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

Title	WP5 – D5.7 – Use Cases Validation Report V3
Date	November 29, 2019
Writers	Mael Audren de Kerdrel, Activeeon Mohamed Boussaa, Activeeon Lars Thomas Boye, TellU Pierre-Yves Gibello, OW2 Jesús Gorroñogoitia, ATOS Vincent Massol, XWiki SAS Fernando Mendez, ATOS Assad Montasser, OW2 Pedro Velho, Activeeon
Reviewers	Benoit Baudry, KTH Mohamed Boussaa, Activeeon Caroline Landry, INRIA Andy Zaidman, TUDelft
Online version	https://github.com/STAMP-project/docs-forum/blob/master/docs/d57_uc_validation_report-final.pdf

Table of Content

1. Executive Summary 7
2. Revision History 10
3. Objectives 10
4. Introduction 10
5. References 11
6. Validation Questions 11
7. Activeeon Use Case Validation 14
7.A. Use Case Description 14
7.A.1 Target Software 14
7.A.2 Experimentation Environment 14
7.A.3 Expected Improvements 14
7.A.4 Business Relevance 14
7.B Validation Experimental Method 14
7.B.1 Validation Treatments 14
7.B.2 Validation Target Objects and Tasks 15
7.B.3 Validation Method 16
7.B.4 Validation Data Collection and Measurement Method 20
7.C Validation Results 23
7.C.1 KPIs Report 23
7.C.2 Qualitative Evaluation and Recommendations 30
7.C.3 Answers to Validation Questions 31
7.D Global Takeaway 32
8. Atos Use Case Validation 33
8.A. Use Case Description 33
8.A.1 Target Software 33
8.A.2 Experimentation Environment 33
8.A.3 Expected Improvements 33
8.A.4 Business relevance 34
8.B. Validation Experimental Method 34
8.B.1 Validation Treatments 34
8.B.2 Validation Target Objects and Tasks 35
8.B.3 Validation Method 45
8.B.4 Validation Data Collection and Measurement Method 46
8.C Validation Results 47

8.C.1 KPIs report 47
8.C.2 Qualitative Evaluation and Recommendations 59
8.C.3 Answer to Validation Questions 61
8.D Global Takeaway 63
9. Tellu Use Case Validation 64
9.A. Use Case Description 64
9.A.1 Target Software 64
9.A.2 Experimentation Environment 65
9.A.3 Expected Improvements 65
9.A.4 Business Relevance 65
9.B. Validation Experimental Method 66
9.B.1 Validation Treatments 66
9. B.2 Validation Target Objects and Tasks 67
9.B.3 Validation Method 68
9.B.4 Validation Data Collection and Measurement Method 76
9.C. Validation Results 76
9.C.1 KPIs Report 76
9.C.2 Qualitative Evaluation and Recommendations 98
9.C.3 Answers to Validation Questions 100
9.D Global Takeaway 101
10. XWiki Use Case Validation 102
10.A. Use Case Description 102
10.A.1 Target Software 102
10.A.2 Experimentation Environment 102
10.A.3 Expected Improvements 102
10.A.4 Business Relevance 102
10.B. Validation Experimental Method 102
10.B.1 Validation Treatments 102
10.B.2 Validation Target Objects and Tasks 104
10.B.3 Validation Method 105
10.B.4 Validation data collection and Measurement method 106
10.C. Validation Results 106
10.C.1 KPIs report 106
10.C.2 Qualitative evaluation and Recommendations 121
10.C.3 Answer to Validation Questions 122
10.D Global Takeaway 123

11. OW2 Use Case Validation 125
11.A. Use Case Description 125
11.A.1 Target Software 125
11.A.2 Experimentation environment 125
11.A.3 Expected Improvements 126
11.A.4 Business Relevance 127
11.B. Validation Experimental Method 127
11.B.1 Validation Treatments 127
11.B.2 Validation Target Objects and Tasks 128
11.B.3 Validation Method 129
11.B.4 Validation data collection and Measurement method 129
11.C. Validation Results 130
11.C.1 KPIs report 130
11.C.2 Qualitative Evaluation and Recommendations 136
11.C.3 Answers to Validation Questions 138
11.D Global takeaway 139
12. Threats to Validity 139
13. Validation Summary 139
13.A Validation Questions Synthesis 140
13.A Validation Questions Synthesis 141
13.B. KPI Synthesis 141
13.C. Quality Assessment 145
14. Conclusion 149
15. Appendix. Comparative STAMP Workshop 150

List of Tables

Table 1 Use Case Experimentations during Period 2 (X-√) and Period 3 (√) 8
Table 2 KPI objectives fulfillment by use case 9
Table 3 Experimental treatments applied in ActiveEon case 14
Table 4 Summary of KPIs for ActiveEon case 23
Table 5 First Catalog Project Scores 26
Table 6 Botsing reproduction generation results 30
Table 7 Control Tasks for Atos Validation Experimentation 34
Table 8 Treatment Tasks for Atos Validation Experimentation 35
Table 9 Atos code base Repositories for Use Cases (SUPERSEDE IF and CityGO CityDash) 35
Table 10 Atos Experiments for KPIs. Sequence of Experimentation Tasks 45

Table 11 Summary of Atos KPI measurements 47
Table 12 K01: code coverage for IF use case; efficiency of STAMP tools 48
Table 13 K02 code coverage metrics for IF use case 50
Table 14 K03 mutation score metrics for IF use case 50
Table 15 K03 mutation score and detected issues for IF use case 51
Table 16 Results of stress test execution for CityDash in each amplified configuration 53
Table 17 Results of functional test execution for CityDash for all amplified configurations 54
Table 18 K04 More unique invocation traces: metrics for CityDash Use Case in performance scenario 54
Table 19 K04 More unique invocation traces: metrics for CityDash Use Case in functional scenario 55
Table 20 K05 System specific bugs: metrics for CityDash Use Case (Performance scenario) 55
Table 21 K05 System specific bugs: metrics for CityDash Use Case (Functional Scenario) 56
Table 22 K06 More Configuration/Faster tests: metrics for CityDash Use Case (Performance scenario) 56
Table 23 K06 More Configuration/Faster tests: metrics for CityDash Use Case (Functional scenario) 56
Table 24 K08 Generated crash-replicating test cases: metrics for IF Use Case 57
Table 25 K09 Amplified production level test cases: metrics for IF Use Case 58
Table 26 Tellu use case source code metrics 65
Table 27 Validation treatment overview 66
Table 28 DSpot experiments on ModelTest class 70
Table 29 DSpot experiments on other TelluLib test classes 70
Table 30 DSpot experiments on Core 71
Table 31 Example of test results 73
Table 32 Summary of Tellu KPI measurements 77
Table 33 Summary of Descartes validation results 78
Table 34 Experiment ModelTest-Selectors - test of different selectors on TelluLib ModelTest 78
Table 35 Experiment ModelTest-Pit-Amplifiers - test of different amplifiers on TelluLib ModelTest 78
Table 36 Experiment ModelTest-Pit-MultiAmplifiers - test of combinations of amplifiers on TelluLib ModelTest 79
Table 37 Experiments on other test classes of TelluLib 80
Table 38 Experiments on project Core 80
Table 39 Final KPI results for DSpot 81
Table 40 Comparison of Descartes and DSpot coverage improvements 82
Table 41 RAMP coverage results 82
Table 42 RAMP generated tests compared to manually written tests 83
Table 43 Table 9.18: Global coverage in TelluLib throughout STAMP 83
Table 44 Global coverage in Core throughout STAMP 84
Table 45 Global coverage in FilterStore throughout STAMP 84
Table 46 Combined coverage results 84
Table 47 Descartes mutation score result summary 85
Table 48 DSpot mutation score results 85

Table 49 Comparison of Descartes and DSpot test quality results 86
Table 50 RAMP mutation score results 86
Table 51 Global mutation score in TelluLib throughout STAMP 87
Table 52 Global mutation score in Core throughout STAMP 87
Table 53 Global mutation score in FilterStore throughout STAMP 88
Table 54 Combined mutation score results 88
Table 55 Test execution times for a selection of generated configurations 89
Table 56 Test results for configuration 12 89
Table 57 Average deviation of each alarm from its sequence 92
Table 58 Botsing replication of introduced errors 93
Table 59 RAMP test generation results for TelluLib classes 96
Table 60 RAMP test generation results for Core classes 97
Table 61 Summary of K09 results 98
Table 62 Controls and Treatments for KPIs for XWiki Software 103
Table 63 Validation Target Objects for XWiki Software 105
Table 64 Validation/Measurements Methods for XWiki 105
Table 65 Summary of XWiki KPI measurements 107
Table 66 K01 Metric Measures for XWiki 107
Table 67 Coverage and Mutation Score Contributions by Descartes Strategy 110
Table 68 Coverage and Mutation Score Contributions by DSpot Strategy 111
Table 69 Flaky Test Measurements 112
Table 70 Floating Configurations Implemented for “latest” 113
Table 71 Floating Configurations Implemented for “all” 114
Table 72 Floating Configurations Implemented for “unsupported” 115
Table 73 Docker-based tests 115
Table 74 Unique Traces Metrics 116
Table 75 System-Specific Bugs Found Thanks to Configuration Testing 117
Table 76 Number of New Configurations Tested 117
Table 77 Percentage of Crash-Replicating Test Cases 118
Table 78 Percentage of generated tests 119
Table 79 KPI measurements 127
Table 80 OW2 summary of validation tasks 128
Table 81 Summary of KPI metrics 131
Table 82 Qualitative evaluation of STAMP tools 136
Table 83 K01 Progress Overview 141
Table 84 K02 Progress Overview 142
Table 85 K03 Progress Overview 142
Table 86 K04 Progress Overview 143

Table 87 K05 Progress Overview 143
Table 88 K06 Progress Overview 144
Table 89 K08 Progress Overview 144
Table 90 K09 Progress Overview 145
Table 91 Task 1 results for each participant 151
Table 92 Task 2 results for each participant 152
Table 93 Task 3 results for each participant 152
Table 94 Task 4 results for each participant 154

List of Figures

Figure 1 CAMP input file to define the replacement strategies and configuration settings 17
Figure 2 Seven test classes automatically generated by DSpot amplification methods. 24
Figure 3 An example test case from DSpot automatic generated tests. 25
Figure 4 PWS configurations generated by CAMP to support multiple DBs. 28
Figure 5 Jenkins jobs/pipelines for Atos Validation Experimentation 36
Figure 6 STAMP Descartes Reports for Supersede IF 37
Figure 7 STAMP DSpot Reports for Supersede IF 38
Figure 8 Configurable STAMP pipeline for Supersede IF (above) and execution instances (below) 39
Figure 9 Case 1- Apache docker compose 41
Figure 10 Nginx service 43
Figure 11 Web service 43
Figure 12 Selenium Grid 44
Figure 13 CAMP pipeline 44
Figure 14 Mutation Score evolution for IF use case reported by STAMP Jenkins plugin 51
Figure 15 Results of stress test execution for CityDash in each amplified configuration 52
Figure 16 Execution time over three test runs 91
Figure 17 Number of alarms created by each pod per 100 alarms 92
Figure 18 Tool chain 126
Figure 19 DSpot Quality Assessment Evolution Figure 20 Descartes Quality Assessment Evolution 146
Figure 21 CAMP Quality Assessment Evolution Figure 22 Botsing/RAMP Quality Assessment Evolution 146

1. Executive Summary

The STAMP test amplification technologies developed by research partners have been validated by industry partners in use cases representative of different business-oriented implementations. The use cases are provided by the 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 all use case partners as shown in Table 1 below (unlike the validation period 2, where some use cases could not experiment all them). Use case providers have applied the STAMP tools, namely DSpot, Descartes, CAMP, Botsing and RAMP, to their target software, according to the "Validation Roadmap and framework V2" described in deliverable D5.4, in close interaction with the tool partners.

Table 1 Use Case Experimentations during Period 2 (X-√) and Period 3 (√)

	Descartes	DSpot	CAMP	Botsing/RAMP
Activeeon	√-√	X-√	X-√	√-√
Atos	√-√	√-√	√-√	√-√
Tellu	√-√	√-√	√-√	√-√
XWiki	√-√	√-√	√-√	√-√
OW2	√-√	√-√	X-√	X-√

All the use case partners have experimented with 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 cases 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. The use cases have spent special effort to measure quantitative results obtained with the STAMP tools. Moreover, each use case conducted several iterations during their experiments to understand the fitness of the tools for their own business purposes and to explore how the tools can be adapted to their needs.

Each use case follows a common structure as described below:

- **A- Description**: this part re-uses some elements already presented in D5.6, it introduces (in the cases that D5.6 description needs to be updated) 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, how the baseline ("control" values) is established, and how the experimental data was sampled or collected during experimentation.
- **C- Validation results**: the third part presents the main validation findings, the results associated with the KPIs: variable measurements, the answers to Validation Questions and the quality assessment.
- **D- Global takeaway**: this last part concludes the use case validation report by outlining the main business benefits acquired by the industrial partner with the adoption of the STAMP tools.

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 last period about testing the STAMP tools can be summarized as follows:

- **All KPIs' objectives have been successfully achieved**. In all KPIs (with the exception of K08) at least one UC has achieved the objective. Table 2 below summarises the fulfillment of KPI objectives per use case.

Table 2 KPI objectives fulfillment by use case

	AEon	Atos	Tellu	XWiki	OW2
K01	√	√	X	X	√
K02	√	X	√	√	X
K03	√	√	√	X	√
K04	√	X	√	√	√
K05	√	√	X	X	X
K06	√	√	√	√	√
K08	X	X	X	X	X
K09	√	√	√	√	√

- **Answers to the validation questions are positive**. There is full consensus among industrial partners that STAMP tools positively i) assist software developers to reach areas of code that are not tested, ii) increase the level of confidence about test cases, iii) increase developers' confidence in running SUT under various environments, and iv) speed up the test development process.

- **All use case partners report improvements in their test suites and in the confidence of their own tests** after using the STAMP tools. These improvements are derived from the actual results provided by the STAMP tools' ability to: i) improve the test suites' quality and their associated mutation score, ii) increase the test base and associated code coverage, iii) enlarge the number of test configurations and improve the ability to detect environmental issues, and iv) manage runtime exceptions on test cases.

- **Development operations have clear feedback from the use cases**. All STAMP tools have been continuously tested by industry partners with real life software, assessing the TRL6. This period has seen a significant increase in the interactions 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 these interactions, all use cases had a clear roadmap for the completion of their experiments.

2. Revision History

Date	Version	Author(s)	Comments
5-Sept-19	0.1	Jesús Gorroñogoitia, ATOS	ToC
8-Nov-19	0.2	All Contributors	Validation report for each UC Validation questions
14-Nov-19	0.3	All Contributors and reviewers	First peer-review.
22-Nov-19	0.4	All Contributors	Addressing peer-review recommendations Consolidated report
26-Nov-19	0.5	All Contributors and reviewers	Second peer-review. Addressing peer-review recommendations
29-Nov-19	1.0	All Contributors	Final version
29-Nov-19	1.1	Caroline Landry, INRIA	Quality check

3. Objectives

This document reports on the validation activities of the STAMP tools conducted by the industrial use cases during the third year of the STAMP project. These activities adhere to the validation guidelines defined in D5.4. The objectives of the validation activities are as follows:

- Assess the functionality and usability of the STAMP techniques and toolset.
- Empirical assess on the practical and industrial use of the current STAMP toolset and technologies.
- 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 (DoA). The primary validation objective is to: Validate the relevance and effectiveness of amplification on the five 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 DoA. The technology KPIs were refined and detailed in deliverable D5.3.

4. Introduction

This document reports on the third period of the STAMP project use cases conducted from M25 to M36. The second period ran from M19 to M24. The validation activities 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; deliverable D5.6 provided updated results obtained during the M19-M24 period; this derivable concludes these series by drawing the final industrial validation results.

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 providing 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 main sections of the deliverable are dedicated to the evaluation report of each use case. Other 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.

As this deliverable is an evolution of the previous one, D5.6, it focuses on reporting updates in the validation activities conducted by the industrial use cases, referencing D5.6 for any material that was already presented therein, hence.

5. References

D3.3: Prototype of amplification tool for common and anomaly behaviors
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
D5.6: Use Cases Validation Report V2
D5.7: Use Cases Validation Report V3

6. Validation Questions

In this section, we introduce the Validation Questions (VQs) we have defined to guide the validation activity. The validation questions are like “research questions” except that they are more adapted to the industrial context of the use cases. These validation questions were already introduced in D5.6. We keep them here (so we do not just refer to D5.6) because i) we have slightly updated them to introduce the influence of the new STAMP tools related to K09 and ii) since the entire industrial validation seeks to answer these questions, including the description of these questions within this document facilitates the reader to refer to them.

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 the software behavior is of paramount importance. Yet, when a program is already well tested, the real challenge is to reduce the amount of untested code[1]. It is very challenging to write test cases that reach the remaining 10 or 20% of code that is not covered by existing tests. Software engineers involved in the STAMP project argue that the main goal of measuring the code coverage is to find unreached code (e.g. untested base code). The goal of this validation question is to assess the capacity of STAMP tools to assist developers to write new test cases that cover unreached code.

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, the metric associated with this K04 indicates how configuration test amplification can help to test more paths through the system under test (SUT). This metric is more use case dependent, with the relevance and methodology varying in accordance with the 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 that are not reached before.
- K09 - More production level test cases: STAMP tools may be able to create tests after observing the runtime behaviors. These new tests may also traverse additional uncovered code paths.

This VQ1 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 (or SUT configuration). This leads to a loss of confidence in tests, having some 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: A flaky test is a form of false positive, where the test sometimes fails and sometimes passes, without changes in the SUT configuration, internal or external (environment) state or test. This metric related to flaky tests will be collected to help answer VQ2.
- K03 - Better test quality: This metric is measured in terms of mutation score. This score tells us to which extent changes in the code are detected by tests. The STAMP tool Descartes finds pseudo-tested and partially tested methods - where the 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 extent, the identification and fixing of such issues increase test confidence.

This VQ2 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 deployment environments? The exact nature of the execution environment depends

[1] https://martinfowler.com/bliki/TestCoverage.html

on the use case - on the nature and scope of the software and how broad or narrow are the potential or intended set of target environments. Yet, all software has external dependencies, whether it could 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 unique paths through the SUT by varying configurations.
- K05 - System-specific bugs: We check if STAMP tools help to find bugs which only occur in specific configurations.
- K06 - More Configuration/Faster Tests: We check if STAMP tools help to run the SUT on more configurations.
- K08 - More crash replicating test cases: If the amplification of configuration testing produces new crashes, STAMP tools may also be able to create tests which reproduce such crashes.
- K09 - More production level test cases: if the behavior of the SUT can drive the generation of new test cases, these can be used to verify the correct behavior of the SUT under different target environments.

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 writing and executing test cases. Here we assess the capacity of the STAMP tools to speed up the development of three specific types of test cases: configuration-dependent test cases, crash replicating test cases, and production level 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: the generation of crash replicating test cases will be evaluated against the manual creation of such tests.
- K09 - More production level test cases: the generation of production level test cases (driven by the observation of the SUT behavior) will be evaluated against the manual creation of such tests.

This VQ4 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*.

All use case partners use the STAMP tools in accordance to their use case(s) in order to answer the VQs (by measuring the KPIs) for their use case(s). The different use cases have alternative goals contributing to their methodology and answers. These are reported in the use case descriptions. Based on the conclusions found in each use case, we will summarize and report the general conclusions for these validation questions.

7. Activeeon Use Case Validation

Activeeon Use Case Highlights

-More generated configurations using CAMP (8 more configurations for database support)

-More configuration-related bugs (5 new more discovered bugs)

-More discovered execution paths thanks to new PWS configurations (from 38% to 69% of total unique paths)

-Catalog code coverage increases from 42% to 52% (21% growth). The mutation score increases from 35% to 43% (23% growth) integrating Descartes.

7.A. Use Case Description

7.A.1 Target Software

ProActive Workflow & Scheduler (PWS) solution is the main product in the context of the Activeeon use case. This product was thoroughly described in Deliverable 5.6 (D5.6). To avoid repeating the content, we hereafter continue from that point and invite the eager reader to read D5.6 for a thorough use case description.

7.A.2 Experimentation Environment

No change from deliverable 5.6.

7.A.3 Expected Improvements

No change from deliverable 5.6.

7.A.4 Business Relevance

No change from deliverable 5.6.

7.B Validation Experimental Method

7.B.1 Validation Treatments

Table 3 Experimental treatments applied in ActiveEon case

Metric	KPIs	STAMP tool	Treatment for measuring
Code coverage	K01	● Dspot ● Descartes ● Botsing	Code coverage is measured using Sonar inside the *Scheduling* project and using Clover inside the *Catalog* project. 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 *Scheduling* because it does not currently support multi-module Gradle projects. The STAMP tools are applied one by one to the project in order to avoid a cumulative result.

			Each change to the project has been tracked and measured to compute the tool result impact.
More flaky tests identified & handled	K02	● Descartes ● CAMP	There is a baseline of flaky tests for the *Scheduling* project, where we keep a list of them. Descartes and CAMP are used to stress out several execution paths. This will help to rise flaky tests that fail due to changes in non deterministic way, these tools are indirectly introducing such vulnerability.
Mutation Score	K03	● Dspot ● Descartes	The mutation score is measured using PIT on the projects *Scheduling* and *Catalog*. The tools are applied to the project one by one to avoid cumulative result. For each change the project mutation score is evaluated to measure the tool impact.
Pseudo Tests Partial tested tests	K03	● Descartes	Count the issues reported by Descartes on the *Scheduling* and *Catalog* project. The relevant issues are fixed and then the issues are counted once again.
Execution Paths	K04	● CAMP	Count how many unique traces are obtained after running tests over different PWS configurations generated by CAMP.
Configuration related bugs	K05	● CAMP	PWS System tests are going to be used to evaluate the different generated configurations related to database support.
More config / Faster	K06	● CAMP	Count the number of new generated PWS configurations using CAMP.
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 tests generated by Botsing that are introduced in the test suite.

