	Mutants killed	Reference resource	Amplified classes	Amplified tests
10.12.2018	49%	21,901 of 43,508	*	*	Control measure	N/A	N/A
10.25.2018	49%	21,901 of 43,508	30%	579 / 1876			

Table 52: OW2 K01 & K03 with DSPot (Control)

- Treatments results:

Date	Code coverage	Lines covered	Mutation Score	Mutants killed	Reference resource	Amplified classes	Amplified tests
10.30.2018	49%	21,964 of 43,674	N/A	N/A	https://GitHub.com/STAMP-project/dspot-usecases-output/tree/master/OW2/Sat4j/20181030-1321	4	4
11.02.2018	49%	21,899 of 43,479	31%	585 / 1876	https://GitHub.com/STAMP-project/dspot-usecases-output/tree/master/OW2/Sat4j/20181102-0930	2	2
11.16.2018	49%	21,921 of 43,481	N/A	N/A	https://GitHub.com/STAMP-project/dspot-usecases-output/tree/master/OW2/Sat4j/20181116-1430	5	242

Table 53: OW2 K01 & K03 with DSPot (Treatments)

We used DSPot to generate new amplified tests in order to improve code coverage (related to K01) and mutation score (related to K03). Our different experimentations lead us to different conclusions, depending on the parameters we used as input.

- Execution time and volumetry:

During this period, our experimentations were only based on the compiled sources, using the command line to execute each time. But we do have in mind that those tools have the purpose to be integrated in the CI in the end. So execution time is of paramount importance in the CI process, it should not prevent a new release.

The table below lists execution times depending on the configuration:

Test configuration	# of test classes targeted	1	16	16	16
	Amplifiers	0	0	1	2
Execution time (ms)		7 594	14 1220	760 811	5 736 117
Generated classes		1	4	6	13

Generated tests	1	4	241	1564
-----------------	---	---	-----	------

Table 54: DSpot Results on Sat4j

Note: We do not mention the test criterion in the parameters above because we noticed it did not have an impact on time: for the same configuration above, whether we used JacocoCoverageSelector or PitMutantSelector, times were approximately the same.

On one hand, we can see that starting from 2 amplifiers, execution time start to be a critical constraint. For the record, at this moment, there are 13 amplifiers for DSpot. We did not succeed in testing above 2 amplifiers: starting from 3 amplifiers, there is a OutOfMemoryException. On the other hand, we can also see that volumetry has a direct correlation with execution time.

In our experimentations, volumetry has little impact on code coverage as we can see in table 53. The reason is that most of the new generated tests are in a functional way similar, so they don't improve significantly the code coverage score.

- Issues:

All issues during the period were reported on Github. Most of them were quickly resolved by the development team, they were often related to compilation problems.

Here are some interesting issues:

- Decrease in code coverage: <https://GitHub.com/STAMP-project/dspot/issues/650>

After collecting code coverage from Jacoco, we realized that despite adding new tests, code coverage has decreased.

- PR#631: <https://GitHub.com/STAMP-project/dspot/issues/632>

This pull request was initiated because we realized Descartes engine needed a specific profile on Sat4j. This led to this PR, which led to a new feature in DSpot 1.2.1: "DSpot does not modify your pom anymore but generates a new one containing all the dependencies and configuration (to run PIT/Dcartes)."

The complete list of issues opened for DSpot on Sat4j are available here:

https://GitHub.com/STAMP-project/dspot/issues/created_by/assadOW2

Descartes:

- Initial control:

Date	Result	Lines covered	Mutation Score	Mutants killed	Reference resource	Methods updated
10.25.2018	49%	21,901 of 43,508	30%	571/1876	Initial control	N/A

Table 55: OW2 K01 & K03 with Descartes (Control)

- Treatment results:

Date	Result	Lines covered	Mutation Score	Mutants killed	Reference resource	Methods targeted
11.08.2018	49%	21,902 of 43,508	31%	581 / 1876	https://gitlab.ow2.org/sat4j/sat4j/commit/309b935d53606fcb2f4e00	org.sat4j.tools.ManyCore.addClause(org.sat4j.specs.IVecInt)

					9da82590c993a857d	
11.12.2018	49%	21,921 of 43,534	31%	573 / 1876	https://gitlab.ow2.org/sat4j/sat4j/commit/6c8c8c13d2743462d070c322bb10bffc0b67374	org.sat4j.minisat.core .Heap.increase(int)
11.15.2018	54%	20,051 of 43,641	32%	604 / 1894	https://gitlab.ow2.org/sat4j/sat4j/tree/958ad97a61d958330ea7bdc5ae3ea1e927bec4c2	org.sat4j.minisat.core .Heap.increase(int)

Table 56: OW2 K01 & K03 with Descartes (Treatments)

We used Descartes to improve the quality of existing test suite. This has the direct implication to improve the mutation score (related to K03) and an indirect implication to improve the code coverage (related to K01).