7.B.2 Validation Target Objects and Tasks

PWS, the Catalog and the Scheduling projects were already described in the D5.6. Here we will describe why we also used the Programming project as a test project.

We experimented with the STAMP tools on 3 projects that belong to PWS, namely: Catalog, Scheduling and Programming.

Catalog was selected as the most relevant project among all Activeeon projects to validate the tools. It has several properties that makes it the most relevant among the 40 projects of Activeeon:

- Java language
- Source code available on Github
- 4 years-old project
- Integration and unit tests
- Spring framework
- Microservice project (~ 5000 lines of code)

The Scheduling and Programming project were chosen in order to increase the heterogeneity of our use case. As a consequence, we expect that they will reveal issues that were not highlighted by the catalog project.

Scheduling is a ten-year old multi-module project. It contains 90085 lines of code as reported in D5.6. It is actively evolving. The Scheduling project is developed in Java and embeds a Jetty server.

Programming project is a library that contains the parallel algorithm that accelerates the Scheduling computations. The programming project contains 74412 lines of code. While this project is not evolving anymore, it is however still maintained.

7.B.3 Validation Method

Descartes

No change from D5.6.

Dspot

Catalog project was used for the Dspot validation. Catalog is a gradle project, as a consequence, Dspot ran with the GradleBuilder option.

We aim at getting the maximum value that Dspot can bring by using it to generate a maximum number of tests. To maximise the chance to generate new tests, we run Dspot 11 times on the same source code with a new amplifier each time.

The tested amplifiers are the following:

- MethodAdd
- MethodRemove
- TestDataMutator
- MethodGeneratorAmplifier
- ReturnValueAmplifier
- AllLiteralAmplifier
- NullifierAmplifier

CAMP

To explore configuration related bugs and execution path related KPIs, we will rely on the CAMP tool. As previously discussed in D5.6, PWS can be configured to run with external databases such as: MySQL, MariaDB,PostgreSQL, SQL server, Oracle, etc. This implies that upon a new release, we currently have to manually test PWS with all these databases to assess the production quality. By default, PWS connects to HSQLDB. To do so, CAMP will allow us to configure PWS in order to generate different configurations for database support. In the following, we describe how CAMP is used to measure CAMP-related KPIs, namely K04-More unique traces, K05-System specific bugs, and K06-More config / Faster.

Configuration generation:

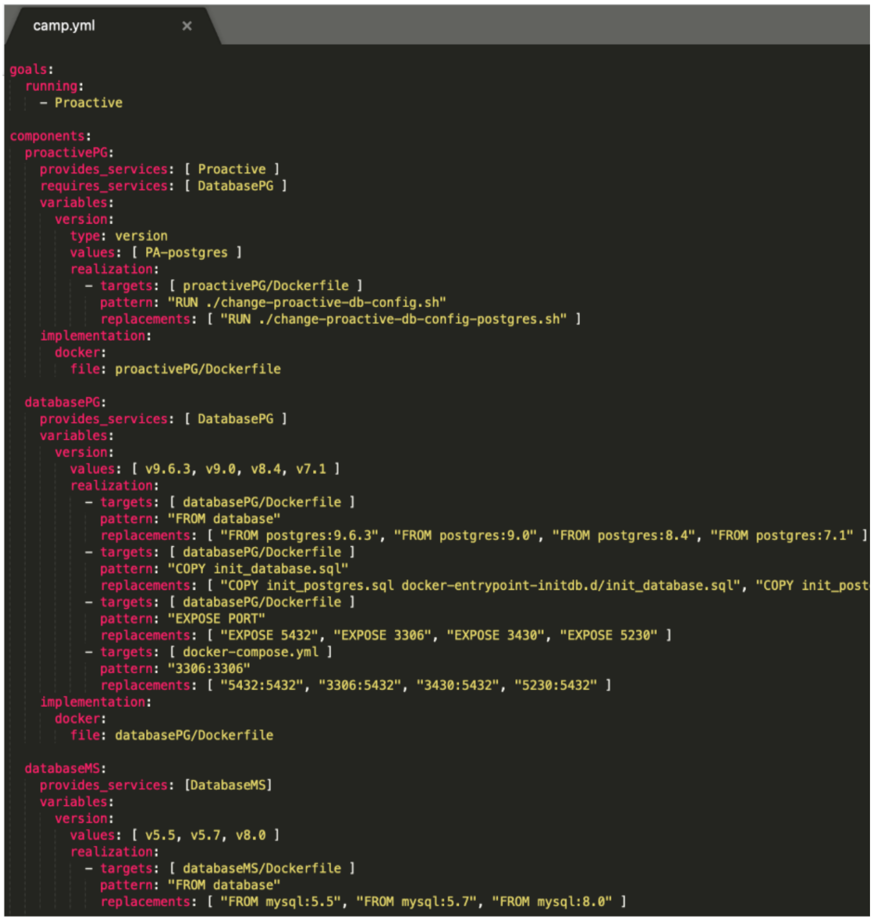
PWS configuration for multi-database support

CAMP will help us to automate the build and the configuration of PWS to support different database systems. The figure below shows an overview of generated configurations. Using CAMP, we will:

- Generate different configurations of Proactive into Docker containers: we are going to use the same nightly build (as produced in our CI) for two configurations: MySQL, and PostGres, with different versions each. To connect to those DB systems, one should manually configure PWS to provide the right connection parameters (the driver, password, etc). By default, PWS is configured to support only HSQLDB. CAMP will do this automatically.
- Generate different DB Docker images (MySQL, and PostGres) with different versions: CAMP will create different DB Docker images that will contain the initial DB schema for every database type.

Building camp last version and generate configurations

Now, we are going to show how to use CAMP to generate these configurations. The project source code is available here https://github.com/STAMP-project/camp-samples/tree/master/activeon.
The figure below shows the input CAMP file used to define the different needed configurations. We distinguish 7 more configurations: PWS with MySQL DB v5.5, v5.7, v8.0 and PWS with Postgres DB v7.1, v8.4, v9.0, v9.6.3.



camp.yml

```yaml
goals:
  running:
    - Proactive

components:
  proactivePG:
    provides_services: [ Proactive ]
    requires_services: [ DatabasePG ]
    variables:
      version:
        type: version
        values: [ PA-postgres ]
        realization:
          - targets: [ proactivePG/Dockerfile ]
            pattern: "RUN ./change-proactive-db-config.sh"
            replacements: [ "RUN ./change-proactive-db-config-postgres.sh" ]
    implementation:
      docker:
        file: proactivePG/Dockerfile

  databasePG:
    provides_services: [ DatabasePG ]
    variables:
      version:
        values: [ v9.6.3, v9.0, v8.4, v7.1 ]
        realization:
          - targets: [ databasePG/Dockerfile ]
            pattern: "FROM database"
            replacements: [ "FROM postgres:9.6.3", "FROM postgres:9.0", "FROM postgres:8.4", "FROM postgres:7.1" ]
          - targets: [ databasePG/Dockerfile ]
            pattern: "COPY init_database.sql"
            replacements: [ "COPY init_postgres.sql docker-entrypoint-initdb.d/init_database.sql", "COPY init_post
          - targets: [ databasePG/Dockerfile ]
            pattern: "EXPOSE PORT"
            replacements: [ "EXPOSE 5432", "EXPOSE 3306", "EXPOSE 3430", "EXPOSE 5230" ]
          - targets: [ docker-compose.yml ]
            pattern: "3306:3306"
            replacements: [ "5432:5432", "3306:5432", "3430:5432", "5230:5432" ]
    implementation:
      docker:
        file: databasePG/Dockerfile

  databaseMS:
    provides_services: [DatabaseMS]
    variables:
      version:
        values: [ v5.5, v5.7, v8.0 ]
        realization:
          - targets: [ databaseMS/Dockerfile ]
            pattern: "FROM database"
            replacements: [ "FROM mysql:5.5", "FROM mysql:5.7", "FROM mysql:8.0" ]
```

Figure 1 CAMP input file to define the replacement strategies and configuration settings

Now, to build CAMP, go to project root https://github.com/STAMP-project/camp and use:

```
docker build . -t camp
```

To run CAMP within the created image we use:

```
docker run -it camp bash
```

Working within the docker container has some drawbacks. For example, the changes on the current drive are not persistent. Alternatively, we can also use a mounting point so CAMP container access an outside folder.

```
docker run -t -v $PWD:/campworkingdir camp camp -h
```

Now, we can proceed to use CAMP to demonstrate Activeeon use case. For instance, for this example clone and access to the Activeeon sample project https://github.com/STAMP-project/camp-samples/tree/master/activeon and then run:

```
docker run -t -v $PWD:/campworkingdir camp camp generate -d /campworkingdir
```

CAMP then generates new Dockerfiles, properly chained together according to the configurations generated by CAMP generate. To do so, you can apply the following command:

```
docker run -t -v $PWD:/campworkingdir camp camp realize -d /campworkingdir
```

You can clean exited container and unlinked images using:

```
docker system prune
```

Running configurations

Now that the configurations are generated, we need to do some modifications to make it work:

- we need to provide the host machine address to configure the PWS server on docker. e.g., on Mac or Linux we can use `ifconfig`.
- To get snapshot version of PWS from Jenkins, we need to provide the username and api-token first. Visit the jenkins site `http://<yourserver>/user/<username>/configure` and then generate the token.
- Now, we edit the script `get-last-proactive-do-not-commit.sh` and replace USERNAME and TOKEN by the credentials.

Run system-tests

- Now it is time to run system-tests over the generated configs.
- The Scheduling-system-tests project is a private repo on Bitbucket. Access to the repository has to be requested by Activeeon developers.

- Clone the project

```
git clone https://bitbucket.org/activeeon/scheduling-system-tests/src/master/
```

- Tests are executed using the following command

```
./gradlew clean -Plocal test --stacktrace
```

Unique execution traces:

We used Docker Flame Graphs to extract unique execution traces from generated CAMP configurations, available here: https://github.com/STAMP-project/camp/tree/docker-flame-graphs/samples/docker-flame-graphs.

Docker Flame graphs (DFG) is a CPU Flame Graph generator that profiles JVMs processes running inside Docker containers.

Flame graphs (FGs) are a visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately.

This project provides scripts and guidelines to:

- Generate different docker-based software configurations using CAMP
- Run stress tests over generated configurations using Jmeter
- Profile and record stack traces of running processes
- Generate configuration-specific Flame Graphs
- Generate differential Flame Graphs, useful to compare different profiles' performance.

In order to generate FGs, one needs a profiler that can sample stack traces. FGs use the Linux Perf toolset (http://www.brendangregg.com/perf.html) to profile any kernel and user space processes in a Linux machine.

Even though Perf and FGs are amazing tools to profile Linux processes, it remains challenging to profile Java processes. Moreover, if the Java process is in a container, it is even more annoying. Hence, to profile JVMs, we use Perf-map-agent (https://github.com/jvm-profiling-tools/perf-map-agent), which generates flame graphs and allows Perf top for Java programs by (1) attaching to the JVM and getting a symbol map, and (2) running Perf top of Perf record with the map file.

When a JVM is inside a Docker container it is inside its own PID namespace typically running as a user that only exists inside the container. This means that the JVM appears to have a different PID on the host than it does in the container.

To make Java perf agent work, we:

- Mount perf-map-agent inside each of our containers (e.g. have it on every server and mount the volume read only).
- Get the symbol map from inside the container using the container PID and copy it to the host temporary directory. Renaming it from the container PID to the host PID.
- Run Perf on the host as root to get top or a FG.

Botsing

Botsing is used to increase the code coverage by providing new test that will be integrated in the test suite. In order to generate tests, Botsing has two strategies.

The first strategy aims to use a production crash to generate a test that will reproduce the crash. This solution can be used to reproduce client issues from the logs they provide us. This strategy increases the number of replicated test cases and pushes for test driven development. Indeed, in order to fix the bug, the developer can add the generated test to his test suite. The generated test should make the test suite fail because the issue is not fixed yet. By fixing the code that makes the generated test fail, the developer should be able to resolve the issue.

The second strategy uses software behavior to generate new tests. It increases the accuracy of tools such as RAMP to generate tests, along the way making tests more readable and eventually more maintainable and usable.

7.B.4 Validation Data Collection and Measurement Method

Descartes

No change from D5.6

Dspot

During the second period, we manage to generate tests with Dspot. However, gathering results regularly may be hard due to Dspot's execution time. In order to be able to easily get results from Dspot, a Jenkins job was created. This job runs Dspot against the Catalog. It runs, one after the other, 11 Dspot amplifiers. The generated tests are gathered and pushed to the Catalog in a second branch. The KPI is gathered for Dspot by comparing the branch 8.5.X without the generated tests metrics to the branch dspot that contains the generated tests metrics.

PIT is used to measure code coverage and mutation testing.

CAMP

For CAMP, we first describe how the test data are collected and we then explain the measurement method for the CAMP experiment.

Configuration generation:

To test CAMP configurations, we choose to run a subset of test suites from the system tests project related to the GraphQL API. We selected to test the GraphQL API because it is used by most of PWS micro-services and it should behave correctly when instantiating different PWS configurations. Our system tests related to the GraphQL Rest API are executed against 2 of the 7 new generated PWS configurations (PWS + MySQL v9.6.3 and PWS + Postgres v5.7). Hence, we add the following gradle configuration to our system tests project:

```
task testStamp (type: Test){

  if (project.hasProperty('local')) {

      systemProperties "spring.profiles.active": "local"
  }

    include 'org/ow2/proactive/systemtests/test/rest/graphql/**'

}
```

Tests are executed using the following command:

```
./gradlew clean -Plocal testStamp --stacktrace
```

We ran these tests for every PWS configuration sequentially.

The subset of executed system tests is provided in the SchedulingApiGraphQlTest.java class available here https://github.com/STAMP-project/confampl-usecases-output/blob/master/activeeon/CAMP_Tests/SchedulingApiGraphQlTest.java. This test class contains 7 tests.

Unique execution traces:

The repository https://github.com/STAMP-project/camp-samples/tree/master/activeon/traces demonstrates how to collect traces from different PWS configurations, generated by CAMP. The measurement is based on the Flame graph and performance tools technologies as previously described.

`configs` folder contains three generated PWS configurations. Each configuration provides a different dockerized PWS version:

- `Config0 (PWS + HSQLDB)`: A preconfigured PWS version that is configured by default to connect to the default HSQLDB database. This is the baseline config used as reference.
- `Config1 (PWS + Postgres)`: A preconfigured PWS version that is configured to connect to the Postgres database.
- `Config2 (PWS + MySQL)`: A preconfigured PWS version that is configured to connect to the MySQL database.

The two latter configurations are generated using CAMP and the first one is used as baseline or reference.

Since profiling basically takes a snapshot of the running app *n* times per second (or at *n* Hz), there is no guarantee it will collect all traces. To increase the quality of the traces, we run the profiler several times, at different frequencies.

The `Dockerfile` for each configuration shows how to instrument PWS to collect traces at different frequencies. The script `run.sh` is executed for every configuration. It executes the following tasks:

- Run the PWS server `/activeeon_enterprise-pca_server-linux-x64-8.5.0-SNAPSHOT/bin/proactive-server &`. The JVM is configured with the options `–XX:+PreserveFramePointer -agentpath:/liblagent.so`
- Get the corresponding `PID` of the Java process
- Attach the profiler to the Java process `java -cp /libperfagent.jar:$JAVA_HOME/lib/tools.jar net.virtualvoid.perf.AttachOnce $PID`

- Wait until the PWS server is up `sleep 500`
- Run some tests/load using the PWS client. Here, we send and execute a `workflow-test.xml` in the scheduler `/activeeon_enterprise-pca_server-linux-x64-8.5.0-SNAPSHOT/bin/proactive-client -s /workflow-test.xml`
- We record traces for 300 seconds. The frequency is the default one `perf record -e cpu-clock -p $PID -a -g -o /data/perf.data -- sleep 300`
- We shutdown properly the scheduler using the PWS client.
- We copy traces to the mounted /data folder

Botsing

Botsing had few results in the report D5.6. In order to get more results, botsing-reproduction was tested against two other Activeeon projects. Those projects are the Programming library and the Catalog micro-service. Botsing was run against those projects using the Botsing Gradle plugin, more information about the plugin is available in report D4.4. The plugin downloads the project binaries and dependencies from Maven.

Changing the log provided and the target frame has a huge impact on Botsing execution. As a consequence, those two parameters were modified to test the execution. Indeed, providing a different log file makes Botsing target a different part of the code which results in a different execution. The provided value for the target frame will always go from the higher value to the less. Indeed, using the higher value will be more complex because Botsing has more path to find and to reproduce. Increasing the target frame increase the complexity for Botsing for reproducing the issue when lowering the target frame reduce the complexity. By testing Botsing with several stack traces and target frames we aim at identifying in which cases Botsing has the better results.

In the programming project, the tested crashes come from production issues that were created on Github by our users or developers. In the catalog project, the crashes come from the client error (bad name, wrong entry format) or handled server error (network issue, missing resources) that may happen.

RAMP will be used against the catalog project. Catalog models will be generated using RAMP model generation runner. Then those models will be used with RAMP as an entry in order to generate tests for the catalog. The number of tests and their quality will be evaluated. The test quality will be evaluated with the following criteria:

- Does the test compile?
- Does the test execute without issues?
- Does the test bring values in the test suite in code coverage?
- Does the test bring values in the test suite in mutation score?
- How big are the similarities between the existing tests and the generated tests?

7.C Validation Results

7.C.1 KPIs Report

KPI Summary

Table 4 Summary of KPIs for ActiveEon case

KPI	**Measure**			**Difference with objective**
	Baseline	**Treatment**	**Difference**	
K01-Execution Paths (Catalog)	**42%** (Code Coverage)	**52%** (Code Coverage)	**+21%**	**+1%** (20%)
K02-Flaky Tests (PWS)	**46 detected**	**23 identified and handled**	**+50%**	**+30%** (20%)
K03-Better Test Quality (Catalog)	**35%** (mutation score)	**43%** (mutation score)	**+23%**	**+3%** (20%)
K04-More unique traces (PWS)	**38%**	**69%**	**+82%**	**+42%** (40%)
K05-System specific bugs (PWS)	**0**	**5**	**+5 (+500%)**	**+460%** (40%)
K06-More config / Faster (PWS)	**1**	**8**	**+7 (+233%)**	**+183%** (50%)
K08-More crash tests (Catalog)	**16**	**1**	**6.3%**	**-64.7%**(70%)
K09-More prod tests (Catalog)	**184 unit tests**	**149 unit tests generated**	**+81%**	**+71%**(10%)

K01 - More execution paths

In this section we describe the tests that were automatically generated by DSpot. Figure 2 shows only the tests that were auto generated by DSpot in the Catalog project.

```
.
└── dspot
        └── src
        └── test
        └── java
└── org
    └── ow2
    └── proactive
    └── catalog
        ├── graphql
        │   └── handler
        │       └── catalogobject
        │       └── CatalogObjectAndOrGroupFilterHandlerTestDspot.java
        ├── repository
        │   └── entity
        │       ├── BucketEntityTestDspot.java
        │       └── CatalogObjectRevisionEntityTestDspot.java
        ├── rest
        │   └── controller
        │       └── CatalogObjectReportControllerTestDspot.java
        └── service
        ├── BucketServiceTestDspot.java
        ├── CatalogObjectServiceTestDspot.java
        └── KeyValueLabelMetadataHelperTestDspot.java
```

Figure 2 Seven test classes automatically generated by DSpot amplification methods.

The tests generated by Dspot generated tests can be found at the following URL: https://github.com/STAMP-project/dspot-usecases-output/tree/master/activeeon/catalog

Among the eight used amplification operators, three have led to generated tests. Dspot generated seven classes containing 15 unit tests. The amplifier allLiteralAmplifier generated one test. The amplifier NullifierAmplifier generated two tests. The amplifier MethodGeneratorAmplifier generated 12 tests.

Below in Figure 3, we find an example of a generated test case. In this particular example, the amplification found that an ArrayIndexOutOfBoundsException was not thrown if an invalid array index is provided.

```
@org.junit.Test(timeout = 10000)
public void testHandleMethod_literalMutationNumber49_literalMutationNumber351_failAssert0() throws
java.lang.Exception {
            try {
java.util.Optional<org.springframework.data.jpa.domain.Specification<org.ow2.proactive.catalog.repository.e
ntity.CatalogObjectRevisionEntity>> specification = andFilterHandler.handle(whereArgs);
        org.ow2.proactive.catalog.repository.specification.catalogobject.OrSpecification orSpecification =
((org.ow2.proactive.catalog.repository.specification.catalogobject.OrSpecification) (specification.get()));
        org.ow2.proactive.catalog.repository.specification.catalogobject.AndSpecification leftAnd =
((org.ow2.proactive.catalog.repository.specification.catalogobject.AndSpecification)
(orSpecification.getFieldSpecifications().get(-1)));
        org.ow2.proactive.catalog.repository.specification.catalogobject.AndSpecification rightAnd =
((org.ow2.proactive.catalog.repository.specification.catalogobject.AndSpecification)
(orSpecification.getFieldSpecifications().get(1)));
        org.ow2.proactive.catalog.repository.specification.catalogobject.OrSpecification rightAndChildOr =
((org.ow2.proactive.catalog.repository.specification.catalogobject.OrSpecification)
(rightAnd.getFieldSpecifications().get(1)));
            org.junit.Assert.fail("testHandleMethod_literalMutationNumber49_literalMutationNumber351 should
have thrown ArrayIndexOutOfBoundsException");
            } catch (java.lang.ArrayIndexOutOfBoundsException expected) {
                org.junit.Assert.assertEquals("-1", expected.getMessage());
            }
}
```

Figure 3 An example test case from DSpot automatic generated tests.

By running PIT against the Catalog project, we obtained the following metrics:

- Line coverage: 43%, 871/2020 lines covered
- Mutation coverage: 35%, 192/546 mutation killed

When we added the generated tests to the test suite, we obtained the following new metrics with PIT:

- Line coverage: 44%, 891/2020 lines covered
- Mutation coverage: 40%, 220/546 mutation killed

Adding Dspot generated test to the test suite increased the lines covered by 1% and the mutation score by 5%. Using Dspot enabled to increase the reliability in the software by increasing the code coverage. It also increased the confidence in our test suite by increasing the mutation score.

Catalog had a code coverage of 42% at the start of the project with 871 lines covered out of a total of 2020 lines of code. At the end of the project, the code coverage is 52% with 1349 out of 2608 lines covered. During the STAMP project the number of lines of code for the Catalog project has increased by 588 lines of code (a 29% growth). The number of covered lines went from 42% to 52%, which is a **21% increase**.

K02 - Less flaky tests

Scheduling has 46 flaky tests identified at the start of the project. Since the STAMP beginning, we have identified and fixed 10 flaky tests in the scheduling project, 10 in the System-tests projects and 5 in the Job-planner project. **23 flaky tests** have been identified and fixed, which is a **50% increase** compared to the baseline.

K03 - Better test quality

From month 18 to month 36, Descartes was evaluated against the new version 10.0.1 of the Catalog. Actions were taken in order to increase the quality of the Catalog test suites. The reports are available on the STAMP project’s Github: https://github.com/STAMP-project/descartes-usecases-output/tree/master/activeeon/catalog

The following table summarizes the results gathered before and after taking action.

Table 5 First Catalog Project Scores

Type of test	Score	Score after focusing on adding tests	Description	Action
line coverage	39%	52%	Percentage of lines of code that are explored by the test set.	Improve and add tests that cover related lines of code.
mutation score	38%	43%	Percentage of lines of code in which mutations were never explored by the tests.	Manually inspect each test case and evaluate survived mutations.
pseudo-tested	36	48	Tests that pass for all related mutations.	
partially-tested	5	7	Tests that pass for at least one related mutation.	

The first action taken to increase code coverage was to add tests. We observed that adding tests also increases the mutation score. However, it also increases the number of pseudo-tested and partially-tests methods. Increasing the code coverage by 13% created 14 new issues. Among those 14 issues, 12 are new pseudo-tested method and 2 are new partially-tested method. In the D5.6 report, most of the remaining issues were false positives that were created by generating code libraries. In the new report, the 14 new issues are relevant and fixing them enables to increase the mutation score and test quality.

K04 - More unique invocation traces

After running the Flamegraph-based profiling tool, we obtain the results below:

```
---- Java traces ----

Configuration 0 contributed 3258/8573 = 38.00 % of all unique traces

Configuration 1 contributed 5916/8573 = 69.00 % of all unique traces

Configuration 2 contributed 3515/8573 = 41.00 % of all unique traces

---- System traces ----

Configuration 0 contributed 1438/3422 = 42.00 % of all unique traces
Configuration 1 contributed 1883/3422 = 55.00 % of all unique traces

Configuration 2 contributed 1575/3422 = 46.00 % of all unique traces
```

-For PWS + HSQLDB: It contributes to 38.00 % of all unique traces for Java traces and 42.00 % of all unique traces for System traces.

-For PWS + Postgres: It contributes to 69.00 % of all unique traces for Java traces and 55.00 % of all unique traces for System traces.

-For PWS + MySQL: It contributes to 41.00 % of all unique traces for Java traces and 46.00 % of all unique traces for System traces.

The traces and raw data are available in the `profiling` folder of each config (see https://github.com/STAMP-project/camp-samples/tree/master/activeon/traces/configs)

The three configurations contributed differently to discover unique traces. For Java traces, the CAMP generated configurations (PWS + MySQL and Postgres) discovered more unique traces than the default HSQLDB PWS configuration (the reference). This shows that the newly generated CAMP configurations have allowed us to discover more JVM traces (which means more code coverage). For System traces, we obtain the same result as for Java traces with different traces coverage respectively, 42, 55, and 46%. We also remark that PWS + Postgres configuration generates the highest trace coverage compared to other configurations when considering Java and system traces. By increasing the unique Java traces from the baseline 38% to 69% in Configuration 2, we increased the number of unique Java traces by **82%**.

K05 - System specific bugs

Configuration generation:

The results of failing system tests are provided in CAMP Results (https://github.com/STAMP-project/confampl-usecases-output/blob/master/activeeon/CAMP_Results)

- Config 1: PWS + MySQL database

7 executed tests for config 1: PWS + MySQL database pass with no reported errors.

See the test report available at index.html (https://github.com/STAMP-project/confampl-usecases-output/blob/master/activeeon/CAMP_Results/Test_Report/Config_1_Proactive-MS/index.html).

- Config 2: PWS + PostgresQL database

5 among the 7 executed tests for Config 2: PWS + PostgresQL database fail.

See the test report available in index.html (https://github.com/STAMP-project/confampl-usecases-output/blob/master/activeeon/CAMP_Results/Test_Report/Config_2_Proactive-PG/index.html).

See the complete Docker container logs of the two running services after running these tests docker-log-errors.log (https://github.com/STAMP-project/confampl-usecases-output/blob/master/activeeon/CAMP_Results/Test_Report/docker-log-errors.log).

The reported error messages are the following:

```
database_1   | ERROR:  operator does not exist: text ~~ bigint at character 1413
database_1   | HINT:  No operator matches the given name and argument type(s).
```

```
You might need to add explicit type casts.
```

The failing tests in Config 2 are due to a wrong configuration of the PWS server with the PostgresQL database. It is related to a wrong given field type to the PostgresQL DB. This bug was not detected before CAMP because system tests were not run against different PWS configurations. Generally, when we would like to test different databases we do it manually and we only run some manual functional tests. Thanks to CAMP, we can now run our system tests automatically and detect eventual miss configurations leading to bugs.

K06 - More Configuration/Faster tests

The figure below shows the 7 CAMP generated configurations: PWS with MySQL DB v5.5, v5.7, v8.0 and PWS with Postgres DB v7.1, v8.4, v9.0, v9.6.3.

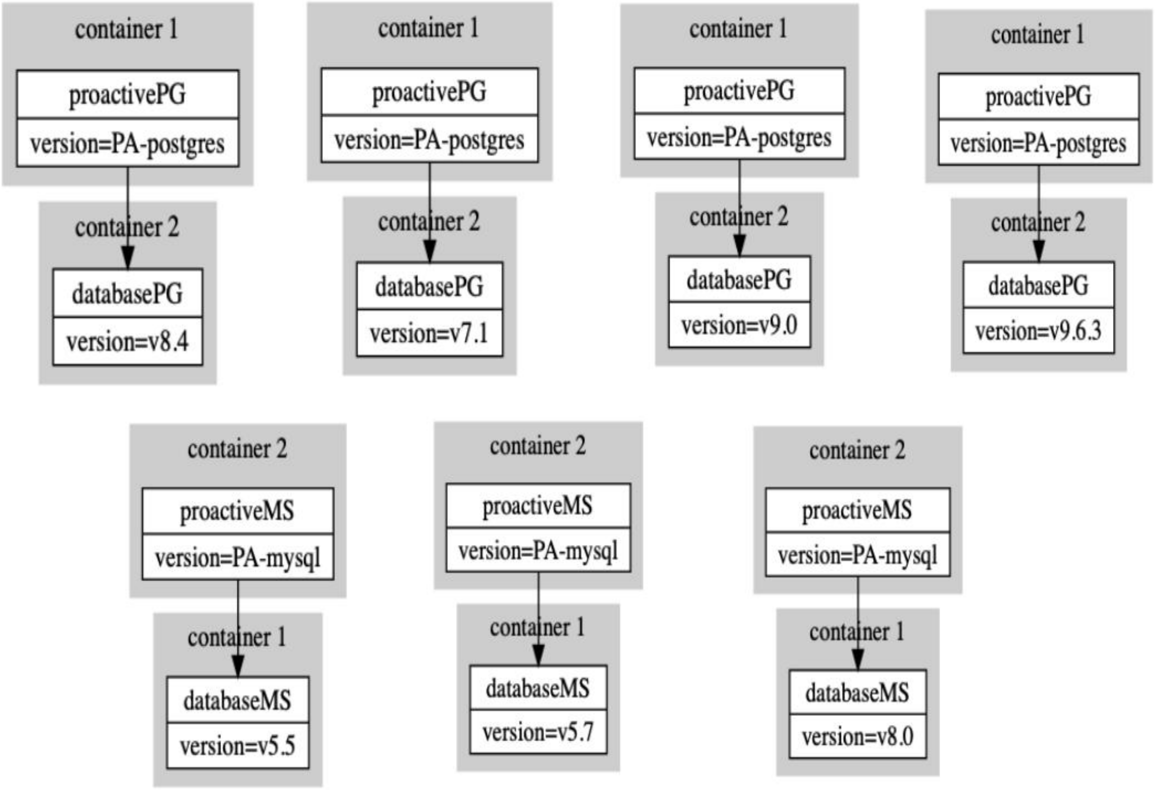


Figure 4 PWS configurations generated by CAMP to support multiple DBs.

CAMP will generate an output folder where all docker-based configuration will be stored. Finally, to start the PWS server with different generated DBs, we go to the output folder generated by CAMP, `cd out/config_1` and `cd out/config_2` and run `docker-compose up` to start both containers (PWS + database). These steps worked well and all generated PWS version are functional.

K08 - More crash replicating test cases

Botsing

Botsing reproduction was executed against three projects: scheduling, programming and the catalog. The executions results can be found on Github: https://github.com/STAMP-project/botsing-usecases-output/tree/master/Activeeon/Botsing

Botsing 1.0.3 against Scheduling

Botsing version 1.0.3 was evaluated against the Scheduling project. We reported in deliverable D5.6 that we managed to generate a test with the first Botsing-reproduction version for the *SpaceNotFoundException* crash. In the new version, we manage to regenerate the test for this crash to ensure that there was no regression during the version upgrade. We tested 7 crashes for the scheduling projects:*BodyTerminateReplyException*, *BodyTerminatedRequestException*, *IOException6*, *JsonParseException*, *NullPointerException*, *ScriptException* and *SpaceNotFoundException*.

Here are the results for each crash:

- *BodyTerminateReplyException* generated an empty test.
- *BodyTerminatedRequestException* generated an empty test.
- *IOException6 generated* an empty test.
- *JsonParseException* was executed with target frame from 4 to 1. None of them were able to generate a test from the stack trace.
- *NullPointerException* was executed with target frame from 8 to 1. None of them were able to generate a test from the stack trace.
- *ScriptException* generated an empty test.
- *SpaceNotFound* was executed with the target frame from 1 to 9. From the target frame 6 to 9, no test was generated. For the target frame 4 and 5, empty tests were generated. For the target frame from 1 to 3, tests were generated.

We have managed to generate a test for one crash, we generated 4 empty tests and 3 times the test generation failed.

Botsing 1.0.4 against Programming

Botsing version 1.0.4 was tested against the Programming project. Botsing was tested with five different stack traces from three crashes: *illegalArgumentException*, *illegalStateException* and *pamrAndJavassistDeadlock*. The crash *illegalArgumentException* had two “caused by” subsections, so it has to be separated in three different inputs for Botsing.

Here are the results for each input:

- *illegalArgumentException* 1 was executed from target frame 10 to 1. It generated an empty test for target frame 1.
- *illegalArgumentException* 2 was executed from target frame 10 to 1. It generated an empty test for target frame 1.
- *illegalArgumentException* 3 was executed with target frame 2. It generated an empty test.
- *illegalStateException* was executed from target fame 9 to 1. It generated two empty tests for frame 2 and 4.
- *pamrAndJavassistDeadlock* didn’t generate anything. It crashed because Botsing does not support stack traces with lock.

Botsing 1.0.7 against Catalog

Botsing version 1.0.7 was evaluated against four crashes: *accessDeniedException*, *bucketAlreadyExistException*, *bucketNameIsNotValidException* and *bucketNotFoundException*.

Here are the results for each stack trace:

- *accessDeniedException* was tested for the frame 2, 5, 6, 23 and 25. It threw a class not found exception for the target frame 5 and 6. For the target frame 2, 23 and 25, it generated empty tests.

- *bucketAlreadyExistException* was executed for the target frame 1, 22 and 43. It threw a class not found exception for the target frame1. For the target frame 22 and 43, it generated empty tests.
- *bucketNameIsNotValidException* was tested for the target frame 12, 13, 53 and 55. Frame 12 and 13 threw a class not found exception. The target frame 53 and 55 generated empty tests.
- *bucketNotFoundException* was executed with the target frame 3, 15 and 32. Frame 3 threw a class not found exception. Frame 48 and 32 enabled to generate empty tests. We ran Botsing two times with the target frame 3. It gathered the binaries and dependencies from Maven or from the local ones provided manually. The aim was to confirm that the missing classes are due to an issue coming from Botsing and not from the provided arguments.

Results summary

The table 6 summarize the results obtained by Botsing reproduction executions.

Table 6 Botsing reproduction generation results

	Botsing 1.0.3 Scheduling	Botsing 1.0.4 Programming	Botsing 1.0.7 Catalog
Number of execution	23	30	15
Execution failed/No test generated	15	25	6
Empty test generated	5	5	9
Test generated	3	0	0
Successful generation (with empty test)	35%	20%	60%
Successful generation (without empty test)	7.7%	0%	0%

Out of the 16 tested crashes only the *SpaceNotFound* stack trace generated a relevant test (not an empty test). On Activeeon projects, Botsing has a **6.3%** success rate.

K09 - More production level test cases

RAMP generated 291 models from the Catalog project. Those models can be accessed in the directory models. RAMP succeeded in generating 149 tests classes from those models. Those tests can be accessed in the directory evosuite-tests. The Catalog project contained 184 unit tests when RAMP was applied on it. By generating 149 unit tests, we have increased the number of tests by **81%**.

7.C.2 Qualitative Evaluation and Recommendations

Descartes

Descartes is currently the simplest and complete tool to evaluate a software test suite. It provides the code coverage and the mutation score as metrics. Moreover, the issue reports enable to quickly identify the tests to solve and the fix to apply. Configuring the tool is easy with Gradle or Maven. In order to improve Descartes, the methods generated by libraries should be removed from the report. It would increase the report relevance and make it more usable for the developers.

DSpot

From our tests within PWS we assess the usability and relevance of DSpot as the test amplification tool we have missed all along. Nevertheless, we do point point out that running DSpot is very time consuming and we feel the solution is too complex because of the multitude of possible parameter values. For example, future work can investigate whether preconditioning steps that detect redundant execution paths and thus reduce the target space can mitigate concerns with regard to running time.

Camp

As CAMP is now able to generate different PWS configurations for database support, Activeeon uses these generated versions to run functional and integration tests. Before CAMP, all PWS configurations are configured manually in order to run tests or to do demos to customers. Moreover, CAMP is going to be used at the CI level to generate more configurations and to find more bugs. On the other hand, tracing PWS executions has been a success as the Flame Graphs tool (to trace JVM traces) is now used to find regression bugs related to CPU and Memory usage. Whenever performance overhead is observed, Flame graphs are used to identify the most called unique traces that caused that issue. Activeeon has already used this technology during the project to identify 2 critical regression bugs.

Botsing

We achieved some success with Botsing, mostly for the simpler projects. For example, the Catalog projects has less lines of code than the Scheduling and the Programming projects. It is our aim to evolve our software architecture towards less lines of code, which will help the future usage of Botsing.

The biggest issue that holds back our use of Botsing in production, is the number of successfully generated tests. In particular, the generation of empty tests and “class not found issues” should be reduced. Another issue that we encountered with RAMP is that generated tests should not have compilation issues. Once these issues are fixed, developers will be able to apply Test Driven Development with the generated 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?

Yes. The CAMP experiment regarding KPI04 (unique execution paths) showed that running tests over different configurations leads to discover more unique execution traces that were not previously reached by the baseline version.

Botsing generated tests reproduce issues that were not detected during the initial testing phase. By running the generated tests, some areas of the code, that were not previously tested, will be executed. Moreover, the test will provide the state of objects where the code will crash. It does not only traverse areas of code that were not reached, but also highlight the undesired code behavior.

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

Yes. Having a high code coverage lower the chance of containing undetected software bugs in the solution. It implies that the software behavior is checked and that the behavior will not be modified without breaking the test suite. The Dspot and Botsing tools enable to generate tests that can be added to the test suite in order to increase the code coverage. By doing so, they increase the confidence in test cases. The mutation score provided by PIT tools also provides a good indicator of the confidence that we can have in the test suite. Dspot generated tests enable to increase the mutation score. Descartes provides a report that lists the tests that lower the mutation score and provides a description of the issues. By following Descartes comments, a developer can fix the test issues. Fixing the test will increase the mutation score and increase the user's confidence in his test cases.

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

Yes. We increase the confidence in the resilience of PWS suite under different configurations on various environments thanks to the use of CAMP. CAMP eases the configuration of the test environment by automatically generating and executing new PWS configurations using Docker. This saves the time to configure the machine which requires a set of tools and libraries to run the tests.

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

Yes. Thanks to CAMP we make the test of different PWS configurations faster as the configurations and tests are automatically executed. Previously, this was manual work.

The STAMP tools plugin enables to easily generate tests from the build tools Gradle and Maven. For the developer who uses those tools, it is easy to run Dspot in order to generate tests that will upgrade the test suite's quality. Generally, STAMP tools improve the code development by increasing both, the code coverage and the mutation score. Descartes will also help developers to quickly identify how to increase the mutation score in order to reach the required level of confidence.

7.D Global Takeaway

The STAMP project enabled to integrate mutation score metrics in the Activeeon QA process. During this project, Activeeon developers learnt the mutation testing technique and how it can be used to validate a test suite. By working with the STAMP partners and tools, code coverage and mutation score were improved for the PWS SUT projects.

Using CAMP enabled to test several PWS configurations that were not automatically tested. It highlighted several configuration bugs that were fixed. The Flamegraph tool is also integrated in the QA process of Activeeon.

When the STAMP project started, the PWS logging mechanism was really basic: all logs were stored in a single file. In order to gather the required logs more easily for Botsing execution, we upgraded the whole PWS logging mechanism making it more powerful and efficient. Now, every microservice has its dedicated log file with its dedicated configuration.

8. Atos Use Case Validation

Atos Use Case Highlights

- High code coverage (65% more) and mutation score (69% more) improvement thanks to the combined application of STAMP techniques/tools (DSpot, Descartes, RAMP).
- Highly improved code/test base quality after refactoring driven by Descartes' report.
- Full automated (CI/CD pipeline integrated), fast test (stress/functional) execution under different amplified environments managed by CAMP (10, 32 new configurations), showing amplification of behavior (e.g. unique traces, a 10% more) and detection of faulty configurations (4 of 10 tested, 1 or 32 tested).
- Fast generation of crash replicating tests (3 of 6 targeted) using Botsing.
- Fast generation of many production tests for model and helper packages using RAMP.

8.A. Use Case Description

8.A.1 Target Software

The context of the Atos use case and the concrete target applications for STAMP industrial experimentation and validation were described in section 8.A.1 of D5.6. This third industrial validation phase has targeted the same SUTs (e.g. SUPERSEDE IF and CityGo CityDash) as described in D5.6.

8.A.2 Experimentation Environment

The experimentation environment where Atos' third phase evaluation has been conducted is the same as the one that was described in section 8.A.2 of D5.6.

Updated versions of the STAMP tools have been installed in this environment. For all these tools, we have also installed CLI scripts, Jenkins jobs and pipelines to run the tools and collect their results:

- DSpot v2.1.0-v2.2.0 from Maven Central repository.
- Descartes v1.2.5 from Maven Central repository.
- CAMP v0.6.2 from DockerHub repository.
- Botsing v1.0.7 from GitHub repository.
- RAMP v1.0.7 from GitHub repository.

8.A.3 Expected Improvements

The Atos validation objectives and expected improvements for this assessment period (M24-M36) evolved from those established during the previous period and described in D5.6:

CityGO CityDash

For technical reasons related to the availability of the functional testing environment for this use case (UC), the previous evaluation period focused on the first main objective described in D5.6:

- **Deliver an optimal CityGO platform for the current runtime infrastructure conditions**.

In this period, apart from evaluating the capability of CAMP's new release on assisting us to find the optimal CityGo platform configurations, we focus on the second objective:

- **Deliver CityGO CityDash onto different compatible runtime platforms provisioned by city administrators**

SUPERSEDE IF

For this UC, the main objective defined in D5.6, that is, to **ensure the runtime functional consistency of the SUPERSEDE backend platform** through a robust test-driven QA process**,** still have high relevance so they will still be pursued during this evaluation period.

Besides, in this period we emphasize our interest on STAMP technology to support SUT owners to understand the reasons causing SUT runtime failures. We rely on the STAMP technologies that generate crash replicating tests and new integration tests.

8.A.4 Business relevance

The business relevance of STAMP techniques and tools on the Atos operational portfolio was described in section 8.A.4 of D5.6.

8.B. Validation Experimental Method

8.B.1 Validation Treatments

The validation control and treatment tasks that have been conducted by Atos in their validation experiments, in order to compute the KPIs metrics, were already described in section 8.B.1 of D5.6. This section updates that one with new control and treatment tasks or with some updates on the existing ones.