When using this tool, our goal was to reduce on each iteration the number of surviving mutants. In the table below, here is a synthesis of the evolution of number of mutants detected by Descartes:

	10.25.2018	11.08.2018	11.12.2018	11.15.2018
New mutants since previous iteration	17	4	9	28
Mutants killed from previous iteration	8	5	1	5

Table 57: Descartes Results on Sat4j

- 10.25.2018:
 - Among new mutants, many were estimated not relevant to correct, due to the fact that they were corresponding to specific cases, such as a test on a specific problem file.
 - This iteration showed us two interesting methods to fix, `increase()` method in `org.sat4j.minisat.core.Heap` and `createClause()` implementing `org.sat4j.minisat.core.DataStructureFactory` interface.
 - The heap is organized depending on a special heuristic, supposed to be one of the main improvements of Sat4j. What Descartes revealed is that this functionality is not tested. By mutating this method, of course there is no failure because elements in the heap are the same, but we cannot be sure of the added value of Sat4j regarding its heuristic. This is one important test to improve.
 - Concerning `createClause()` method, the interesting point is that this implementation is used by other classes, so we are looking to improve other classes by updating the test case that didn't detect this mutation, class `test BugSAT34`.
- 11.08.2018:
 - The difference between those two iterations is that `createClause()` test was fixed. As expected, mutants related to this method were successfully killed.
 - Among new mutants, we detected one is a false positive, related to one already detected by previous iteration ²⁷.

The reason for this false positive is that timeouts are considered as failures in original test suite. By removing all the code in `onFinishWithAnswer()` method, this causes an infinite loop during execution, which leads to a timeout detected by the original test suite.

 - The other new mutants are related to the same issue, that make them not relevant to investigate further.
- 11.12.2018:

²⁷ <https://GitHub.com/STAMP-project/pitest-Descartes/issues/84>

- Regarding our experimentations on Descartes, there was only one correction related to increase() method in Heap.
- We expected increase() method would be killed but this was not the case. We opened an issue ²⁸

New test cases were not detected by Descartes, and therefore increase() mutation is still surviving.

- 11.15.2018:
 - increase() method is no more surviving, the issue has been fixed.
 - Most of the new detected methods are in the org.sat4j.minisat.core.SolverStats class. Those classes are not relevant to improve, they are simple getters.

K02:

Metric	Measure date	Value
Number of flaky tests identified	07-01-2018	2
Number of flaky tests fixed	11-20-2018	2

The first flaky test fixed during this period is org.sat4j.ModelIteratorTest. Increasing timeout showed that the test was no more flaky ²⁹.

The second flaky test is org.sat4j.minisat.core.TestGroupedTimeoutModelEnumeration. Increasing again timeout fixed the flakiness ³⁰.

During this period of experimentation with DSpot and Descartes, we went through numerous build operations. This gave us the opportunity to stress the SUT and check if the flakiness behaviour of those tests has been fixed or not. In that way, we can assert that STAMP tools helped the Sat4j project to improve the quality of its test suite by reducing the number of flaky tests.

11.C.2 Qualitative Evaluation and Recommendations

In this section, we present for each tool our conclusions and suggest improvements based on our experimentations on Sat4j project.

DSpot

Improvements:

After reviewing the generated source code with the SAT4J leader, we suggested an improvement for DSpot ³¹. Daniel Le Berre noticed that there is a relevant way to improve the generated tests by adding assertions testing null. This was inspired by one of the tests he improved after Descartes execution. This suggestion is currently handled by the DSpot development team.

We would appreciate a global report of DSpot analysis like the one provided by Descartes. For instance, execution time is reported in the console output, but if we do not redirect it to a file or manually copy/paste it after execution, we lose this information because there is no global report.

²⁸ <https://GitHub.com/STAMP-project/pitest-Descartes/issues/85>

²⁹ <https://gitlab.ow2.org/sat4j/sat4j/commit/bd420df52763db3dda96abf78878f4d38f533b2#53669388582d76b618bd4aff56afb0c5df205ba6>

³⁰ <https://gitlab.ow2.org/sat4j/sat4j/commit/6605bc5e6b7d5fcf73456a65bd89955847c8d989#be238dea05625a4fb9b3f51945007e2cf29b88a4>

³¹ <https://GitHub.com/STAMP-project/dspot/issues/648>

Conclusions on DSpot:

We consulted Sat4j leader, Daniel Le Berre, to get his feedback on DSpot. He is interested in test amplification and using DSpot even though he estimates the technology is not yet mature for industrialization. In a first phase, the plan is to configure a distinct branch to test those tools.

There are today two constraints to DSpot usage in a CI: the first is the execution time we already discussed above and the second is the nature of the code generated. Due to automatization, the generated code cannot be read easily by humans, and therefore is not easily maintainable. Moreover, this could lead to concerns about intellectual property.

Descartes

Improvements:

We manually kept track of the differences between the iterations in a table (figure 10). We think this could be a nice feature if Descartes could generate automatically a similar report. For instance, there could be an option in the pom configuration profile that specifies a folder location containing previous iteration. This could be useful in the CI process³², this information could be extracted in order to appear in the logs. This feature can make project leader save time on investigations on the reports.

Conclusions on Descartes:

Although K03 may not reflect it, Descartes results are bringing valuable informations to the Sat4j project. Due to its nature, Sat4j may not be a good candidate for amplification in terms of metrics, nonetheless the tool is relevant in terms of improving test quality, which is the purpose of validation question 2 (VQ2). As far as we are concerned regarding this specific use case, our main concern is more about the complexity to use the tool. As we said, we had some issue configuring the pom configuration profile. In the next period, we are planning to use the tool on another project and this is a point we will be looking at closely.

11.C.3 Answers to Validation Questions

The reports provided by STAMP tools helped to point out some weaknesses in current Sat4j test suite. One intriguing finding was that removing the heart of the solver (the learning scheme) did not break the tests. The solver was simply slower, and the test suite could complete since most of the 1500 functional tests cases are trivial to be executed in a few minutes. Adding a more challenging³³ test case fixed the issue.

That feedback was also useful for STAMP since it pointed out that it was also important to take into account runtime when deciding on the impact of mutations on the test suite.

The interaction between OW2 and the project leader allowed to spot a few suggestions of improvements of the tools.

In conclusion, we can answer to the initial validation questions in the following manner:

- *VQ1 - Can STAMP technologies assist software developers to reach areas of code that are not tested?* During this period, we focused on DSpot to increase code coverage. But as the table 56 shows, Descartes was more helpful to increase the code coverage. It is difficult to improve code coverage with DSpot because SAT4J is a modular library of tools, and thus automatic amplification is more likely to generate non relevant tests. Although Descartes aims more to answer the second validation question, our experimentations lead us to the conclusion that this tool has an indirect impact on code coverage, and thus let us answer in a positive way to VQ1.
- *VQ2 - Can STAMP tools increase the level of confidence about test cases?* We stated above that STAMP tools provided valuable information to improve Sat4j test suite. Metrics do not reflect this improvement, there is no doubt for Sat4j leader, Daniel Le Berre. STAMP tools detect relevant weaknesses in the original test suite, and therefore increase level of confidence about test cases.

32 <https://GitHub.com/STAMP-project/Jenkins-stamp-report-plugin/issues/4>

33 <https://gitlab.ow2.org/sat4j/sat4j/commit/6c8c8c13d2743462d070c322bb10bffc0b67374>

11.C.4 Next Steps for Validation

In the next period, we are planning to work on validation question 3 (VQ3) and validation question 4 (VQ4). To answer those questions, we will work with CAMP tool on DocDoku project and with Botsing tool on Sat4j project. We present below our roadmap for the next three months.

Task	Date											
	December 2018				January 2019				February 2019			
CAMP: Configuration on DocDoku												
CAMP: Experimentations on DocDoku												
CAMP: Results analysis												
Botsing: Configuration on Sat4j												
Botsing: Experimentations on Sat4j												
Botsing: Results analysis												

Table 58: OW2 Roadmap for VQ3 and VQ4

The final step after those experimentations is to do another round on two different projects in our code base. The identified candidates are AuthzForce, Bonita, Lutece and SeedStack. We have already contacted Bonita's team and they are looking forward to working on STAMP tools. We plan to do the same with the three other projects.