Table 7 collects new and updated control tasks for computing baseline and control KPI metrics in SUPERSEDE IF and CityGO CityDash use cases:

Table 7 Control Tasks for Atos Validation Experimentation

Metric	KPIs	Task Name	Task ID	Description	UseCase
#configuration related bugs	K05	Detect configuration bugs	DCB	Stress tests and functional tests are executed on amplified deployment configurations using CI. **Stress tests**: Runtime logs are inspected to search for reported exceptions. Occurrences are manually investigated and reported. 5 different queries are sent to CityDash by 50 concurrent users created within a time slot of 10s **Functional tests**: 6 Selenium-base tests suites are designed, implemented and executed over each generated CityDash configuration. CI pipeline/job computes test results and reports.	CityDash
#configurations tested	K06	Compute number of configurations tested	CNCT	STAMP CAMP CI Executor is used within a Jenkins pipeline/job to generate, instantiate and test (using stress tests and functional tests) the	CityDash

				amplified configurations that it generates. CAMP reports the number of configuration tested.	
Time to deploy SUT in configuration	K06	Compute deployment time	CDT	STAMP CAMP CI Executor in Jenkins pipeline/job reports the deployment time for each generated configuration and the total aggregated deployment time (for all generated configurations).	CityDash
Number of existing Tests	K09	Count number of existing tests	CNET	Count the number, T, of existing tests for SUT in the baseline	IF

Table 8 describes the new and updated treatment tasks applied in SUPERSEDE IF and CityGO CityDash use cases:

Table 8 Treatment Tasks for Atos Validation Experimentation

Treatment	Treatment ID	Description	Tool (version)	KPIs	Use Case
Test Amplification with CAMP	TAC	SUT configurations are generated. Stress and functional tests are executed against a SUT instance deployed for several amplified configurations.	CAMP (0.6.2)	K04K05K06	City Dash
Production test amplification with RAMP	PTAWES	RAMP with behaviour model seeding is conducted upon the SUT baseline test suite. The number, P, of new production tests generated is counted. K09 = P/T (see control treatment CNET in table 8.1)	RAMP (1.0.7)	K09	IF

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 already described in section 8.A.1 of D5.6. Several artefacts have been developed to support and assist the experimentation process. This section describes the updates on these artifacts w.r.t. what it was described in section 8.B.2 of D5.6.

Software and artifacts are located in repositories (see Table 9) in GitHub (public access) and in Atos GitLab (restricted access)

Table 9 Atos code base Repositories for Use Cases (SUPERSEDE IF and CityGO CityDash)

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

	supersede-project/integration[2]	stamp-treatment	
CityGO CityDash	citygo stamp_Docker_citygoApp[3]	Master	CityGO/ Citygo/www/ShowcaseServer/demo_site

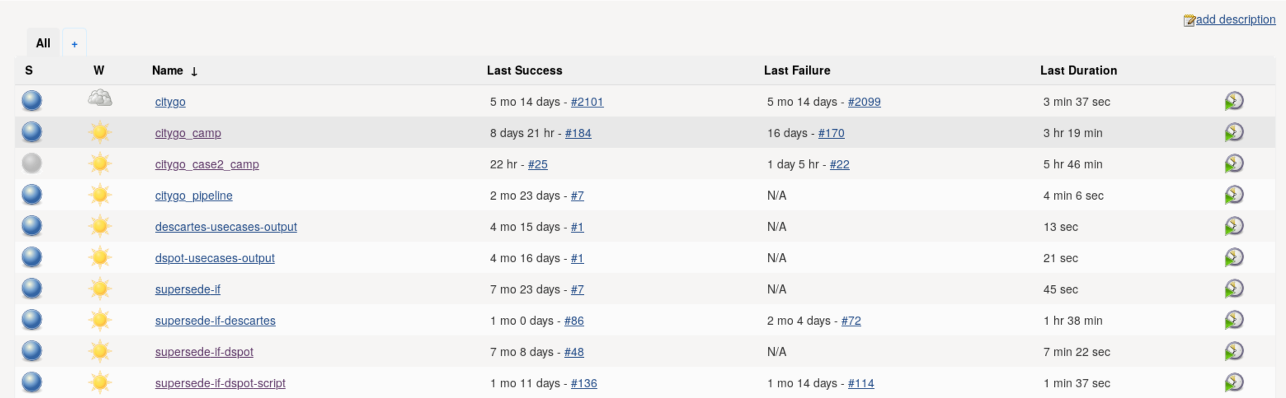
Use case: SUPERSEDE IF

The building and delivery process of the IF component is managed by a CI/CD Jenkins job. Building is managed by Maven.

DSpot and Descartes

STAMP control and treatment tasks for DSpot and Descartes have been incorporated to the Atos Jenkins CI[4] jobs and pipelines. Figure 5 shows the Atos jobs for IF and CityGo use cases. The main ones are:

- citygo_camp job: executes the performance scenario,
- citygo_case2_camp: job executes the functional scenario,
- supersede-if-descartes job: executes Descartes during the IF build,
- supersede-if-dspot job: executes DSpot during the IF build.



S	W	Name ↓	Last Success	Last Failure	Last Duration
●	☁	citygo	5 mo 14 days - #2101	5 mo 14 days - #2099	3 min 37 sec
●	☀	citygo_camp	8 days 21 hr - #184	16 days - #170	3 hr 19 min
○	☀	citygo_case2_camp	22 hr - #25	1 day 5 hr - #22	5 hr 46 min
●	☀	citygo_pipeline	2 mo 23 days - #7	N/A	4 min 6 sec
●	☀	descartes-usecases-output	4 mo 15 days - #1	N/A	13 sec
●	☀	dspot-usecases-output	4 mo 16 days - #1	N/A	21 sec
●	☀	supersede-if	7 mo 23 days - #7	N/A	45 sec
●	☀	supersede-if-descartes	1 mo 0 days - #86	2 mo 4 days - #72	1 hr 38 min
●	☀	supersede-if-dspot	7 mo 8 days - #48	N/A	7 min 22 sec
●	☀	supersede-if-dspot-script	1 mo 11 days - #136	1 mo 14 days - #114	1 min 37 sec

Figure 5 Jenkins jobs/pipelines for Atos Validation Experimentation

supersede-if-descartes Jenkins job:

This job[5] executes Descartes related control tasks (i.e. CMS, CTQ) with the following process:

1. Pull IF repository from SCM (GitHub)
2. Build IF project using Maven (skipping tests)
3. Execute Descartes tasks (CMS, CTQ)
4. Commit Descartes report into descartes-usecases-output repository
5. Visualize Descartes report using STAMP Tools Reports Publisher plugin for Jenkins. An example of Descartes report is shown in Figure 6, which shows the evolution of the mutation score over the

[2] https://GitHub.com/supersede-project/integration

[3] https://gitlab.atosresearch.eu/ari/stamp_Docker_citygoApp

[4] http://62.14.219.13:7777/ (restricted access)

[5] http://62.14.219.13:7777/job/supersede-if-descartes/configure (restricted access)

different job executions (left-top graph), the evolution of the number of detected pseudo-tested SUT's methods and the evolution of the number of detected partially tested SUT's methods.

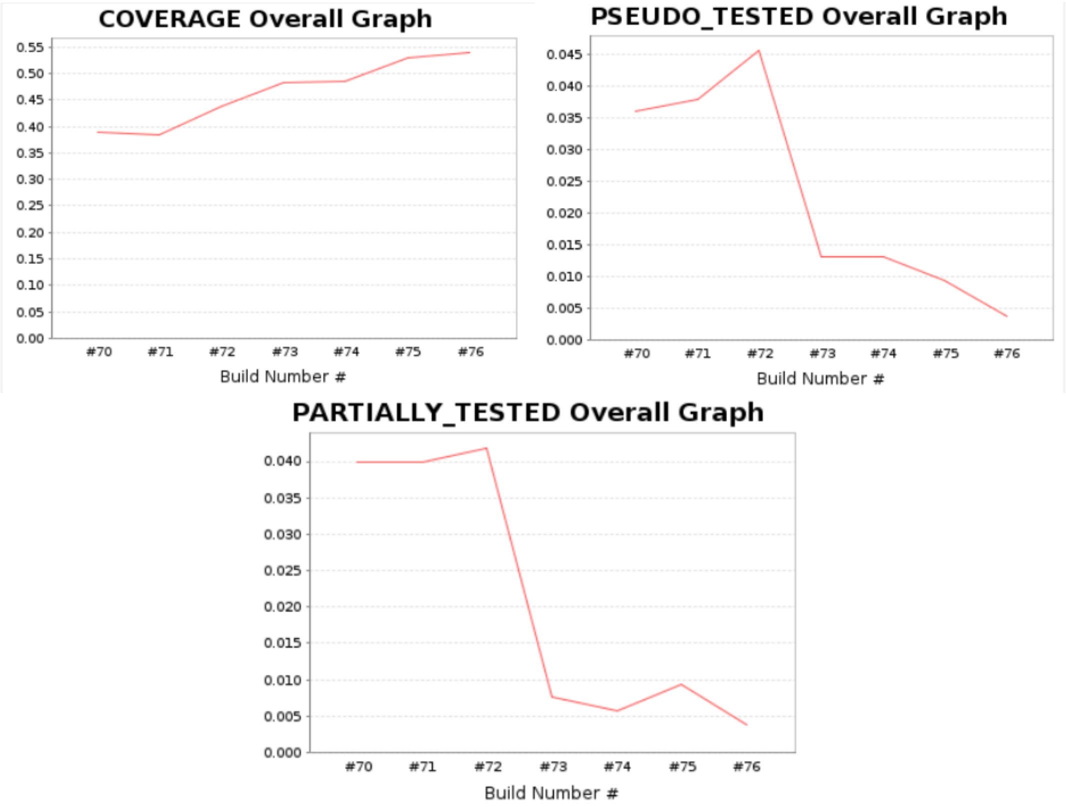


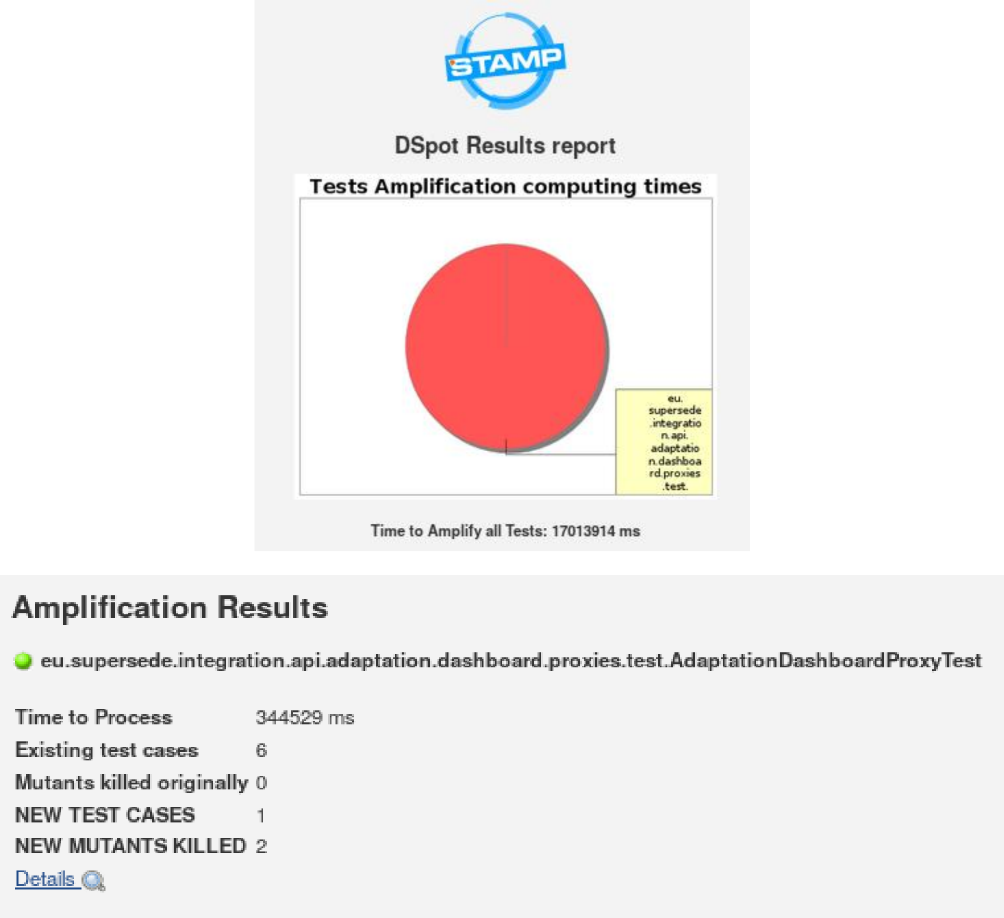
Figure 6 STAMP Descartes Reports for Supersede IF

supersede-if-dspot-script Jenkins job:

This parameterized job[6] executes DSpot related control and treatment tasks (i.e. CCC, TADS) with the following process:

1. Parameterize DSpot execution for flags: --test, --amplifiers, --test-criterion, --iterations
2. Pull IF repository from SCM (GitHub)
3. Build IF project using Maven (skipping tests)
4. Execute DSpot tasks (CCC, TADS)
5. Commit DSpot report into dspot-usecases-output repository
6. Visualize DSpot report using STAMP DSpot Reports plugin for Jenkins. An example of DSpot report is shown in Figure 7, which shows a summary of the statistics of DSpot results for concrete target IF test suite: *AdaptationDashboardProxyTest*.

[6] http://62.14.219.13:7777/job/supersede-if-dspot-script/configure (restricted access)



DSpot Results report

Tests Amplification computing times

Time to Amplify all Tests: 17013914 ms

Amplification Results

● eu.supersede.integration.api.adaptation.dashboard.proxies.test.AdaptationDashboardProxyTest

Time to Process	344529 ms
Existing test cases	6
Mutants killed originally	0
NEW TEST CASES	1
NEW MUTANTS KILLED	2

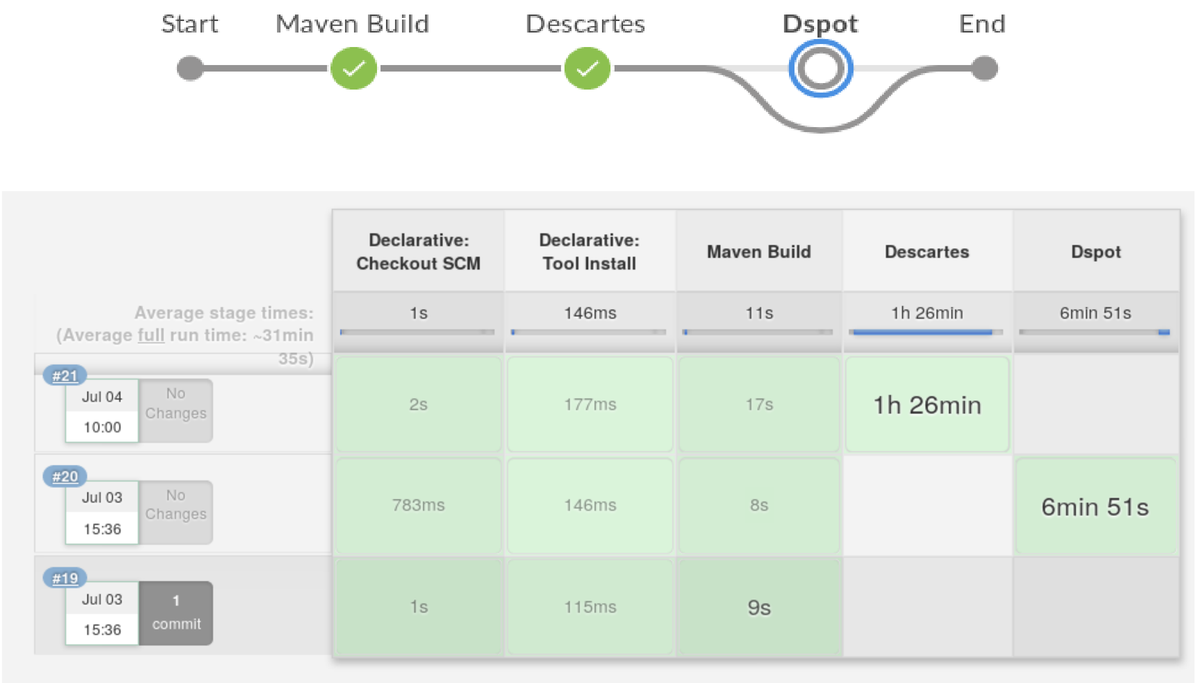
Details

Figure 7 STAMP DSpot Reports for Supersede IF

supersede-if-pipeline Jenkins pipeline:

This parameterized pipeline executes Descartes and DSpot related control and treatment tasks with the following process (see Figure 8 for a graphical overview):

1. Parameterize whether to execute Descartes and/or DSpot tasks
2. Parameterize DSpot execution for flags: --test, --amplifiers, --test-criterion, --iterations
3. Pull IF repository from SCM (GitHub)
4. Build IF project using Maven (skipping tests)
5. Execute Descartes tasks (CMS, CTQ)
6. Execute DSpot tasks (CCC, TADS)
7. Commit Descartes report into descartes-usecases-output repository
8. Commit DSpot report into dspot-usecases-output repository



	Declarative: Checkout SCM	Declarative: Tool Install	Maven Build	Descartes	Dspot
Average stage times: (Average full run time: ~31min 35s)	1s	146ms	11s	1h 26min	6min 51s
#21 Jul 04 10:00 No Changes	2s	177ms	17s	1h 26min	
#20 Jul 03 15:36 No Changes	783ms	146ms	8s		6min 51s
#19 Jul 03 15:36 1 commit	1s	115ms	9s		

Figure 8 Configurable STAMP pipeline for Supersede IF (above) and execution instances (below)

Botsing and RAMP

The experimentation of the STAMP Botsing and RAMP tools has been conducted within a Docker container that isolates the SUT of potential damage caused by Botsing running without the Security Manager protection. As mentioned in D5.6, section 8.C.1, Botsing will not consider to instantiate most of the IF SUT classes because the restrictions imposed by its sandbox (e.g. Security Manager). Therefore, in order to apply Botsing on the IF SUT, we need to disable its sandbox and create a new one using a Docker container that prevents Botsing from causing damage in the environment.

The Botsing image for the IF SUT is described by this Dockerfile and the container instantiated by this run_image.sh script. The IF SUT classpath is generated (within the container) using this create_classpath.sh script. Once created, the Botsing behavioral model that feeds the Botsing crash replicating test generation is generated by the generate_behavioral_models.sh script.

Botsing experiments to generate crash replicating tests for IF production crash exceptions have been conducted by executing the Botsing scripts *run_botsing_exception<XX>.sh* located in this folder (there is a script for each target crash exception).

RAMP experiments to generate test cases for production IF test cases have been conducted using the script run_evosuite.sh, whose parameter *-class "eu.supersede.integration.api.security.test.IFAuthenticationManagerTest"* traverses through the remaining IF test suites.

Use case: CityGO CityDash

Docker configuration

CityGO CityDash sub-components and services are delivered as Docker containers within a composite managed by Docker Compose. We can differentiate two main cases for building and deployment. The main difference is the server configuration. For the first one, Citygo runs over an Apache server and it supports running performance tests (scenario1). For the second one, Citygo runs over Nginx and Gunicorn services and it allows running functional tests (scenario2). Each scenario is managed by different container descriptors:

Docker descriptors for scenario1:

https://github.com/STAMP-project/camp/tree/master/samples/stamp/atos/performance/template

Docker descriptors for scenario2:

https://github.com/STAMP-project/camp/tree/master/samples/stamp/atos/functional/template

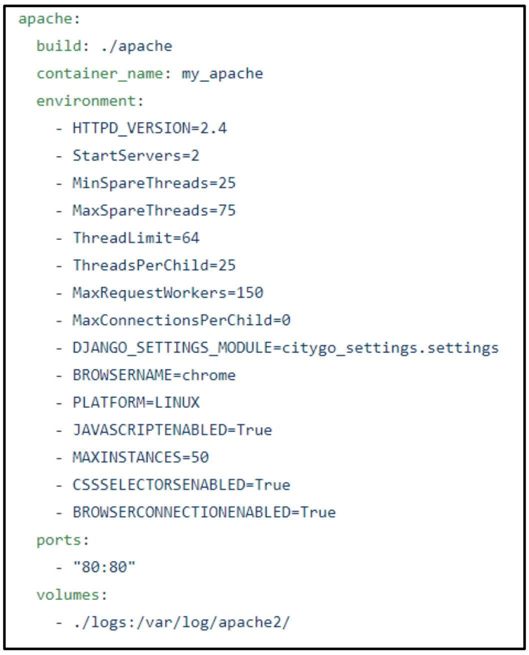
A) Scenario 1: CityGo deployment over Apache

In the first case, CityGo App is deployed inside an Apache container that is responsible to host CitygoDash (e.g. the backend of the CityGo system, see D5.6). See Figure 9 for a snippet of the Apache docker compose.

Below tree shows the structure of the folder that contains the Docker descriptors and other artefacts required to build, deploy and launch all the services of the CityDash system for this scenario configuration:

```
├── camp.yml
├── configurations.png
├── Jenkinsfile
└── template
        ├── apache
        │       ├── demo_site_1.conf
        │       ├── Dockerfile
        │       ├── httpd.conf
        │       ├── mpm_event.conf
        │       └── requirements.txt
        ── browser
        │       ├── citygo.jmx
        │       ├── Dockerfile
        │       └── requirements.txt
        ├── citygo
        │       ├── access.log
        │       ├── backend
        │       ├── citygo_settings
        │       ├── dashboard
        │       ├── DEBUG.log
        │       ├── Dockerfile
        │       ├── entrypoint.sh
        │       ├── init-db.sql
        │       ├── manage.py
        │       ├── requirements.txt
        │       └── static
        ├── docker-compose.yml
```

```
    └── python
          └── Dockerfile
```



```
apache:
  build: ./apache
  container_name: my_apache
  environment:
    - HTTPD_VERSION=2.4
    - StartServers=2
    - MinSpareThreads=25
    - MaxSpareThreads=75
    - ThreadLimit=64
    - ThreadsPerChild=25
    - MaxRequestWorkers=150
    - MaxConnectionsPerChild=0
    - DJANGO_SETTINGS_MODULE=citygo_settings.settings
    - BROWSERNAME=chrome
    - PLATFORM=LINUX
    - JAVASCRIPTENABLED=True
    - MAXINSTANCES=50
    - CSSSELECTORSENABLED=True
    - BROWSERCONNECTIONENABLED=True
  ports:
    - "80:80"
  volumes:
    - ./logs:/var/log/apache2/
```

Figure 9 Case 1- Apache docker compose

The parameterization of the Apache engine for performance is collected in the environment variables of the docker-compose descriptor. During the set up, they are injected into the configuration files, namely the *mpm_event.conf* and *httpd.conf,* of the apache modules that influence the overall performance*. These variables are also collected in the CAMP input template. as ENV variables, in order to generate multiple configurations*.