12. Threats to Validity

In this section, we report and discuss potential threats to the validation validity due to the choice of the validation design. The following paragraphs elicit these threats and introduce mitigation strategies. We identify the following threats to validity:

- T1: Experimenters background and skills
- T2: Adequacy of use cases (size, complexity, completeness of evaluation)
- T3: Tool compatibility/interoperability with use case baseline (or standards)

T1 - Experimenters background and skills

The experimenters' background and skills may be an issue to test the tools. For example, DSpot and Descartes require an understanding of mutation testing to know how to optimally use the tools and how to process and interpret their output. Descartes in particular requires in-depth knowledge of the code being tested to take full advantage of the tool.

Mitigation strategy: In order to reduce the tool usage risk, we limited our development to Java, a language

all use case providers are familiar with. Moreover, we have organised workshops, both online and physical, between the use case providers and the tool builders, to create a shared understanding of how the tools work and what is further needed to make them easier to work with. In addition, we have used an issue tracker to document issues and support the assistance and problem solving. This has led to improvements in the tooling, and it facilitated the use of the tool by other partners.

T2 - Adequacy of use cases (size, complexity, completeness of evaluation)

The adequacy of the use cases for validating the tools could be a threat. The research tools are ambitious in pushing forward the state-of-the-art of automatic testing. Yet, they might fail in the light of too much complexity, or they might take too long to execute. In similar vein, a use case with low code coverage limits the relevance of DSpot and Descartes, because these tools rely on existing tests. While Botsing does not require pre-existing tests, its search-based strategy does suffer from projects that have a high cyclomatic complexity.

Mitigation strategy: The diversity of the use cases from the use case providers is an excellent solution to make sure that the tools can be evaluated. Furthermore, the diversity of use cases also provides an interesting perspective on which tools can be relevant and efficient in which particular context.

T3 - Tool compatibility/interoperability with use case baseline or standards

The tools' compatibility and interoperability with the use case standards may be troublesome. The diversity of use cases is a strength because it helps explore many different possibilities for the tools. However, because of this diversity, it is challenging to make the STAMP tools ready to use in all these different contexts and specific environments.

Mitigation strategy: We opted for Java as the common denominator between use cases. While this does reduce the scope somewhat, it does help with interoperability. Moreover, the tools make use of standard Java build technology (Maven, Gradle) to facilitate operability. We also maintain the constant conversations between tool developers and use case providers to address this challenge.

13. Validation Summary

This section provides a cross-use case summary of the results. It presents a visual summary of the global results and consolidates some recommendations for each of the STAMP tools that have been evaluated.

13.A Validation Questions Synthesis

This section summarizes answers provided by use cases to validation questions. These questions are introduced in section 6, they reflect the main expectations from the point of view of professional software developers, vendors and systems integrators.

Answers to the validation questions are not homogeneous, they may differ from one use case to another because of the differences in the use cases. In the following paragraphs we focus on results that may be shared by all use cases. The specifics are detailed in the "Answers to Validation Questions" section in each use case.

VQ1 - Can the STAMP tools assist software developers to reach areas of code that are not tested?

The objective of both DSpot and Botsing is to generate new tests, and code coverage is a way to validate this. At this stage of the development of the tools and the progress of the use cases, the results obtained by industrial partners with regard to this validation question are limited, yet generally positively oriented. Test case amplification has been observed from DSpot. Positive results are typically obtained indirectly by proceeding to test and code adjustment after applying Descartes and DSpot and measuring again. All use case partners except one report improvements albeit of a limited scope. While not really appearing in the metrics, use case partners find that STAMP help increase the quality of the tests.

VQ2 - Can STAMP tools increase the level of confidence about test cases?

There is a definite positive consensus among use case partners with regard to this question even if it is not always reflected so much in the metrics. STAMP helps to identify weaknesses in test suites. Descartes is

particularly useful here. Use case partners found Descartes helps to efficiently compute mutation scores and significantly reduce the number of pseudo and partially tested methods. Use case partners find Descartes analysis and report useful to increase the level of confidence on test cases, once the required test/code refactorings that can be derived from the report are applied. Manual intervention remains required if only to sort out methods that were voluntarily, for any reason, kept out of testing scope, but relevant weaknesses are nevertheless identified thanks to STAMP. And this is a well appreciated result, as one use case partner puts it: “we now know what is tested”. A related impact of the STAMP tools is that they help to generally improve testing culture, behavior and processes thus contributing to a higher level of confidence in test cases.

VQ3 - Can STAMP tools increase developers confidence in running the SUT under various environments?

This question essentially relates to CAMP. There is another positive consensus here. Use case partners report that, since it greatly simplifies the generation of alternative configurations, CAMP increases the level of confidence with a variety of execution environments. STAMP can help to identify optimal configurations or testing applications with different workloads that require different configurations. Two use case partners have been able to generate and test up to ten and more different configurations. A key business advantage is that being able to test different configurations contributes to enhancing the quality of a software as perceived by end-users.

VQ4 - Can STAMP tools speed up the test development process?

The most significant positive result with regard to this question at this stage is the huge reduction in time it takes for developers to test an application in different configuration thanks to CAMP. CAMP facilitates testing in different environment. In one use case, what might have taken hours and great difficulty is now achieved easily and in five minutes. CAMP will create a real disruption with CI/CD approaches when/if it can explore the configuration space looking for alternative valid configurations beyond what is available in Docker and provide recommendations for optimal configurations. Use case partners have also great expectations with DSpot and Botsing the tools that automatically generate new test cases. DSpot must still progress in enhancing its test generation capacity and reducing its execution time. Botsing helps understand why code fail but it will be most effective when it can automatically generate regression tests that can be integrated in test suites.

13.B. KPI Synthesis

13.B.1 K01 - More execution Paths

Objective: 40% reduction in the non-covered code since baseline

K01	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon		√		
ATOS		√		
Tellu		√		
XWiki		√		
OW2		√		

Table 59: K01 Progress Overview

Atos obtained a total of 9.47% combining Descartes and DSpot treatments, noticing however that. Descartes has a negative effect on coverage (-1.2%). Tellu found that fixing Descartes issues gave 1.7% reduction for TelluLib and 0.5% for Core. XWiki reached 30.47% of target. For OW2, KPIs are related to Sat4j project, one out of four different projects constituting the use case. It is difficult to improve code coverage with DSpot because Sat4j is a modular library of tools, and thus automatic amplification is more likely to generate non relevant tests; better results are expected with the other projects.

13.B.2 K02 - More flaky tests identified & handled

Objective: Difference with baseline > 20 %

K02	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon		√		
ATOS		√		
Tellu				√
XWiki				√
OW2		√		

Table 60: K02 Progress Overview

Activeeon's measurements did not report new flaky test. Atos tagged all candidate tests for flakiness as not flaky because the reasons for alternative failure/pass were identified, most of cases due to SUT and environment causes. With no baseline, at Tellu the 3 flaky tests of a suite of 6 tests were fixed. XWiki reports an outstanding result of +368.45% of objective. Due to its nature, OW2 Sat4j is less likely to have flaky tests but more flaky tests are expected to be detected when working on a UI project.

13.B.3 K03 - Better test quality

Objective: 20% increase in mutation score

K03	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon		√		
ATOS				√
Tellu			√	
XWiki		√		
OW2		√		

Table 61: K03 Progress Overview

Activeeon reports a 1% increase in Catalog project. Atos measured a 25,64% increment in mutation score, 33.33% reduction on the number of pseudo tested methods and 97.49% reduction in the number of partial tested methods. Tellu found 4.2% increase in TelluLib, 14% increase in Core, 0% in FilterStore (average weighted by LOC is 10.4% increase, as Core is by far the biggest of the projects). XWiki measured an average of 6.5% of mutation score increase on affected modules. Descartes results are bringing valuable informations to the OW2 Sat4j project. Due to its nature, Sat4j may not be a good candidate for amplification in terms of metrics, nonetheless the tool is relevant in terms of improving code quality.

13.B.4 K04 - More unique invocation traces

Objective: 40% more unique invocation traces (microservices) or 20% coverage increase (monolithic)

K04	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon	√			
ATOS			√	
Tellu		√		
XWiki		√		
OW2	√			

Table 62: K04 Progress Overview

Atos reports an increment of 13.92% on the number of unique external inter-service invocation paths (service base monolithic system). For Tellu, the result is 0% so far as they are only changing infrastructure components. If in the future they create multiple versions of their own services they expect this number to increase. At XWiki the result is also a 0% coverage increase so far and unlikely to change much due to the architecture of the application. Activeeon and OW2 did not experiment with CAMP during this period.

13.B.5 K05 - System specific bugs

Objective: 30% more such bugs discovered

K05	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon	√			
ATOS				√
Tellu			√	
XWiki		√		
OW2	√			

Table 63: K05 Progress Overview

Atos detected 4 new configuration specific system failures; percentage is set to >100% by convention because previously this kind of system failures (no bugs in code, but in runtime behaviour due to configuration) were not managed. At Tellu with a baseline of 10, 2 new detections account for a 20% improvement. XWiki is at 7.14% at the moment; lots of system-specific bugs are discovered by the community and thus the baseline value of 56 configuration-related bugs found in a 1.5 years is a large number, and increasing it by 30% is challenging. Activeeon and OW2 did not experiment with CAMP during this period.