B) Scenario 2 Citygo deployment over NGINX

Below tree shows the structure of the folder that contains the Docker descriptors and other artefacts required to build, deploy and launch all the services of the CityDash system for this scenario configuration:

```
├── camp.yml
├── configurations.png
├── Jenkinsfile
└── template
    ├── nginx
    │     ├── Dockerfile
    │     └── nginx.tmpl
    ├── browser
    │     ├── citygo.jmx
    │     ├── Dockerfile
    │     └── requirements.txt
    ├── citygo
```

```
│          ├── access.log
│          ├── backend
│          ├── citygo_settings
│          ├── dashboard
│          ├── DEBUG.log
│          ├── Dockerfile
│          ├── entrypoint.sh
│          ├── init-db.sql
│          ├── manage.py
│          ├── requirements.txt
│          └── static
├── docker-compose.yml
└── python
           └── Dockerfile
```

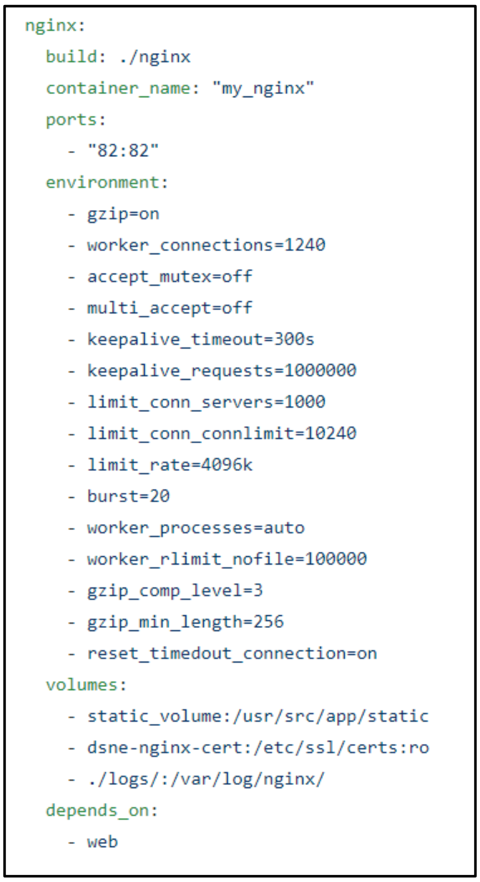
CityGo deployment relies on two services, **Nginx** (see Figure 10 for a Nginx service code fragment) and **Web** docker service (see Figure 11).

Nginx service acts as a proxy. It sends requests to Gunicorn[7] (e.g. a Python HTTP server), which receives and pass them to the CityDash *app* (e.g. the Citygo service in the composite).

The *CityGo* service contains the Python source code of CityGo application[8] hosted in the Gunicorn server. This latter starts a module containing a WSGI application object named citygo.settings.

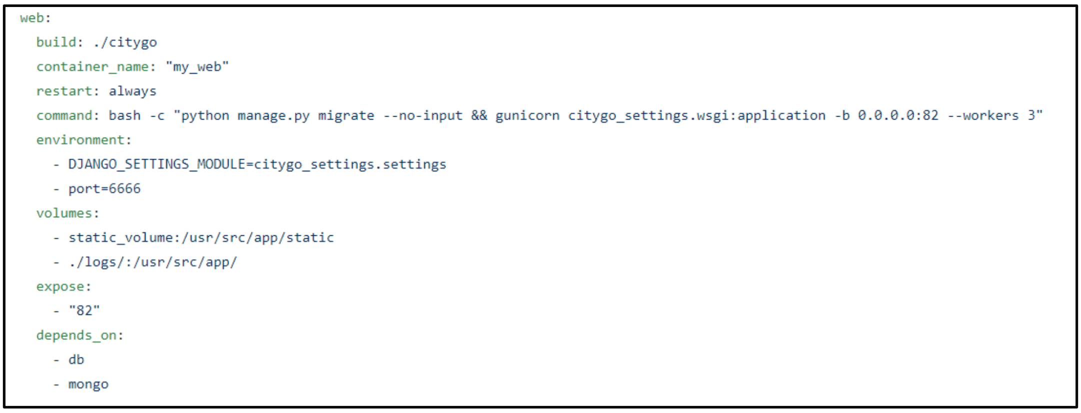
For data persistence, we included a **MongoDB[9]** service. It is used to store data from third party applications. Finally, for running the functional tests on different environments, we included a **Selenium**[10] grid composed by a hub and a node (Figure 12 shows the Selenium Grid services declared in the Docker composite).

[7] https://gunicorn.org/
[8] https://gitlab.atosresearch.eu/ari/stamp_docker_citygoApp/tree/master/dashboard
[9] https://www.mongodb.com/es
[10] https://www.seleniumhq.org/



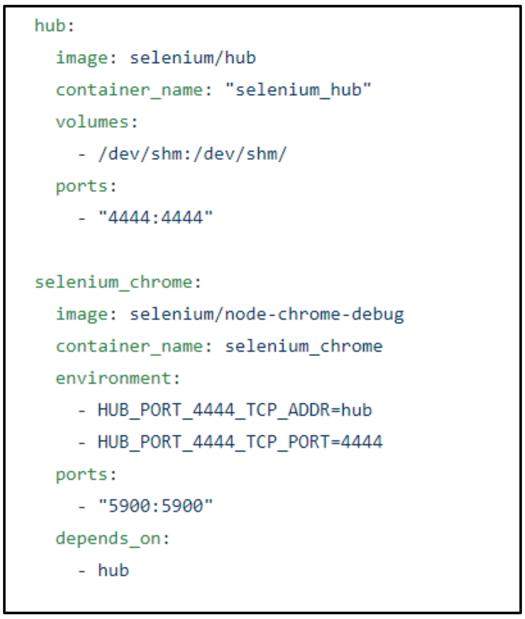
```
nginx:
  build: ./nginx
  container_name: "my_nginx"
  ports:
    - "82:82"
  environment:
    - gzip=on
    - worker_connections=1240
    - accept_mutex=off
    - multi_accept=off
    - keepalive_timeout=300s
    - keepalive_requests=1000000
    - limit_conn_servers=1000
    - limit_conn_connlimit=10240
    - limit_rate=4096k
    - burst=20
    - worker_processes=auto
    - worker_rlimit_nofile=100000
    - gzip_comp_level=3
    - gzip_min_length=256
    - reset_timedout_connection=on
  volumes:
    - static_volume:/usr/src/app/static
    - dsne-nginx-cert:/etc/ssl/certs:ro
    - ./logs/:/var/log/nginx/
  depends_on:
    - web
```

Figure 10 Nginx service



```
web:
  build: ./citygo
  container_name: "my_web"
  restart: always
  command: bash -c "python manage.py migrate --no-input && gunicorn citygo_settings.wsgi:application -b 0.0.0.0:82 --workers 3"
  environment:
    - DJANGO_SETTINGS_MODULE=citygo_settings.settings
    - port=6666
  volumes:
    - static_volume:/usr/src/app/static
    - ./logs/:/usr/src/app/
  expose:
    - "82"
  depends_on:
    - db
    - mongo
```

Figure 11 Web service



```
hub:
  image: selenium/hub
  container_name: "selenium_hub"
  volumes:
    - /dev/shm:/dev/shm/
  ports:
    - "4444:4444"

selenium_chrome:
  image: selenium/node-chrome-debug
  container_name: selenium_chrome
  environment:
    - HUB_PORT_4444_TCP_ADDR=hub
    - HUB_PORT_4444_TCP_PORT=4444
  ports:
    - "5900:5900"
  depends_on:
    - hub
```

Figure 12 Selenium Grid

CityGo Jenkins pipeline:

Both scenarios use the same pipeline stages as shown below (Figure 13).

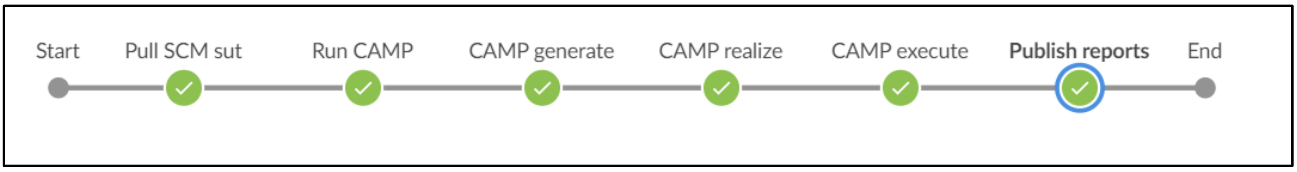


Figure 13 CAMP pipeline

This pipeline applies the CAMP treatment to CityGo CityDash using an instance of the **CAMP**[11] Docker image. The main stages of this pipeline are the following:

1. Pull SCM SUT: Citygo code base is pulled from the SCM repository[12] (Gitlab repository),
2. Run CAMP: a CAMP instance is built and instantiated from its DockerHub image.
3. CAMP generate: Camp generates new configurations,
4. CAMP realize: Camp realize these configuration (e.g. Docker artifacts are generated),
5. CAMP execute: for each new configuration, Camp executes instantiates a CityGo CityDash instance, runs selected tests (i.e. performance, functional) and collect results,
6. Publish reports: it recollects the metrics and log files generated by the tests

Two pipelines have been implemented and deployed within Atos Jenkins CI/CD:

- Pipeline used for scenario 1: (performance tests): https://github.com/STAMP-project/camp-samples/blob/master/atos/performance/Jenkinsfile .
- Pipeline used for scenario 2 (functional tests):

  https://github.com/STAMP-project/camp-samples/blob/master/atos/functional/Jenkinsfile

[11] https://hub.docker.com/r/fchauvel/camp

[12] https://gitlab.atosresearch.eu/ari/stamp_docker_citygoApp/

8.B.3 Validation Method

The validation method adopted by Atos consists on a number of experiments, conducted independently, and suggested in D5.3 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 on the KPIs and to answer our validation questions. The method adopted during the second evaluation period (M12-M24) was described in D5.6. This section updates this method in Table 10, which defines the experiments conducted in this third evaluation period (M24-M30) for each KPI. Tasks are identified by their identifiers (see tables 7 and 8 above).

Table 10 Atos Experiments for KPIs. Sequence of Experimentation Tasks

Experiment	**KPI**	**Use Case**	**Tasks**
Amplification of code coverage	K01	IF	- Preparation of the SUT baseline and STAMP treatments (Setup) - Compute code coverage (CCC) - baseline DSpot treatment: - Apply DSpot to amplify the test suite (TADS) - Compute code coverage (CCC) DSpot treatment is applied to the baseline and after applying the Descartes treatment. Botsing Treatment: - Apply Botsing to amplify the test suite (TAB) - Compute code coverage (CCC) Descartes treatment: *Iteration: - Compute test quality analysis (CTQ) - SUT base/test refactoring based on Descartes report (TF) - Compute code coverage (CCC) Influence of treatments (DSpot, Descartes and Botsing on test coverage) is evaluated separately
Reduction of Flaky Tests	K02	IF	- Preparation of the SUT baseline and STAMP treatments (Setup) - Identify flaky tests by repeatedly executing test suites (IFT)
- Amplification of mutation score - Reduce pseudo/partially tested methods	K03	IF	- Preparation of the SUT baseline and STAMP treatments (Setup) - Compute mutation score (CMS) Descartes treatment: *Iteration: - Compute test quality analysis (CTQ) - SUT base/test refactoring based on Descartes report (TF)

			- Compute mutation score (CMS) DSpot treatment: - Apply DSpot to amplify the test suite (TADS) - Compute test quality analysis (CTQ) - Compute mutation score (CMS) Influence of treatments (DSpot and Descartes on mutation score) is evaluated separately
Amplification of #execution paths	K04	City Dash	- Preparation of the SUT baseline and STAMP treatments (Setup) - Amplification of the SUT deployment configurations with CAMP (TAC) *Iteration over each generated configuration: - Compute the SUT runtime paths during the execution of the stress tests and functional tests (CEP)
Amplification of detection of configuration related bugs	K05	City Dash	- Preparation of the SUT baseline and STAMP treatments (Setup) - Amplification of the SUT deployment configurations with CAMP (TAC) *Iteration over each generated configuration: - Detect runtime failing tests (stress, functional) executed against the SUT deployed with each configuration (DCB) - Collect the number of failing tests. Identify configuration-related bugs
- Amplification of #configurations tested - Reduction SUT deployment	K06	City Dash	- Preparation of the SUT baseline and STAMP treatments (Setup) - Amplification of the SUT deployment configurations with CAMP (TAC) - Compute the number of new configurations generated (CNCT) *Iteration over each generated configuration: - Compute the computation time required to instantiate the SUT under each deployment configuration (CDT)
Amplification of the #crash replicating tests	K08	IF	- Preparation of the SUT baseline and STAMP treatments (Setup) - Crash replicating test generation with Botsing (TAB) - Compute number of generated crash replicating tests (CNCRT)
Amplification of # of production tests	K09	IF	- Preparation of the SUT baseline and STAMP treatments (Setup) - Compute the total number of SUT tests (CNET) - Amplify production tests with RAMP (PTAWES)

8.B.4 Validation Data Collection and Measurement Method

Data/Metrics resulting from the experiment tasks described above are collected by adopting different methods depending on the task's nature that were already described in D5.6:

- Data/Metrics produced as output by the execution of baseline (e.g. Clover, Jacoco, PIT, etc) or STAMP tools (Descartes, DSpot, CAMP, Botsing, RAMP) are pushed onto the corresponding STAMP repositories located in GitHub for collecting industrial use case experiments (see Table 13 in D5.6). Jenkins jobs/pipelines automatically push experiments collected data onto these Git repositories. In the case of experiments conducted manually (using scripts), data pushing is managed by these scripts.
- Code/Test bases modified manually (in reaction to STAMP tools reports) or extended by amplified tests generated from the STAMP tools (DSpot, Botsing, RAMP) are manually pushed into the corresponding Use Case repositories (see Table 14 in D5.6)

8.C Validation Results

8.C.1 KPIs report

This section analyses and discusses the results obtained in the conducted Atos experiments. They are described in Table 8 (see also the original one in D5.6) as a sequence of baseline and treatment tasks. Table 11 shows a summary of these results, focusing on the obtained measures for the STAMP KPIs. Results and detailed analysis for each KPI are presented in the following paragraphs.

Table 11 Summary of Atos KPI measurements

KPI	**Measure**			**Difference with objective**
	Baseline	**Treatment**	**Difference**	
K01-Execution Paths (IF)	**47.7%** (Code Coverage)	**78.7%** (Code Coverage)	**+65.68%**	**+44.68%** (21%)
K02-Flaky Tests (IF)	**0**	**N/A**	No Flaky tests detected (candidates did not pass definition)	**-20%** (20%)
K03-Better Test Quality (IF)	**39%** (mutation score)	**66%** (mutation score)	**69,23%**	**+49,23%** (20%)
K04-More unique traces (CityGo) *performance scenario*	**92** (suboptimal config)	**102** (optimal config)	**10.86%**	**-29.13** (40%)
K04-More unique traces (CityGo) *functional scenario*	**537** (suboptimal config)	**581** (optimal config)	**7.57%**	**-32.43** (40%)

K05-System specific bugs (CityGo) *performance scenario*	**0** (one single default config)	**4** (of 10 configs)	**N/A**[13]	**N/A** (30%)
K05-System specific bugs (CityGo) *functional scenario*	**0** (one single default config)	**1** (of 10 configs)	**N/A**	**N/A** (30%)
K06-More config / Faster (CityGo) *performance scenario*	**1** (#config)	**10**(#config)	**+900%**	**+850** (50%)
K06-More config / Faster (CityGo) *functional scenario*	**1** (#config)	**32**(#config)	**+3100%**	**+3050** (50%)
K08-More crash tests (IF)	**0**	**3 (3/6)**	**50%**	**- 20%** (70%)
K09-More prod tests (IF)	**196**	**2180**	**1012.24%**	**+1002.24** (10%)

K01 - More execution paths

The efficiency of STAMP tools on the IF amplification of code coverage (e.g. our measurements focused on covered code lines rather than on covered branches) is depicted in Table 12.

Table 12 K01: code coverage for IF use case; efficiency of STAMP tools

Treatment	**Code Coverage**	
	Clover	**Jacoco**
Baseline 1 (D5.6)	47.5%	51%
D5.6 Dspot Treatment	52%	54%
D5.7 DSpot amplification #1 (on baseline)	54,9%	56%

[13] The difference (in percentage) cannot be computed since the baseline is 0.

D5.7 DSpot Amplification #2 (on treatment, after Descartes)	53.4%	56%
D5.7 RAMP Amplification (on treatment, after Descartes)	78.3%	71%
D5.7 All treatments	78.7%	71%
Delta (D5.7 vs Baseline)	**65.68%**	**39.21%**

This table shows the coverage measured by Clover and Jacoco tools. Reported coverage is not exactly the same (although equivalent) because both tools use different definitions and computations for code coverage[14]. This table shows a significant STAMP-wise amplification in code coverage, in particular w.r.t. the results reported in previous evaluation (D5.6). DSpot increases the code coverage (from 47% to 54.9%) thanks to the new tests cases it created by amplifying most of the existing IF test suites. Descartes seems to have a negative influence on the overall code coverage, as reported in D5.6, and observed in Table 12 - by comparing the rows for DSpot amplification before and after the Descartes treatment on the base/test code (from 54.9% to 53.4%) -, but the results are not concluding. The most significant code coverage improvement is achieved by applying RAMP test amplification (from 53% to 78%). The combination of all STAMP techniques and tools has achieved an overall code coverage improvement of 65.68%. This achievement is 39.21% better than the objective (21%).

DSpot modest coverage improvement (7.4%) required 50 amplified test suites and 1317 new test cases. DSpot generated test cases for the most interesting IF business logic classes, e.g. its functional proxy classes that dispatch messages to the SUPERSEDE micro-service network. On the contrary, RAMP generated 153 new test suites consisting of a total of 2180 new test cases. However, RAMP could not succeed in generating new test cases for IF business logic, but it did for the helper and model classes. These IF classes were scarcely covered by the existing test cases developed by the IF team; this is the main reason that DSpot (which relies on existing test cases) could not improve coverage for those aspects of IF component. In turn, RAMP was precisely guided by developers to amplify test cases for these IF code aspects, achieving a significant code coverage improvement.

DSpot amplified tests are close to the original ones. An inspection of amplified tests reveals some relevant aspects:

- significant management of unfavorable, unexpected SUT behavior, that covers all the possible exceptions that the SUT can throw under a wrong usage of the IF component interface. This test management of unfavorable behavior was already reported in D5.6 and perceived by the IF development team as a significant improvement on the existing test base,
- amplified tests exercise some of the IF model API that was not tested by the original test cases, although some of the injected IF model-targeting assertions are irrelevant for the purpose of the functional tests,
- there are some duplicated assertions, others are trivial (as they are always true); there are also test cases that duplicates API invocations; many amplified test cases look pretty similar among them,
- unintentionally, DSpot can generate test cases that reproduce some runtime IF exceptions (see K08),
- DSpot is setting up and mocking up correctly the preparation of the test objects,
- DSpot is able to generate amplified test cases exercising favorable (e.g. returning positive SUT responses) behavior, but only in few cases (e.g. for a small set of IF public interface).

RAMP amplified tests could only be generated for IF helper and model packages. See K09 for a more detailed

[14] See this link for tool comparison: https://confluence.atlassian.com/clover/comparison-of-code-coverage-tools-681706101.html

analysis.

K02 - Less flaky tests

The situation of this KPI under the SUPERSEDE IF analysis has not changed since the D5.6 report (Table 13 below). IF is not facing test flakiness. IF tests may eventually fail because of issues encountered in the internal state of the SUPERSEDE SUT or its environment. Therefore, this KPI cannot be evaluated in Atos IF use case.

Table 13 K02 code coverage metrics for IF use case

Metric	Measure Tool	Baseline	Treatment
Number of flaky tests	Manual	0	N/A

K03 - Better test quality

The efficiency of STAMP tools on the IF amplification of mutation score is depicted in Table 14. We have obtained a significant improvement on the IF test ability to detect mutations (e.g. mutation score) with the combined application of STAMP techniques and tools, namely Descartes, DSpot and RAMP. Base/test code refactoring addressing the Descartes issues report have improved the mutation score up to a relative 43.59% (from 39% to 56% mutation score). More modest, but also relevant is the DSpot contribution to the mutation score improvement thanks to its amplified tests that supported the detection of a larger set of mutants: a relative mutation score improvement of 5.66% (from 53% to 56%). Significantly larger is the RAMP contribution, supported by the newly generated production level tests: a relative mutation score improvement of 22.64% (from 53% to 65%). Combining all STAMP treatments, the relative mutation score improvement has been 69.23% (from 39% to 66%). This achievement is 49.23% better than the objective (20%).

Table 14 K03 mutation score metrics for IF use case

Iteration	Mutation score (Descartes)
Baseline 1 (D5.6)	39%
D5.6 Descartes treatment	43%
D5.7 Descartes treatment	53%
D5.7 DSpot treatment (after Descartes)	56%
D5.7 RAMP treatment (after Descartes)	65%
D5.7 All treatments	66 %
Increment (D5.7 - Baseline)	**69.23%**

The number of issues (i.e. pseudo and partially tested) reported by Descartes for IF main/test code base is shown in Table 15, after applying code refactoring. The number of issues dropped down from 41 (in the baseline commit) to 4 (after applying the last Descartes-driven code refactoring). The individual effect of these refactorings in the mutation score is also included in Table 15. This represents a reduction of 90% in the number of detected issues.

Table 15 K03 mutation score and detected issues for IF use case

Iteration	Mutation Score (Descartes)	Issues	
		Pseudo-tested	**Partially tested**
Baseline - 20190701	38%	20	21
Baseline - 20190911	44%	24	22
Code refactoring: 20190925	48%	7	4
Code refactoring: 20190925	48%	7	3
Code refactoring: 20190926	53%	5	5
Code refactoring: 20190927	54%	2	2
Variation	**42%**	**-90%**	**-90%**

Figure 14 shows the STAMP Descartes report for Jenkins that reports the evolution of mutation score for the Supersede IF component.

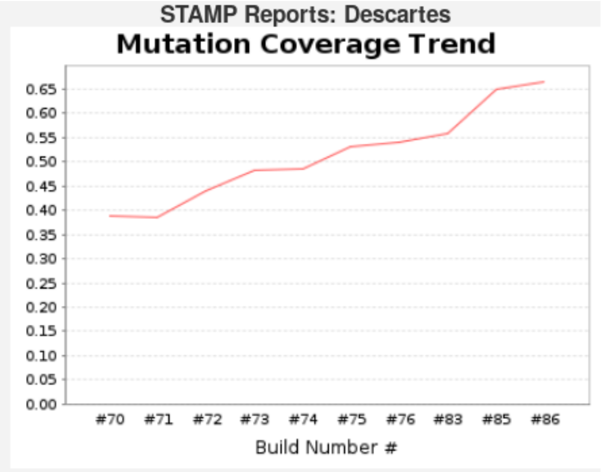


Figure 14 Mutation Score evolution for IF use case reported by STAMP Jenkins plugin

CAMP related KPIS: K04, K05, K06

CAMP technology for amplifying Docker deployment configurations has been applied to the CityDash component in two scenarios:

- A search for an optimal performance scenario (see D5.6), where Apache and NGINX settings are tuned to reach the highest performance, driven by the results of stress tests executed against a CityDash instance launched by CAMP for each generated configuration),
- A search for compatible functional CityDash configurations where variability is introduced at the dependency level (i.e., web containers “Apache” vs “NGINX” and versions of dependencies: Python/Django, Mongo, etc.). The functional compatibility is checked by running the Selenium functional tests.

Performance scenario

A CityDash performance scenario has been conducted using a CAMP pipeline designed for CityDash and realized in our CI/CD Jenkins instance (see section 8.B.2). We have conducted experiments for an increasing number of users (in the range [300, 500]) launching HTTP requests against a CityDash instance launched by CAMP for each generated configuration (Table 15).

Job	#Users	Time spend (s) per configuration
#174	500	990
#176	450	972
#177	400	930
#180	350	1026
#181	300	1032
#182	250	1146
Average		1016 +/- 74

Figure 15 Results of stress test execution for CityDash in each amplified configuration[15]

For each pipeline run, CAMP generates amplified configurations, and for each of them, it instantiates a CityDash instance, launches stress test using BlazeMeter, collects the test results and the CityDash execution traces, and then it generates reports.

[15] Execution time per configuration seems to increase when we decrease the number of concurrent users (which is counterintuitive). This merely a statistical fluctuation that appears because of the small number of experiments run.

Table 16 Results of stress test execution for CityDash in each amplified configuration

Config	% Percentage of errors	#Unique traces	Average Response Time (s)	Has errors after warm up (>0.5%)
1	1.47	650	15.21	NO
2	2.8	96	15.86	YES
3	2.27	359	14.61	YES
4	2.93	102	38.55	NO
5	0.33	99	16.58	NO
6	0.47	94	15.59	NO
7	0.6	92	14.18	NO
8	2.2	94	27.87	NO
9	1.33	96	20.59	YES
10	1.73	95	21.22	YES

Table 16 shows consolidated results of stress tests for each amplified configuration for a run with 300 users, where:

- Percentage of errors indicates the number of browser user's requests CityDash could not manage,
- "Unique traces" indicates the number of unique CityDash traces shown in runtime logs,
- Average response time is the average time CityDash takes to process a single request,
- Has errors after warm up indicates whether or not there are request errors after CityDash start up period. This is a way to discard requests not processed during the initialization of CityDash, before it is responsive. This filter needs to be applied as CAMP executes the stress test right after the launching of the CityDash docker composition, not giving time CityDash to initialize properly. Therefore, only errors after initialization are considered for KPI computation.

The number of unique traces has been calculated by a Shell script that processes the CityDash log (for each test run in a different environment), extract relevant entries, removes duplicates and counts the total number of occurrences.

Functional scenario

CityDash configuration has been amplified by CAMP with regards to their functional requirements, including versions of Python and Django, MongoDB and PostgreSQL. For each new configuration, CAMP instantiated CityDash and launched 6 Selenium tests suites using our citygo_case2_camp[16] Jenkins project pipeline (Table 17).

[16] http://62.14.219.13:7777/job/citygo_case2_camp/

Table 17 Results of functional test execution for CityDash for all amplified configurations

Pipeline #	Number configurations	Execution time (s)	Number failing tests
16	32	15240	1 (1/6 in 1 configuration)
17	32	15360	0

Unique execution traces are also computed (e.gl. using a script similar to the above case) for the 32 configurations tested. Unique traces range from the suboptimal value (537) to the optimal one (581), with a standard deviation: 16.32%. This variation in the number of unique traces executed for each configuration results in a **7.57%**.

Based on these results, we compute the CAMP related KPIs.

K04 - More unique invocation traces

Performance scenario

The main metric associated to this KPI is the total number of unique invocation traces. It is expected that this number will not change across the experiments conducted over the different CityDash amplified configurations, particularly for a monolithic application such as CityDash (see D5.3). Furthermore, stress testing is not expected to modify the SUT tested functionality, not expecting changes in the number of unique traces, hence.

However, Table 18 shows differences in the number of unique traces among the CityDash tested configurations, despite that the SUT was tested with stress tests. Once we reject pathological results (e.g. samples diverging the number of unique traces: configs 1, 3), we can use the remaining ones to compute the associated KPI, using the following interpretation: as baseline situation we select the configuration that shows the lowest number of traces. As a treatment situation, we select the configuration that shows the largest number of traces. In other words, we are using as KPI the maximum variation in the total number of unique SUT traces to estimate the influence of CAMP configuration amplification in the diversity of executed code.

Table 18 K04 More unique invocation traces: metrics for CityDash Use Case in performance scenario

KPI K04-More unique traces	**Measure**		
	Baseline	**Treatment (CAMP)**	**Difference (%)**
Number of unique CityDash traces	**92** (suboptimal config)	**102** (optimal config)	**+10.86%**

As aforementioned, the remarkable conclusion for this KPI is that exercising the performance of CityDash under different configurations shows a significant (above 10%) increment in the total number of unique CityDash test executions, despite its monolithic nature.

Functional scenario

Table 19 shows differences in the number of unique traces obtained from a CityDash instance tested under different configuration settings. They show variability in the versions of CityDash requirements. In this scenario, functional Selenium tests are executed against the CityDash instance. K04 metric has been computed using the same interpretation described above for the performance scenario. The difference obtained in the number of unique invocation traces, **5.57%**, is lower than the one obtained in the performance scenario.

Table 19 K04 More unique invocation traces: metrics for CityDash Use Case in functional scenario

KPI K04-More unique traces	**Measure**		
	Baseline	**Treatment (CAMP)**	**Difference (%)**
Number of unique CityDash traces	**537** (suboptimal config)	**581** (optimal config)	**+5.57%**

K05 - System specific bugs

Performance scenario

Similarly, we processed the results of Table 8.10 to compute the number of system specific bugs. A bug, in this context, is a configuration that results in a CityDash misbehavior. From the results of Table 8.10, we conclude that CityDash shows four failing configurations for a workload scenario of 300 users. On these configurations, CityDash is perceived as failing by an important number of users. Therefore, if we start from a baseline configuration that does not show a misbehavior, we find, thanks to CAMP, four configuration-related bugs (these results are collected in Table 20).

Table 20 K05 System specific bugs: metrics for CityDash Use Case (Performance scenario)

KPI K05-System specific bugs	**Measure**		
	Baseline	**Treatment (CAMP)**	**Difference (%)**
Number of failing configurations	**0**	**4**	**N/A**[17]

In this case, we cannot compute a different percentage as we are starting from a non-failing baseline. In any case, given a particular workload scenario, CAMP can help us to identify valid and failing (i.e. misbehaving) CityDash configurations.

Performance scenario

Functional Selenium tests executed over the 32 amplified configurations (for different runs) detected only one amplified configuration showing a failing test (Table 21). This indicator shows that CityDash is functionally compatible with the different versions of its requirements (Python, Django, MongoDB, PostgreSQL, etc.) that were tested.

[17] We are cannot compute a difference (in percentage) because we are comparing with a zero baseline

Table 21 K05 System specific bugs: metrics for CityDash Use Case (Functional Scenario)

KPI K05-System specific bugs	**Measure**		
	Baseline	**Treatment (CAMP)**	**Difference (%)**
Number of configurations showing failing tests	**0**	**1**	**N/A**

K06 - More Configuration/Faster tests

CAMP experiments applied to the CityDash performance scenario (Table 22) show that up to 10 configurations are generated by CAMP, starting from the only configuration that was specified for CityDash before adopting CAMP. In the case of the functional scenario (Table 23), up to 32 new configurations are generated. According to the CAMP development team, the number of generated configurations cannot be anticipated, because this number results from a stochastic process that depends on the number of features, and their constraints, defined (for CityDash) in the camp.yml file. These features are the Apache/NGINX configuration parameters (for performance scenario) or the ranges of acceptable versions for functional requirements (i.e. Python, Django, etc.). The total CAMP execution time per configuration is also computed. This time consists of the time CAMP spent on: i) generating the images of the CityDash services defined in the composition, ii) starting the CityDash service containers, iii) conducting the performance tests, iv) collecting the results and report. This execution time is compared with the time required to conduct an identical experiment by hand. Overall, KPI6 metrics are collected in Table 22 (Performance scenario) and Table 23 (Functional Scenario).

Table 22 K06 More Configuration/Faster tests: metrics for CityDash Use Case (Performance scenario)

KPI K06-More config / Faster	**Measure**		
	Baseline	**Treatment (CAMP)**	**Difference (%)**
Number of amplified configurations	**1**	**10**	**+900%**
CAMP execute time	**1722**	**1016 s (average per config)**	**-41.00%**

Table 23 K06 More Configuration/Faster tests: metrics for CityDash Use Case (Functional scenario)

KPI K06-More config / Faster	**Measure**		
	Baseline	**Treatment (CAMP) in sec**	**Difference (%)**

Number of amplified configurations	**1**	**32**	**+3100%**
CAMP execute time	**1588**	**476 s (average per config)**	**-70.03%**

The number of CAMP generated configurations can change by tuning the variation points of the configuration and their ranges of acceptable values. Despite the fact that this number is relatively small for the performance scenario, it helps us to identify optimal CityDash configurations for a given workload (e.g. number of user's requests). In the functional scenario, a reasonably good number of configurations have been tested.

CAMP significantly reduces the overall time required to test CityDash under a given configuration. It saves 41% of the computation time in the performance scenario, and 70% in the functional one, compared to the manual instantiation of CityDash and the execution of tests. This is a significant time cost reduction when CityDash owners are exploring the configuration space looking for optimal and functional-compatible configurations for a production environment.

K08 - More crash replicating test cases

Botsing was applied to some of the crash exceptions reported in production logs (see D5.6), in particular to 6 of the 12 exceptions identified in D5.6. The remaining 6 could not be targeted as they were not reproducible in a development environment, either because the IF code base fixed them in recent commits, or because they could not be reproduced offline because the current Supersede platform setup does not offer anymore the services involved in these exceptions. For the 6 target exceptions, Botsing was executed within a Docker container sandbox to limit the side effects of switching off the Botsing Security Manager[18]. Bosting was fed with a pre-computed behavioral model, using probabilities for model pool in the range [0.0, 0.5, 1.0], and configured with different search budget (360, 720 sec) and initial populations [100, 200]. Results are shown in Table 24.

Table 24 K08 Generated crash-replicating test cases: metrics for IF Use Case

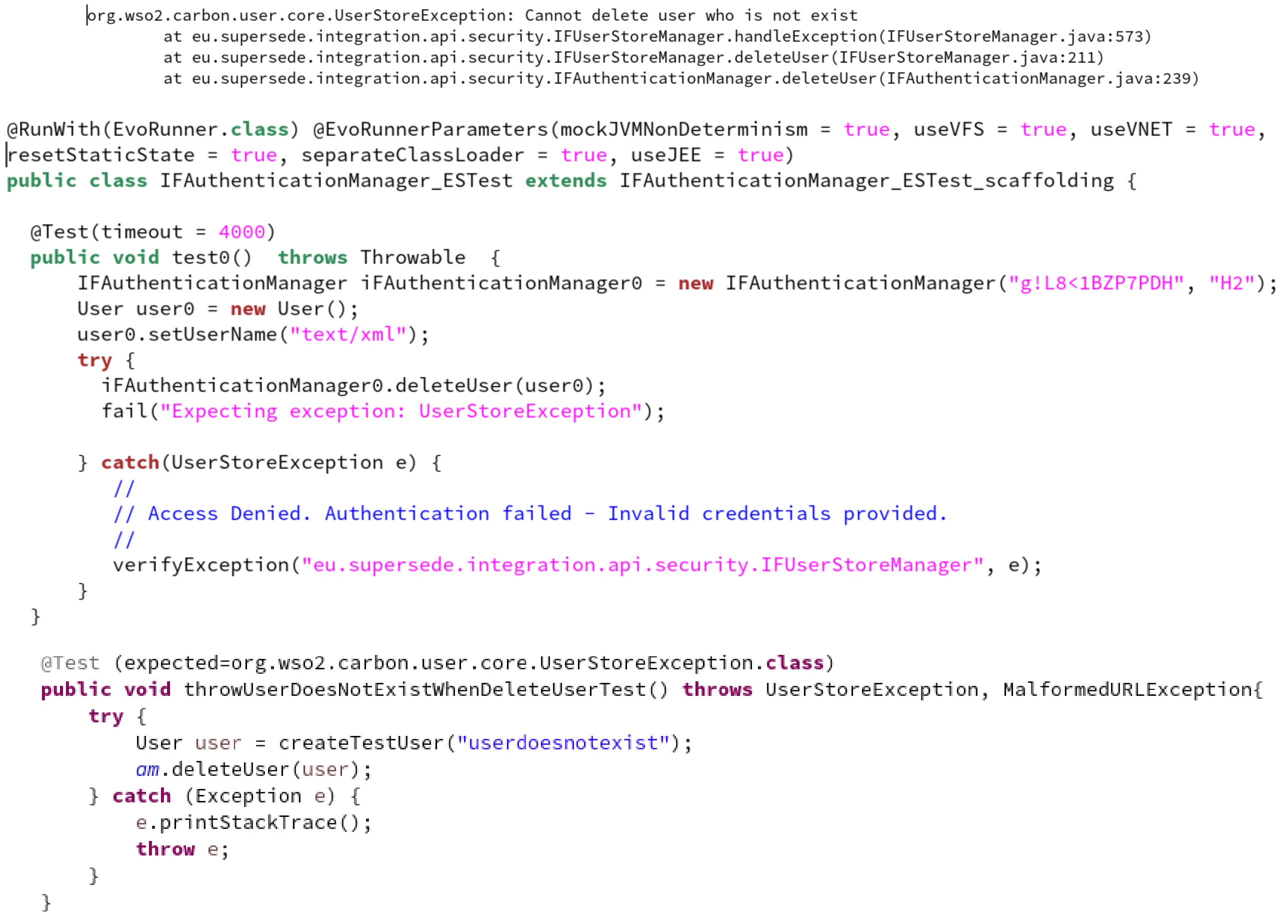
KPI More crash tests	**Measure**		
	Baseline	**Treatment (RAMP)**	**Difference (%)**
Number of crash replicating test cases	**0**	**3 (3/6)**	**50%**

The IF test baseline does not include any test case that pursues the detection of runtime production exceptions. For such reason, we cannot interpret K08 main metric as the one defined in D5.3, as the increment in the number of crash replicating test cases existing in our test base. Otherwise, as we are comparing with a 0 number, we would get an infinite value. In Table 24, we compute this metric as the number of exceptions successfully managed by Botsing (i.e. the number of them for which Botsing computed a replicating test) over the total number of exceptions Botsing targeted. In this sense, Botsing did a remarkable work, as it succeeded in 50% of the cases.

As an example of a crash replicating test generated by Botsing, listing 1 compares a crash replicating test generated by Botsing (middle) for the exception shown on top, with a test manually developed by a member

[18] SecurityManager needs to be disable in Botsing, otherwise it bans the instantiation of the IF proxies involved in inter-service communication that are required to reproduce the target exceptions (see D5.6)

of the IF team, showing the similarities between both tests.



```
org.wso2.carbon.user.core.UserStoreException: Cannot delete user who is not exist
        at eu.supersede.integration.api.security.IFUserStoreManager.handleException(IFUserStoreManager.java:573)
        at eu.supersede.integration.api.security.IFUserStoreManager.deleteUser(IFUserStoreManager.java:211)
        at eu.supersede.integration.api.security.IFAuthenticationManager.deleteUser(IFAuthenticationManager.java:239)
```

```
@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET = true,
resetStaticState = true, separateClassLoader = true, useJEE = true)
public class IFAuthenticationManager_ESTest extends IFAuthenticationManager_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      IFAuthenticationManager iFAuthenticationManager0 = new IFAuthenticationManager("g!L8<1BZP7PDH", "H2");
      User user0 = new User();
      user0.setUserName("text/xml");
      try {
        iFAuthenticationManager0.deleteUser(user0);
        fail("Expecting exception: UserStoreException");

      } catch(UserStoreException e) {
         //
         // Access Denied. Authentication failed - Invalid credentials provided.
         //
         verifyException("eu.supersede.integration.api.security.IFUserStoreManager", e);
      }
  }
```

```
@Test (expected=org.wso2.carbon.user.core.UserStoreException.class)
public void throwUserDoesNotExistWhenDeleteUserTest() throws UserStoreException, MalformedURLException{
    try {
        User user = createTestUser("userdoesnotexist");
        am.deleteUser(user);
    } catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
}
```

Listing 1: Example of crash replicating test generated by Botsing: i) the runtime exception (top), ii) test generated by Botsing (middle), iii) test generated by an IF developer

K09 - More production level test cases

RAMP was applied for each class in the model and the helper packages of the IF component. A set of amplified production level test cases were generated, as shown in Table 25.

Table 25 K09 Amplified production level test cases: metrics for IF Use Case

KPI More production tests	**Measure**		
	Baseline	**Treatment (RAMP)**	**Difference (%)**
Number of test cases	**196**	**2180**	**1012.24%**

RAMP could not generate production level tests for the more interesting IF business classes, those that offer communication proxy capabilities, since RAMP failed in correctly mocking the proxy objects. Nonetheless, RAMP could generate up to 2180 test cases for the IF helper and model packages in more than 2 hours (total computation time was not recorded). RAMP was guided by a behavioral model seeding.

Visual analysis of the generated test classes leads to the following insights:

- many generated tests are asserting uninitialized model class properties,
- quite similar tests are generated, increasing the total number of generated tests,
- tests are simple, asserting few SUT API invocations (in many cases, a single behavior or API invocation),
- some generated tests assert failing behavior, as DSpot did for IF test cases (see K01 report above in this section).

As reported for K01, amplified production level test cases (generated by RAMP) have largely incremented the code coverage. Whether or not these tests can also help test developers for detecting regression bugs has not been confirmed by this evaluation analysis. The number of generated tests is high, with many similar generated tests. This number could be reduced with strategies that identify test similarities and remove duplicated or similar tests.

8.C.2 Qualitative Evaluation and Recommendations

Descartes

Descartes is a very mature tool. The 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, the visual support in the Eclipse IDE, and the DevOps CI/CD support for reporting in Jenkins freestyle jobs and pipelines**.

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

The Descartes issue report is 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 few hours. The main refactorings introduced in IF code and test base, thanks to Descartes were:

- better management of exception triggering
- replacement of void methods so they confirm a successful/failing computation or return the result.
- better assertion checking within IF main API
- returning HTTP codes in inter-service transactions
- verification of message delivery in pub-sub inter-service communication
- improved JSON parameter serializer
- validation of dates and POJOs

Descartes integration with Jenkins CI freestyle jobs and pipelines have largely improved. Descartes reports can be visualized within the job/pipeline dashboard, showing the historic evolution of the mutation score.

Unfortunately, the Descartes extension that offers recommendations for users to refactor their base/test code was not available at the time of this evaluation.

DSpot

The manual inspection of the tests amplified by DSpot provides insights for the Atos SUPERSEDE IF development team to improve the test base. Many amplified tests deliberately inject invalid input values into the API and verify the thrown 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 development phase. DSpot could be applied to most of the packages included into the IF API, with all selected amplifiers, and working with both Jacoco and PIT selectors, unlike the experience reported in D5.6. Moreover, most of the issues reported in DSpot tracker[19], during the second evaluation phase, were addressed by the DSpot development team:

- Remarkably, reported DSpot memory issues were solved so that it could complete its execution on those IF tests suites for which it failed,
- DSpot was reengineered to be more robust and resilient, so that it completed execution despite it could face runtime exceptions that did not prevent it from completing its job. These exceptions are reported to the user at the end of its execution,
- Timeouts that populated our experiments reported in D5.6 were gone, so that DSpot could complete most of our experiments to amplify IF test cases.