13.B.6 K06 - More configuration/faster tests

Objective: 50% increase

K06	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon	√			
ATOS				√
Tellu				√
XWiki				√
OW2	√			

Table 64: K06 Progress Overview

Atos obtained up to 10 new configurations generated compared to one default configuration that was managed before. Tellu went from 1 to 3 configurations and XWiki had 1 config before and now have 26, so that's +2600%! Activeeon and OW2 did not experiment with CAMP during this period.

13.B.7 K07 - (this indicator was deprecated)

13.B.9 K08 - More crash replicating tests

Objective: 70% increase of crash replications with test cases

K08	Not measured	<50% achieved	>50% achieved	≥100% achieved
Activeeon		√		
ATOS		√		
Tellu		√		
XWiki			√	
OW2	√			

Table 65: K08 Progress Overview

Activeeon obtained 1 test replicated. But Atos obtained 0/4: no crash replicating test were generated from 4 cases treated by Botsing. At Tellu there were none as baseline and no new crash were replicated. XWiki however report a current increase of 60%. OW2 did not work with Botsing during this period.

13.C. Quality Assessment

This section provides an overview of the tools quality assessment provided by the use cases. The first part is a reminder of the assessment criteria and the second details the results tool by tool as they have been reported by the use cases.

Overall quality model

We have defined a quality model for STAMP tools, in deliverable 5.2 “Validation roadmap and framework”. This model is inspired by ISO/IEC 25010:2011, in this standard the product quality of a software system is evaluated according to eight characteristics with thirty-one sub-characteristics. Not all these characteristics are relevant to STAMP tools and some cannot be evaluated at the actual level of maturity of the tools. We selected six characteristics with ten sub-characteristics for each use case partners to use to evaluate the STAMP tools. The quality model is structured as follows.

	Sub-Characteristics	Definition
Functional suitability	Functional Completeness	Degree to which the set of functions covers all the specified tasks and user objectives.
	Functional Correctness	Degree to which the functions provides the correct results with the needed degree of precision.
	Functional Appropriateness	Functional degree to which the functions facilitate the accomplishment of specified tasks and objectives
Compatibility	Co-existence	Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
Performance Efficiency	Time-behavior	Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements
Usability	Appropriateness recognisability	Degree to which users can recognize whether a product or system is appropriate for their needs.
	Learnability	Degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations
	Operability	Degree to which a product or system is easy to operate, control and appropriate to use.
Reliability	Maturity	Degree to which a system, product or component meets needs for reliability under normal operation.
Portability	Installability	Degree of effectiveness and efficiency in which a product or system can be successfully installed and/or uninstalled in a specified environment.