Other aspects that have been significantly improved w.r.t the previous evaluation phase, including:

- **performance:** In previous evaluation, DSpot executions took around 3 hours to complete for a single run, with few selected amplifiers. The new release evaluated in this period ran over the same IF target test cases for no more than 2 hours in the most time-consuming case.
- **operability**: It offers more client interfaces to end users, including Eclipse IDE, Maven and Jenkins interfaces for execution and reporting (for freestyle jobs and pipelines), and a significantly higher number of configuration parameters. Documentation has been largely improved.
- **efficiency**: DSpot capacity to increase the SUT code coverage has been improved (see K01 report) with a lower number of generated test cases. In D5.6, DSpot needed to generate up to 1127 new test cases to achieve the code coverage improvement reported in D5.6. In this reporting period, DSpot generated (for more selected amplifiers) 564 new test cases, improving the code coverage as reported in K01 subsection.

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 effectively tune the CityGO service deployment (e.g. Web Container: Apache, NGINX, or the database: PostgreSQL), satisfying expressed constraints, in order to search for optimal configurations.

CAMP does explore the parameters solution space to find optimal configurations with regards to a target optimization problem (i.e. performance), although it is not driven by an objective (or fitting) function model, so the identification of optimal configuration remains a manual task. **In this sense, CAMP is still useful, but the analysis of the solution space remains a manual, tedious and time-consuming task**.

CAMP has also been applied to the generation of deployment configurations that change their functional requirements, replacing some baseline dependencies with other functional equivalente ones. It has been able to detect one incompatible configuration that breaks the functional behavior. **CAMP proves to be quite useful to explore the requirements space to find out functional compatible configurations**.

CAMP management and generation of Docker configurations is fast and precise. CAMP now supports the instantiation of CityGO for a given number of amplified configurations, and this feature has been integrated within our CI/CD process. However, this integration has been more complex than expected, requiring significantly more resources than the ones that were initially planned and allocated. Nonetheless, once the integration was complete, their application in CityDash CI/CD workflow was conducted smoothly.

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

[19] https://github.com/STAMP-project/dspot/issues/created_by/jesus-gorronogoitia

usage.

There is an aspect where CAMP needs to be improved. The application of CAMP to any use case and scenario requires the specification of an input model "camp.yml" that defines the variability points and strategy. The generation of this YAML file is still a manual, complex, prone-to-error task that requires the CAMP team to provide support to the user.

Botsing and RAMP

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 about the causes of the issue. They could also speed up the development of patches.

Botsing succeeded in the generation of the crash replicating test cases for some (i.e. 50%) of the production runtime exceptions it was targeted, once we disabled the Botsing sandbox (e.g. Security Manager) within an isolation container, and we adopted its new model seeding feature. Indeed, **Botsing was significantly fast on test generation**. This feature is quite appreciated by our developers as this tool may help them to easily reproduce and understand occurring failures.

Botsing documentation has been largely improved compared to our previous evaluation. It is much more precise and complete about its usage and includes quite descriptive examples (including the usage of model seeding). The **Botsing interface is straightforward, easy to understand and use** by users. Additionally, its **Eclipse IDE support largely simplifies its adoption** for projects developed within that IDE.

RAMP, using behavioral model seeding, has proven very useful to automate the generation of test cases for IF model and helper classes, largely improving the IF code coverage. However, generated test cases look pretty simple and RAMP was not capable to generate test cases for the IF proxies, which are the most interesting IF business logic classes.

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?

Yes, they can.

In *IF component use case*, we have obtained a significant improvement, **65.68%**, on the number of tested execution paths, measured through the code coverage metric. This is achieved after applying three STAMP treatments: a) test and code base refactoring for addressing the Descartes test quality report, b) the test base amplification with DSpot, and c) the test base amplification with RAMP. We have also got an increment in the number of crash replicating tests generated by Botsing, but this number is quite small (3 new tests of 6 targeted exceptions) that we have not included them in the test base that we used to compute the coverage increase.

In our opinion, the most relevant STAMP technology for addressing this VQ1 is the combination of DSpot and RAMP, particularly the latter. DSpot and RAMP are still producing a significant high number of (quite similar) amplified test cases, though. In the case of RAMP, these tests look rather simple, but we acknowledge the high potential of its behavioral model seeding approach to increase the level of richness of the generated tests. A relevant aspect that have not been addressed unfortunately, it the ability of this amplified tests to reduce the occurrence of regression tests. This would require STAMP tools to generate test cases similar to those authored by human testers, so that they could be incorporated into the master test branch.

For *CityGO CityDash use case*, however, as we argued in D5.6, we cannot consider that the increment we have got for the number of unique invocation traces in CityGO CityDash use case, namely 10.86%, 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 the *IF component use case*, we have obtained significant STAMP technique-driven assistance to increase the observed quality of our test cases on what concerns their test verdict confidence. We could increase the mutation score up to **69,23%** with the combined STAMP treatment for addressing the reported Descartes issues and amplifying the test base with DSpot and RAMP. In addition, the number of method issues reported by Descartes (they give a direct metric on the quality of the test and code bases) decreases **90%** for the number of pseudo and partially tested methods. 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?

Yes, they can

The *CityGO CityDash use case* depends on its execution environment to improve the experience perceived by users, both through the CityDash portal and the Android App. In this context, the STAMP CAMP tool offers great assistance for the optimal configuration of the CityGO backend, increasing its team confidence during its delivery and maintained operation.

We have got good results to increase our confidence in CityDash under different workloads (e.g. number of concurrent users) requiring optimal configurations. First, we observe significant variability (up to **10.86%**) on the number of unique invocation traces, depending on the CityDash deployment configurations tested. Similarly, we found faulty configurations[20] - four (of ten) in the performance scenario; one (of thirty two) in the functional scenario. Assisted by CAMP, we can classify configurations between those that are reliable vs those that are not. This feature helps us to determine optimal deployment configurations for a given workload. Similarly, CAMP detected 1 (of 32) functional incompatible configuration that broke the functional testing. This facility can increase developers' confidence on hosting CityDash on compatible environments.

Finally CAMP largely simplifies 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: CAMP can manage the instantiation of CityDash and the execution of tests for each generated configuration; and this process can be integrated within our CI/CD pipeline, so verification of the compatibility of CityDash under different environments can be automated.

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

Partially yes

In the case of the CityGO CityDash use case, the execution of functional and stress tests can be fully automated by CAMP in a Jenkins pipeline, so that this process is executed much faster than when it is triggered manually. Even more useful is the way test results are collected by the pipeline and stored for each configuration, what speed up the analysis of results.

In case of the IF use case, we could generate some crash replicating test cases. This for sure offers some assistance to developers to speed up the generation of test cases for some exceptional situations, in particular for those developers with less familiarized with IF. However, this is only true for a few cases, and it cannot be generalized for any potential runtime exception. Nonetheless, we acknowledge the benefits of adopting Botsing

[20] i.e. understood as a configuration that shows unprocessed requests when CityDash is exposed to the workload

for fast test generation.

RAMP has proven quite fast and efficient in the generation of new test cases for IF model and helper classes, but it failed to generate them for the more interesting IF proxy business classes. Besides, generated test cases are rather simple executions of a few class API methods.

8.D Global Takeaway

Atos has got significant benefits participating in the development and adoption of the STAMP technologies and tools. They all show a great potential impact on the QA of the Atos Research & Innovation (ARI) software development lifecycle methodology, and as such, they have been promoted within ARI internal standardization of this process[21]. The benefits of STAMP adoption in ARI can be classified into two groups:

- Know-how
- Improvements in software development methodology

Know-how

The exposure of Atos team to TDD related technologies have promoted them among the Atos team and ARI. Before the participation on STAMP, most of ARI development teams were neither adopting TDD nor any test-driven methodology, without writing unit tests, except in few cases. Most of our QA testing culture was targeting the development of integration tests (exclusively in the Java realm) and usability tests (i.e. for Web development using Selenium, in the AJAX realm). There was no wide-spread culture about TDD, advance test assertion, test amplification, code coverage, mutation score, etc. Now, after the participation in STAMP, this team has acquired a significant knowledge on these topics. Besides, thanks to STAMP results applied to Atos UC this team has learnt strategies to improve our test bases and has acknowledged the importance of measuring QA indicators such as mutation score and code coverage to strengthen our test base in order to reduce the occurrence of regression bugs.

Improvements in software development methodology

There are two main STAMP technologies that are promoted within ARI internal methodology because of its straightforward application: Descartes' test quality report and CAMP test execution support against multiple environments. Descartes test quality management has proven quite useful for Atos team to improve our test base and to increase the mutation score. It has been incorporated to our pipelines since, showing low integration and operational costs. CAMP is quite useful for Atos to find optimal configurations of target systems for production environments and to identify functionally compatible software infrastructures.

Less clear is our adoption of test amplification technologies based on DSpot and RAMP. Nonetheless, thanks to DSpot, we learn to improve our test base to manage unexpected behavior, so that related recommendations to reinforce test bases will be included within our methodology. Less clear is the impact of RAMP on test amplification, despite its success on amplifying code coverage, because generated tests are rather simple. Nevertheless, we acknowledge its high potential to assist test coders in amplifying our test base, so that this technology will be reevaluated once it gets mature. Similarly, the Botsing capacity to generate crash replicating tests is perceived as very promising, despite the modest results obtained after its application in Atos use cases.

[21] This internal task force was initiated in 2017, being participated by one of the Atos members of the STAMP team, and it is currently ongoing, concluding recommendations for ARI software development methodologies.

9. Tellu Use Case Validation

Tellu Use Case Highlights

- Descartes is working well on TelluCloud unit tests, and has led to fixes of many important issues of pseudo-testing (34 issues fixed). It has been included in the TelluCloud DevOps process.
- DSpot automatic test amplification produced larger increase in coverage and mutation score than the manual fixing of Descartes issues. Both DSpot and RAMP show that amplification is feasible.
- The use case projects saw very significant increases in code coverage (44,5% increase) and mutation score (45,5% increase) thanks to STAMP work and tools (Descartes, DSpot, Botsing and RAMP).
- CAMP has been used to amplify both Docker configurations for a single service instance and Kubernetes configurations for cloud deployments of multiple instances. It greatly speeds up the process of deploying and testing a Docker image (2300%). It has let us explore the configuration space for cloud deployments, finding a 495% performance increase.

9.A. Use Case Description

9.A.1 Target Software

Tellu's use case is unchanged from the previous validation phase, and the description given in D5.6 is for the most part still accurate. We give a brief summary and update of the target software. The use case is the Java code for Tellu's TelluCloud platform, a cloud platform for collecting and processing data, mainly used within IoT domains. For unit test work, we selected three Maven projects, which represent different levels in the source code hierarchy. An updated description of the projects:

- TelluLib: A library of a general nature, for use in any Tellu project. All TelluCloud modules have a dependency on this code. The code is simple and well-suited to unit testing. It had a good number of unit tests to begin with, which runs quickly, so it is a natural starting point for unit test experiments.
- Core: TelluCloud is made up of micro-services. The Core library contains common code used by a number of the micro-services, primarily a common domain model and data access levels. It is still the largest of all TelluCloud projects (in lines of code). It has some test coverage, but lower than TelluLib, and some of the tests are actually integration tests which take some time to execute.
- FilterStore: This is one of the micro-services of TelluCloud - arguably the most complex of them. However, the project contains little code, and depends heavily on Core. After the previous round of experiments, some code was refactured out of this project, and this was the code best suited to unit testing. We are left with a project with little code of its own and low potential for unit tests, but we still find it useful to include an executable project which goes together with the other two to make up a micro-service.

TelluLib and Core have been very stable since the experiments reported in D5.6, with minor additions to the code. FilterStore has lost some of its code, but it is such a small project in any case, and not suitable for unit testing, so this has little impact on the overall stats.

Updated source code metrics, for the revision used for most of the new experiments are presented in Table 26 (baseline details are given for each experiment in the KPI report).

Table 26 Tellu use case source code metrics

	TelluLib	Core	FilterStore
Java classes	113	375	17
Lines of Code	8245	27462	1777
Unit tests	77	83	2
Coverage	51,9%	26,2%	39,4%
Mutation score	49%	14%	54%

9.A.2 Experimentation Environment

The Tellu development environment is mainly as described in D5.6, with Maven as build system and Jenkins for CI. Windows 10 has been the primary platform for running unit test experiments. Due to compatibility issues, the DSpot experiments are running on Linux in this final phase, while Windows continues to be the platform for non-CAMP experiments.

Non-STAMP tools used in validation:

Apache Maven 3.5.4: https://maven.apache.org/

Eclipse Photon: http://www.eclipse.org/

Clover: http://openclover.org/index

TelluCloud consist of multiple Docker Components that have different responsibilities. In the case of the CAMP experiments we have used the components **Web** and **Actions**. The components have been slightly changed to allow better tracing of the tests.

The web component is the frontend of TelluCloud, and also support a rest API interface for interacting with TelluCloud. The Actions component execute various actions for TelluCloud, such as notifying users through email, sms, notifying systems or creating alarms. For the CAMP experiments we have focused on creating alarms. The Actions component has been modified to add the identity of the pod that creates an alarm to strengthen traceability. The web have been given functionality to create a huge number of alarms in a single request.

The tests are executed in one of Tellu's System Test environments, where the components are deployed in Kubernetes. Results have been computed using javascript testing framework jasmine-node and frisby.

9.A.3 Expected Improvements

Expected Improvements were discussed in D5.6, and this is still valid. More specifically for this final phase of validation, we expect to be able to generate new unit tests and see significant improvements in test coverage. For configuration amplification, we expect to be able to amplify Kubernetes configuration for deployment of the micro-services.

9.A.4 Business Relevance

Automation of testing continues to be very relevant for Tellu. STAMP tools have helped to ensure the quality of the TelluCloud code. In the course of the project, the micro-service version of TelluCloud has been deployed commercially and further developed, now having been deployed in three different cloud infrastructures. It will continue to be deployed in new types of cloud infrastructure, and automated testing is more important than ever. Tellu continues to be dedicated to DevOps, and sees the vital importance of automated testing.

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. A table of treatments based on KPIs was presented in D5.6. This table has been revised (Table 27, below), with new rows for KPI K09 and the separation of global and tool-specific mutation score improvement, as well as refinements in treatments. Some treatments were final in D5.6, while most have been continuing and will be reported here. The details of the tasks are described in the next sections.

Table 27 Validation treatment overview

Metric	KPI	STAMP tool	Treatment for measuring
Global Coverage	K01	All	Code Coverage is computed using Clover, on each of the three selected use case projects. At various times during the project, we compute the percentage of uncovered code. At the end, we can compute the total reduction in uncovered code, after merging in all effects of tool experiments.
Tool-specific coverage improvement	K01	Descartes DSpot Botsing	Code Coverage is computed using Clover, on each of the three selected use case projects. ● A baseline is measured on each use case project before a tool experiment. ● Coverage is measured after tool usage, on each use case project where the tool produced results. Each tool is applied to the same baseline, so effects are not cumulative until they are merged for global coverage.
#Flaky Tests	K02	-	Based on a test suite with flakey tests: ● 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.
Global Mutation Score	K03	All	Mutation score is computed using PIT and STAMP Descartes tools, on each of the three selected use case projects. As with global coverage, these are computed at various times during the project, measuring the final improvement at the end which includes all effects of STAMP work.
Tool-specific Mutation Score improvement	K03	Descartes DSpot	As with tool-specific coverage, this is based on running experiments with one tool and measuring the

		Botsing	effect of that one tool. Mutation score is computed using PIT and STAMP Descartes tools, on each of the use case projects where the experiment cause changes. ● A baseline is measured on each use case project before a tool experiment. ● Mutation score is measured after tool usage. Each tool is applied to the same baseline, so effects are not cumulative until the merge for global score.
#Pseudo Tests #Partial tested tests	K03	Descartes	Count the issues reported by Descartes on each use case project, categorize the issues and try to fix them.
#Execution Paths	K04	CAMP	Track messages through the system, finding the variability in message paths based on amplified configurations.
#configuration related bugs	K05	CAMP	Identify and count bugs which are found based on amplified configurations.
#configurations tested	K06	CAMP	Count number of configurations generated by CAMP which are tested.
Time to deploy SUT in configuration	K06	CAMP	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. ● Count number of stack traces. ● Attempt to replicate crashes with Botsing, counting the number of replicating tests. Experiments with production traces were done and described for D5.6. Experiments with introduced errors are done and described for D5.7.
#production level tests	K09	RAMP	Count models generated by RAMP model generation, and tests generated based on these models.

9. B.2 Validation Target Objects and Tasks

From the validation treatments, we get two types of tasks. One type is tool-specific experiments, where we evaluate a specific STAMP tool. We also have some measurements which are not based on one tool. These are for global coverage and mutation score, where we measure the total effect of all project activity, and flaky tests (K02) which are not addressed by any of the main STAMP tools. On the top level, we have the following main tasks, revised from D5.6:

- Overall code coverage and mutation score
- Descartes testing and validation*
- DSpot testing and validation
- CAMP testing and validation
- Botsing crash replication testing and validation

- Testing and validation of RAMP test generation
- Flaky test experiment*

Task with (*) were finished in D5.6 and are not described in detail here. Most other tasks continue on from the previous iteration reported in D5.6, while the RAMP task is new.

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 subtasks and methodology details will be described in the following sections.

As before, we have three use case projects for unit tests: TelluLib, Core and FilterStore. Test generation experiments are conducted on these three projects. TelluLib and Core both have substantial amounts of code and tests, and are well-suited for the experiments. The FilterStore project is high-level and does not have many unit tests of its own. It will also be considered in each task, but some tasks may be less relevant here. CAMP and flaky tests target the TelluCloud system as a whole. Botsing crash replication also targets the system as a whole, although selected experiments primarily target the selected projects.

9.B.3 Validation Method

Detailed method descriptions for each of the main tasks.

Validation method - Overall code coverage and mutation score

This task continues as described in D5.6, with the addition that we now formally include mutation score. Both coverage and mutation score have in any case been collected together for each of the three use case projects at each measurement time. The goal of the task is to capture the overall unit test improvements from the STAMP activities. As baseline, we use the first measurements collected once the use case projects had stabilized. Since then, these projects have been stable, with very little change to the code outside of STAMP work.

Changes done by specific STAMP tools are captured in the tool experiments. Some of these improvements may be overlapping, as two different tools can generate a test which adds coverage to the same method. Results of tool experiments are kept on separate branches. All will be merged at the end of the project, before measuring the final code coverage and mutation score of each project. STAMP work also involves improvements which are not necessarily captured in tool experiments, such as manual improvements in code and tests to support STAMP toolchains. The global scores will reflect all STAMP-related changes.

The same tools are used in the final period, to make sure results are comparable. The Eclipse plugin version of OpenClover 4.3.1 computes coverage. For mutation score, we have standardized on PIT with the Descartes engine, running from Maven. KPI K03 targets a reduction in the non-covered code, which means we also want the inverse of the coverage, to calculate this reduction (we take a code coverage of 40% to mean 60% uncovered code).

While we compute the global code coverage and mutation scores for each of the three use case projects, we also want to compute a single value for K01 and K03 respectively, to measure the sum of the improvements on the use case as a whole. This makes sense in our case, since the three projects are very much related, being part of the same TelluCloud system and being compiled together. We have the number of lines of code in each project and their sum. We will assume for this computation that a 40% code coverage in one project means that 40% of the lines are covered and the rest are uncovered. This way we get a sum of covered and uncovered code across the three projects (it is basically an average weighted by the project sizes in NCLOC). For an aggregate mutation score, we will likewise compute an average weighted by the lines of code.

Validation method - Descartes

The method of the main validation is described in the corresponding section in D5.6. In summary, we run Descartes on the use case projects, find pseudo and partially tested methods, analyse and categorize these issues, and fix relevant issues manually. Mutation score and coverage are measured before and after. This main validation of Descartes was completed in the previous phase and reported in D5.6. We fixed the relevant issues and found the tool useful enough to include it in the Tellu DevOps process when developing new tests. Since there have not been any significant changes in the TelluLib or Core projects, there is no reason to repeat the experiments.

However, we continue to use Descartes to compute mutation scores and find issues in the other tool experiments. In this way we validate the latest version of Descartes. And in the DSpot validation we make a comparison between the manual fixes done based on Descartes reports and the automated test improvements made by DSpot.

Validation method - DSpot

The method described for DSpot validation in D5.6 continues to be valid. In the previous iteration, we were not able to generate tests with DSpot, and we had a number of issues with running it on our developer systems. We were only able to run on one of three Windows machines, and there we encountered various issues. We therefore spent the first part of the final phase addressing Windows compatibility, doing more testing and looking for solutions. The machine where DSpot did run had slightly older versions of Maven and Java than the others. Java 8 is used for TelluCloud. Upgrading from 1.8.0_121 to 1.8.0_202 broke DSpot on this last system. We concluded that a Java + Windows system is no longer able to run DSpot in its current form. Newer versions of DSpot have been tested, up to the end of this phase, without improving the situation.

DSpot experiments were moved to Linux and a new developer without prior participation in the STAMP project. We then repeated the experiments, varying the selectors and amplifiers on a limited set of tests, before using what we learned to apply DSpot to a broader scope of tests. As in other experiments, mutation score and coverage are measured before and after.

The dspot.properties file is as before, limiting the scope to no.tellu.*. Notes on other key arguments, updated from D5.6:

- time-out: Specify a timeout of 1000 (1 second), unless the experiment specifies otherwise.
- Test class: Most experiments are limited to a specific test class, chosen because interesting and fixable issues were found by it in the Descartes experiment.
- PitMutantScoreSelector engine: Using the default, which is Descartes.
- 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 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 do experiments testing each amplifier in turn, as well as experiments combining multiple amplifiers.
- jvmArgs: It is possible to increase the amount of memory available to Java processes. Something to try if we have trouble getting an experiment to work.