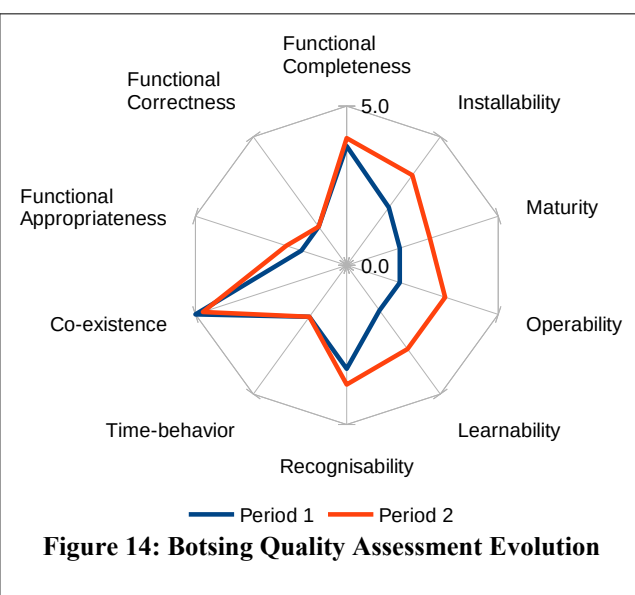
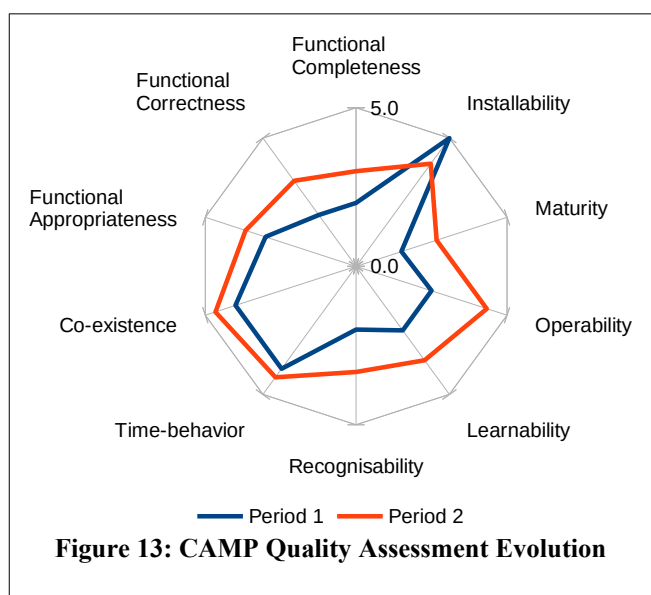
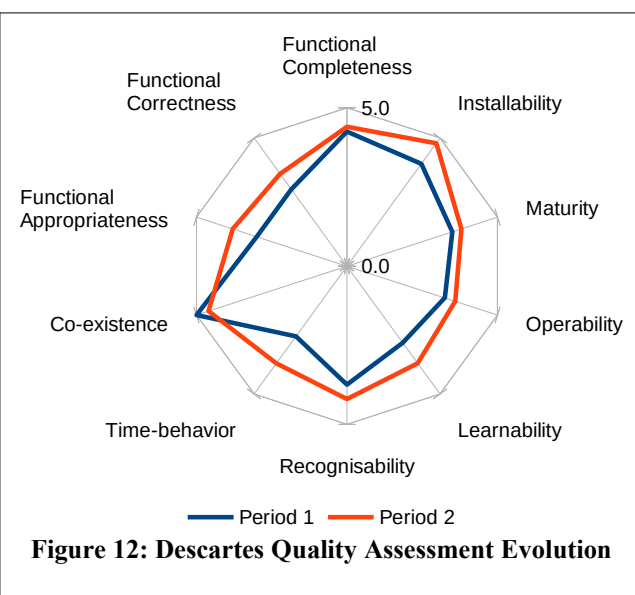
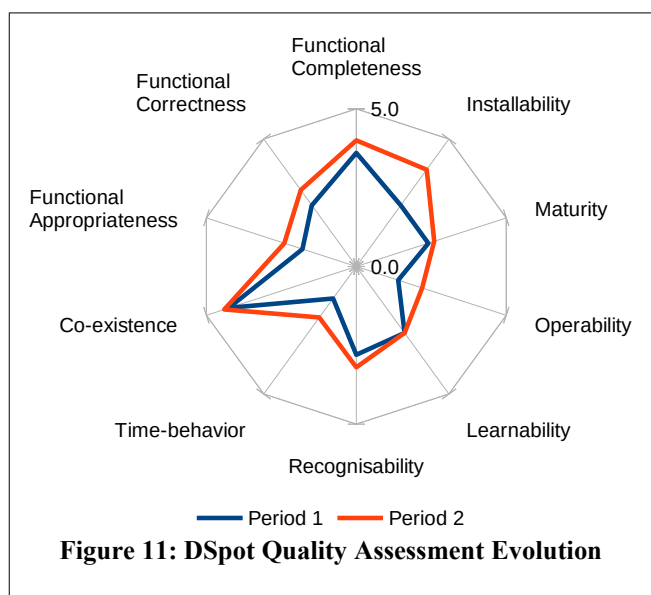
Table 66: Global Quality Model Applied by the Use Cases

Each criteria is to be evaluated on a 1 to 5 scale. One being the worst notation and five being the best. The evaluation is provided by users who experimented with the STAMP tools in their use cases. This qualitative assessment methodology is based on subjective notations as opposed to the KPI evaluations that are purely quantitative. However, perception based on experience as reported by the use case partners is representative of real-life usage and is just as a critical factor in project quality as pure performance

measurement. The following section provides the perception provided by use case tool by tool.

Quality assessment of the tools

The four diagrams below show the evolution of the quality notations given to the tools by the use case partners. In the above diagram, we present the result of the average quality evaluation conducted by use case. The DSpot and Descartes tools were evaluated by all partners and Botsing by two partners. CAMP's evaluation is postponed to the next period. The results demonstrate a very good (>4) co-existence through their integration into existing testing tools, good (>3.5) installability and functional completeness, and an acceptable (>2.5) appropriateness. Overall Descartes by use case partners with the highest rating by use case partners. Time-behavior and Operability are the criteria where progress is much needed for DSpot.



DSpot

DSpot provides useful insights to improve the test suite for a code base. The tool has great potential since automated test generation is of huge interest.

Significant progress has been accomplished on DSpot since the earlier version reported in D55. However the tool is not yet ready for industrialization. Thus most validation comments focus on industrialization challenges and guidance for improvements. For example, it is important that DSpot produces a report that summarizes key metrics about the test exploration and generation process.

The execution time of DSpot has drastically decreased since previous release. Yet, it remains the main area of expected improvement. Use case partners report execution time measured in hours instead of minutes. Using DSpot is still time consuming especially when combining several amplifiers. Other appreciated improvements would cover: support Windows and improve the understandability of generated test code.

The figure below reflects the average perception of DSpot by the use case partners. It can be observed in the figure that the overall users' impression has improved for all quality properties during this period, compared to the previous one.

Descartes

Descartes is recognized by all use case partners as a useful and mature tool. It helps fix issues related to pseudo and partially tested methods and thus concretely contributes to improve the quality of test suites. The benefits brought by Descartes are real and significant even if they are not reflected in the mutation score and code coverage metrics. Descartes works well, it is easy to use in command line and in the CI, and it produces good reports that are clear to understand. Use partners appreciate the tool integrations in Eclipse, Maven and Gradle.

The most important feature request from use case partners is the possibility to compare two reports so as to help identify fixed, remaining and new issues between two runs. At this stage, using Descartes involves important manual efforts to analyse its results and improve the tests.

All use case partners acknowledge that Descartes requires intimate knowledge of the code being tested to be fully beneficial. It should be used by the developers themselves, those who understand the specifics of the methods being tested. From that, a best practice has been suggested which is to use Descartes at development time to immediately challenge a test that has just been written.

The figure below reflects the average perception of Descartes by the use case partners. It is worthy to note the overall users' perception improvement for all the quality factors in this period.

CAMP

CAMP has been significantly improved compared to the version evaluated in D55. It has also seen a new add-on called CAMP/TestContainers which can be used to execute various configurations directly from Java test cases. The tool is easy to install and use. Its function, focused on configuration testing has huge potential. CAMP. Use case partners can demonstrate that CAMP helps test a variety of configurations and find issues. CAMP can actually generate valid configurations and simplifies the generation of new configurations, for example for container-based deployments. This is a key benefit. CAMP can be very useful to validate that a software can run correctly using different versions of back-end services or help explore new configuration environments.

A key challenge as of today is to differentiate from the conventional CI/CD approach and, for instance to automate the instantiation of new configurations. CAMP would create a real disruption with existing CI/CD approaches when/if it can explore the configuration space looking for alternative valid configurations beyond what is available in Docker and provide recommendations for optimal configurations.

The figure below reflects the average perception of CAMP by the use case partners. Note that overall users' perception in this period has improved for several quality properties.

Botsing

Botsing is an in-depth refactoring of the tool, EvoCrash, reported in D55. Compared to EvoCrash, Botsing is more understandable with improved usability and accessibility. The documentation is correct and it runs as a test invoked from Maven or Eclipse.

Botsing is able to generate test cases that reproduce a crash that appears in a log file. This feature is well



appreciated by developers. However, to run Botsing successfully currently seems to require specific conditions. For instance one use case partner found that stack traces collected over one month were not relevant for replication. Other limiting conditions currently include network access restrictions and some necessary symmetry between incoming data and test database that are seldom encountered in actual use cases. Use case partners also would appreciate if the regression tests generated by Botsing were in a form that could be integrated in a test suite.

The figure below reflects the average perception of Botsing by the use case partners (three partners). The overall users' perception on the tool has improved in this period for six out of ten of the quality properties.

14. Conclusion

In his second period, STAMP use case partners and development teams have made a real effort to demonstrate significant progress in the evaluation of the STAMP tools. All use case partners have developed experimentation in relation to the specifics of their business processes and all STAMP tools have been experimented in real life conditions.

All use case partners at their own pace have been able to experiment with the STAMP tools, conduct initial measurements and iterate so as to produce KPIs and analyse results. Results vary greatly depending on the experimentation environment and the specifics of the target software.

The STAMP tools have shown their potential albeit at different degrees given the difference in their development stages and maturity. Use case partners are in permanent communication with development teams either directly by mail or indirectly via the issue trackers of their development platforms.. They have been able to share progress and report valuable and pragmatic (result-oriented) feedbacks.

All use case now have a pretty clear roadmap for completing their evaluation tasks, an important aspect of which is the alignment with the specifics of their own business processes.