The following tables define the experiments. One experiment consists 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 and output from each run.

The first set of experiments, detailed in Table 28, explore the DSpot arguments, to see the effect of the different selectors and amplifiers on TelluCloud code. These are conducted on the test class ModelTest, which is the largest and most interesting test class in the TelluLib project, and which has several issues found by Descartes.

Table 28 DSpot experiments on ModelTest class

Experiment name	**Arguments / Description**
ModelTest-Selectors	Testing different selectors (test-criterion) on a single test class ModelTest. --test no.tellu.lib.data.model.ModelTest --test-criterion <selector>: ● PitMutantScoreSelector (default) ● JacocoCoverageSelector ● TakeAllSelector
ModelTest-Pit-Amplifiers	Using default selector PitMutantScoreSelector (Descartes), trying different amplifiers. --test no.tellu.lib.data.model.ModelTest --amplifiers <amp>: ● Default (no argument) ● MethodAdd ● MethodDuplicationAmplifier ● MethodRemove ● FastLiteralAmplifier ● TestDataMutator ● MethodGeneratorAmplifier ● MethodAdderOnExistingObjectsAmplifier ● ReturnValueAmplifier ● AllLiteralAmplifiers ● NullifierAmplifier Skip some if this takes a lot of time.
ModelTest-Pit-MultiAmplifiers	As with Amplifiers, but test combinations of amplifiers. --test no.tellu.lib.data.model.ModelTest --amplifiers <amps> For each run, select a set of at least two amplifiers, such as "--amplifiers MethodAdd:MethodRemove". Try at least four runs with different combinations. Select amplifiers based on the results of previous experiment, favoring amplifiers which produce tests and amplifiers which run fast.

The next set of experiments (Table 29) try to generate tests for the rest of the TelluLib project, focusing on the main test classes. Based on the experiments on ModelTest, select one or two sets of arguments to use in these experiments, such as a specific combination of amplifiers, which seem to give good results without taking too long.

Table 29 DSpot experiments on other TelluLib test classes

Experiment name	**Arguments / Description**
JsonTest	--test no.tellu.lib.data.serialize.JsonTest
TemplateTest	--test no.tellu.lib.data.template.TemplateTest

XmlTest	--test no.tellu.lib.data.serialize.XmlTest
TelluLibAll	One run of DSpot without limiting it to a specific test class, meaning it will run on all. This may take a long time. Select a set of arguments based on the other experiments, selecting something which is fast but hopefully can generate some tests.

Based on experiences with the TelluLib experiments, we do similar experiments for some test classes in the Core project. For each experiment, try one or a few combinations of arguments which were found to work best in the TelluLib experiments, in order to try to generate tests. Details in Table 30.

Table 30 DSpot experiments on Core

Experiment name	**Arguments / Description**
JsonUtilTest	--test no.tellu.findit.util.JsonUtilTest Try one or two different (combinations of) arguments.
RuleTemplateSubstituterTest	--test no.tellu.findit.util.RuleTemplateSubstituterTest Try one or two different (combinations of) arguments.
DatabaseServiceImplTest*	--test no.tellu.findit.services.DatabaseServiceImplTest --timeOut 2000 Try one or two different (combinations of) arguments.
AddressLookupServiceTest*	--test no.tellu.findit.services.AddressLookupServiceTest --timeOut 2000 Try one or two different (combinations of) arguments.

* These are not really unit tests, so DSpot may take too long/not work.

For each run within each experiment, we start from the same baseline, and add any generated tests to the test suite. We log the time taken by DSpot, the reported number of tests generated and additional mutants killed, and the resulting coverage. The changes are reverted for each run. After completing the experiments, we will augment the production test suite with generated tests. Where different runs produce different sets of new tests for the same test class, we select the best results in terms of coverage increase. If the experiments are successful, we will then have improved test suites for the use case projects. We compute coverage and mutation score for the result, and their improvements.

We will also compare the DSpot results to those achieved with Descartes. We will compare the coverage and mutation score improvements achieved by the automatic amplification of DSpot with that achieved with the manual fixes done to address Descartes' issues. We will study the issue reports to see how many pseudo- and partially tested cases DSpot is able to fix.

Validation method - CAMP

In the previous validation phase, reported in D5.6, CAMP was validated for Docker configuration amplification. CAMP worked on docker-compose files, providing images with different database backends and allowing these to be easily deployed and tested. This allowed us to find additional system-specific bugs (K05) and to greatly speed up the process of testing different configurations (K06). The TelluCloud use case does not have much variability on the docker-compose level. The Docker containers are usually deployed in cloud infrastructure with Kubernetes, and the Kubernetes yaml files hold parameters to configure the cloud deployment. Tellu has worked with SINTEF to get CAMP to amplify Kubernetes files, and this is the target for validation in this final phase. Overall process:

- Identify parameters to vary in the Kubernetes files of TelluCloud components, and their ranges.
- Model the variability for CAMP and generate amplified configurations.
- Deploy the configurations with Kubernetes.
- Test the deployment by applying messages and tracking the corresponding output.

One TelluCloud service was selected for the experiments, called Actions. The Actions component performs actions such as sending e-mails and generating alarms, based on incoming messages. The input in the tests are alarm messages, and the output is generated alarms.

Tellu focuses on two aspects in these experiments. One is performance optimization - testing different configurations for performance, to see how various attributes impact performance and find optimal configurations. This goes well with Kubernetes configuration, as the most interesting attributes deals with allocation of resources. Optimal configurations will be selected manually, based on the performance relative to the allocated resources.

Three different attributes of the Kubernetes yaml files were selected for amplification:

- `replicas: 1 # CAMP will amplify this value with [1,3,5]`
- `cpu: 200m   # CAMP will amplify this value with [50,200, 350,500]`
- `memory: 500Mi  # CAMP will amplify this value with [200,300,500,600,700,900]`

Replicas is the number of instances of the component.

The other aspect is to amplify unique traces, for measuring K04. For a system of micro-services such as TelluCloud, where system-level operation consists of messages being passed between components for processing, we view a trace as a message path through the system. The variability in this case, where we only have one component type, comes from varying the number of instances (replicas) of that component. The baseline configuration has only one. Messages are processed in the order they are sent to the system, with alarms generated in the same order. With three replicas processing messages in parallel, messages can take three paths and system behaviour may change as a consequence. Five replicas represent a third option. We measure the number of different alarm sequences produced, as well as the average sequence deviation for each configuration.

We generate 1000 alarms for each configuration and compare the execution time for all configurations. These are the measures we use:

Queue Peak: The highest number of recorded messages in the queue simultaneously

Throughput Peak (TP): Throughput is the number of messages consumed per second by the component
Actions Throughput Peak (AT): How long messages stay in the queue on average

Time Difference: The difference in time between the first alarm created and the last alarm created (through comparison of timestamps).

Total Time: The measured time in Grafana from when the queue started growing until it reports all messages consumed. (This is probably a poor measure. It registers in a window of 15, so if there is a message in the queue at the end of the last second of the first window and then one in the first second of the last window 30 seconds will be added. Not very accurate)

CPU Usage peak: As read for the deployment in Kubernetes

Memory Usage peak: As read for the deployment in Kubernetes

After the initial tests it was revealed that the order in which the alarms appeared was not distributed uniformly. As the alarms where indexed it was shown that the first alarm request created on the web, and scheduled in the queues, was not the first to get a resulting alarm from the actions module.
Because of this we decided to re-run tests on some configurations to identify if there were differences on how the pods distribute the workload. We also decided to run tests multiple times for each configuration, to see if there were any performance changes over time. For each test we create 1000 alarms.

The alarms are created in the form: *Alarm Created <Index> <POD>*
For instance, *Alarm Created 965 actions-5f565f747c-52n2t*

For the latter use case we used different measures. These are:
Number Of Alarms Created - Number of alarms created by this pod, of the total 1000 alarms
First Alarm Indexes - The first 5 indexes of the alarms created by this pod, that indicates the order in which the alarms were created by Actions
Last Alarm Indexes - The last 5 indexes of alarms created by this pod, that indicates the order in which the alarms were created by Actions
Total Time spent - Time difference from the last alarm and the first created by this pod in milliseconds
First Five alarms - The first 5 indexes (from web) of the alarms created, that indicates the order in which the alarms were ordered by the web
Last five alarms - The last 5 indexes (from web) of the alarms created, that indicates the order in which the alarms were ordered by the web

Table 31 Example of test results

	POD	Number Of Alarms Created	First Alarm indexes (0 - 999)	Last Alarm indexes	Diff first to last ts millis	First five	Last Five
Test 1	actions-5f565f747c-52n2t	250	9,13,17,19,21	909,910,912,913,915	11062	0,5,8,13,16	981,984,989,992,997
	actions-5f565f747c-2rb2h	250	0,1,3,6,7,12	992,994,996,998,999	8030	4,7,12,15,20	983,988,991,996,999
	actions-5f565f747c-jxc9d	250	2,4,11,16,20	565,569,571,575,576	8935	1,6,9,14,17	982,985,990,993,998
	actions-5f565f747c-w94ds	125	5,10,18,24,32	987,990,993,995,997	8221	3,11,19,27,35	963,971,979,987,995

	actions-5f565f747c-lhr5r	125	8,14,22,28,35	653,659,666,672,677	9434	2,10,18,26,34	962,970,978,986,994

In the example in Table 31, the pod *actions-5f565f747c-52n2t* created the alarms *Alarm Created <0,5,8,13,16,..., 981, 984, 989, 992, 997>* …..

In total it created 250 alarms, using 11062 milliseconds to do so.

The name indicates the order in which the alarms were ordered by the web. However, the order in which they appear in the list of created alarms is not linear to the order they were created, as can be seen in the First Alarm Indexes column. The alarm *Alarm Created 0* was the 9th alarm created by actions, *Alarm Created 4* was the first alarm created.

Validation method - Botsing crash replication

This corresponds to the method for what was simply called Botsing in D5.6. As reported in D5.6, we completed the first two of the three sub-tasks described there, setting up the tool and extracting crash stack traces from the logs of running TelluCloud instances. We found that none of the stack traces were suitable for replication (see D5.6 9.C.1, section “Botsing K08 and K01”)). We have also not been able to find any suitable stack traces in Jira issues.

Earlier experiments have shown that only some TelluCloud stack traces can be replicated, and analysing the code showed that we often have external context such as configuration, database content and queue connections which play an important role in the runtime behaviour. We also have some relatively complex code where a number of conditions must be met. We described why one randomly introduced error could not be replicated in the qualitative evaluation of Botsing in D5.6, section 9.C.2. We wished to do an experiment to better gauge how large a percentage of stack traces in the use case code which can be replicated by Botsing. We have therefore done an experiment where we introduce errors in the code on purpose, causing an Exception to be thrown and a stack trace to be logged. The idea is to test the tool itself and its ability to replicate crashes in the TelluCloud use case code, even though it does not directly produce useful tests. This can give some insight into the use case code as well as the Botsing tool, and hopefully be useful input to the tool developers.

We selected the FilterStore service, which we run as a stand-alone application with test input. This project is one of the use case projects and also includes the other two projects, so that we cover the entire use case code base. Specifically, the projects as they were when starting the experiment:

- no.tellu.cloud.filterstore 4.3-SNAPSHOT from 21.08.2019
- Using no.tellu.cloud.core 3.12-SNAPSHOT
- In turn using no.tellu.lib.tellulib 3.0

We use the latest (at the time) version of Botsing, specifically botsing-reproduction-1.0.7.jar, also testing improvements since the previous iteration. We use default settings, which means Botsing will try for 30 minutes before giving up. For the results to be as relevant as possible, we try to spread the introduced errors throughout the code of all three projects and in different types of code. We aim to test 12 stack traces. For each one, we have the following procedure:

1. Introduce one error in the code, compiling the service as a test application and running it. If we get an Exception with stack trace, we can proceed.
2. Save the stack trace in a file.
3. Compile the service test application as a jar with all dependencies. This is done with Maven (mvn clean install -DskipTests=true), making sure to start with TelluLib if it is affected, then Core, then FilterStore.
4. Run botsing-reproduction with the stack trace and compiled jar as input. We select a frame to replicate

by looking at the stack trace, typically starting with frame 2 or 3.

5. Based on the result of the first run, we may select to try more frames. This could be either if we were unsuccessful and think there may be hope for another frame, or if we were able to replicate the frame quickly and think we may also have success with the next frame.
6. Log the results and time taken (full tool output is stored with the results).
7. If replicating code is generated, we put this in a JUnit test and try to execute it, checking that it does replicate the frames of the trace. All successful tests are stored with the results.
8. Revert the change to the code, removing the error. We are then ready to start from the top with another error.

At the end, we can compute the percentage of stack traces which could be replicated by Botsing, as well as the average frame which could be replicated (using 0 for unreplicated traces). This does not impact the main KPI of K08, since we do not create real test cases or consider real errors.

Validation method - RAMP test generation

Our tests of RAMP are based on the tutorial provided in the project[22]. The process involves two main tools, one to generate models of the software under test, and one to generate unit tests based on these models. Main sub-tasks:

- Run the tutorial, to ensure that we get the tools working on the Tellu development environment.
- Run the tools on the three use case projects - TelluLib, Core and FilterStore.

For each application of RAMP, we have the following procedure (all commands run from project root):

1. Copy the bin folder from the tutorial into the project folder, so the tools are easily available.
2. Build the project: `mvn clean install`
3. Copy dependencies: `mvn org.apache.maven.plugins:maven-dependency-plugin:copy-dependencies` (alternatively use jar files in local repository in class path).
4. Get the class path: `mvn dependency:build-classpath`
5. `export` the classpath to a variable we can refer to when running the tools.
6. Run `botsing-model-generation` to generate models for all packages we are interested in. Log the number of models generated.
7. Select classes we want to generate tests for. Selecting one class at a time, we run `evosuite-master`, specifying the class with the -`class` argument.
8. Also test running on a package with the `-prefix` argument.
9. Check generated tests. These are in the results folder, so not currently integrated to be used. As long as we have some generated tests, we can continue with the rest of the steps.
10. Compute baseline coverage with Clover, logging it with the number of unit tests.
11. Compute baseline mutation score with Descartes, along with number of pseudo- and partially-tested methods.
12. Copy all generated tests into the test folder of the project.
13. Some tests may have RAMP dependencies. Add a test dependency to evosuite-master 1.0.6 in the project pom file, and the necessary import statements for the test class.
14. Run all the generated tests for each class. Some tests may fail. Inspect these to see why. Delete them if there is no obvious fix.
15. Log the total number of tests generated for each class and package tried, and the number of working tests.
16. Running the completed test suite including the generated tests, compute the resulting coverage and the change from the baseline.
17. Compute resulting mutation score and issues with Descartes, and the change from baseline.

All command line operations are done through Git Bash on Windows 10. We use the RAMP tools available in

[22] https://github.com/STAMP-project/evosuite-ramp-tutorial

the tutorial at the time: botsing-model-generation-1.0.4-SNAPSHOT and evosuite-master-1.0.7-SNAPSHOT. We compute code coverage with the OpenClover 4.3.1 Eclipse plugin. We compute mutation score through Maven with PIT 1.4.7 and Descartes 1.2.5 mutation engine. The commands are logged in the test reports, so the details and any changes for individual projects can be found there (see 9.B.4, below).

We run an additional experiment on select classes, to compare the generated tests with manually written tests. This requires classes where we have a good existing suite of tests, and where we are able to generate tests. We want a class which provides the high-level functionality within its package, using most of the other classes in this package, so that the tests provide decent coverage for the package. We compare the coverage provided by the manual versus the generated tests, for the package and for the project as a whole.

Validation method - Flaky tests

The method for our flaky test treatment is described in D5.6, and was not repeated in this iteration.

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 described in D5.6, test reports and outputs from experiments are put into repositories set up by the project on GitHub for this purpose. A new repository for Botsing model seeding has been added to the set. Here is an overview of the repositories, with a brief guide to Tellu reporting.

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

Contains reports from PIT/Descartes 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 new experiments are found in Fall2019 in the Tellu main folder, with one sub-folder for each experiment. The README.md contains the full output logged by DSpot, and all files produced are included.

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

Contains generated configurations from CAMP and test results.

Botsing crash replication: https://GitHub.com/STAMP-project/evocrash-usecases-output

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

RAMP: https://github.com/STAMP-project/evosuite-model-seeding-usecases-output

The Tellu folder contains one folder for each project we have performed model seeding experiments on. The README.md contains a report, the outputs from the tools are in the results folder and additional folders contain reports on coverage and mutation score before and after the experiment.

9.C. Validation Results

9.C.1 KPIs Report

Table 32 gives an overview of the overall KPI results. The rest of the report goes through each KPI in detail.

Table 32 Summary of Tellu KPI measurements

KPI	**Measure**			**Difference with objective**
	Baseline	**Treatment**	**Difference**	
K01-Execution Paths	29,1% coverage **27932 uncovered**	42% coverage **21725 uncovered**	44,5% cov. increase **22,2% decrease in uncovered code**	-18% (40%)
K02-Flaky Tests	3 flaky of 6 tests	Fixed 3	100% fixed (50 % of total)	+80% (20%)
K03-Better Test Quality	**24,7%** (mutation score)	**35,9%** (mutation score)	**45,5%**	+25,5% (20%)
K04-More unique traces	**1 message path**	**5 message paths**	**500%**	+460% (40%)
K05-System specific bugs	**10**	**3 new**	**20%**	-10% (30%)
K06-More config / Faster	**1 configuration** **4 hours to deploy**	**48 configurations** **10 minutes to deploy**	**4800%** **2300%**	+2250% (50%)
K08-More crash tests	12 crashes	3 replicated	25% replicated **No test cases**	-45% (70%)
K09-More prod tests	**162 tests**	**1407 generated tests**	**869%**	+859 (10%)

K01 - More Execution Paths

We first look at the results from tool experiments, and finally the global coverage change throughout the project.

Descartes

The main Descartes experiments were reported in D5.6. All relevant issues found by Descartes were fixed. While the main benefit was on the mutation score, there was a small improvement in coverage as a side-effect of improving some tests. Table 33 is a summary, see D5.6 for details.

Table 33 Summary of Descartes validation results

Project	NCLOC	Baseline coverage	Resulting coverage	Improvement
TelluLib	7 638	53,6%	54,4%	1,49%
Core	27 018	26,5%	26,9%	1,51%
FilterStore	4 662	35,8%	35,8%	0%

DSpot

Report for TelluLib experiments. Baseline is 77 tests and a Clover coverage score of 51,9%.

Experiment ModelTest-Selectors results in Table 34 - test of different selectors on TelluLib ModelTest.

Table 34 Experiment ModelTest-Selectors - test of different selectors on TelluLib ModelTest

Selector	Time (seconds)	Amplified tests	More mut. killed	Coverage
PitMutantScoreSelector	131	1	0	52,0%
JacocoCoverageSelector	16	0	-	-
TakeAllSelector	18	4	-	52,0%

Experiment ModelTest-Pit-Amplifiers results in Table 35 - test of different amplifiers on TelluLib ModelTest.

Table 35 Experiment ModelTest-Pit-Amplifiers - test of different amplifiers on TelluLib ModelTest

Amplifier	Time (m:s)	Amplified tests	More mutants killed	Coverage
Default (none)	2:11	1	2	52,0%
MethodAdd	14:54	3	4	52,1%
MethodDuplicationAmplifier	13:39	3	4	52,1%
MethodRemove	7:06	6	22	Tests broken
FastLiteralAmplifier	41:22	1	2	52,0%
TestDataMutator	42:19	1	2	52,0%
MethodGeneratorAmplifier	24:11	6	9	52,4%

MethodAdderOnExistingObjects Amplifier	25:05	10	16	52,7%
ReturnValueAmplifier	7:26	1	2	52,0%
AllLiteralAmplifiers	1h 32m	2	3	52,0%
NullifierAmplifier	30:13	1	2	52,0%

Mutations can produce unforeseen problems, and we had one such case here. After running DSpot with the MethodRemove amplifier, a number of tests would consistently fail, breaking the build and further experiments. It turned out one of the test resource files had been modified. A number of tests involving TelluLib's DataModel class initialize model contents from file with the following code:

```
model.setMainFile(new File("src/test/resources/data.json"));
model.initFromFile();
model.setMainFile(null);
```

The last line is vital, removing the connection to the file, as otherwise any changes to model contents are persisted back to the file. The MethodRemove amplifier removed this line from a test, with the result being that the file was modified and subsequent tests fail. This type of issue illustrates how important it is to understand what the tool is doing in order to diagnose problems.

Experiment ModelTest-Pit-MultiAmplifiers results in Table 36 - test of combinations of amplifiers on TelluLib ModelTest.

Table 36 Experiment ModelTest-Pit-MultiAmplifiers - test of combinations of amplifiers on TelluLib ModelTest

Amplifiers	**Time (m:s)**	**Tests**	**More mutants killed**	**Coverage**
MethodAdd MethodDuplicationAmplifier	15:03	2	3	52,0%
MethodAdd FastLiteralAmplifier	45:53	2	3	52,0%
MethodDuplicationAmplifier MethodGeneratorAmplifier	31:12	9	12	52,5%
MethodAdderOnExistingObjects Amplifier MethodGeneratorAmplifier	29:59	6	9	52,3%

All tested selectors and amplifiers completed without error, and only JacocoCoverageSelector failed to produce any tests. We see that some amplifiers fail to improve on the default of 1 test and 2 more mutants killed. The time taken with one amplifier varies, up to 1,5 hours for AllLiteralAmplifiers (which combines a number of literal amplifiers). The most promising amplifiers here are *MethodDuplicationAmplifier*, *MethodGeneratorAmplifier* and *MethodAdderOnExistingObjectsAmplifier*. Combining two amplifiers, we see that the time tends to be a bit more than the highest of the two run on their own. So, we should definitely run them together, if we are interested in results from more than one.

Table 37 Experiments on other test classes of TelluLib

Experiment & Amplifiers	**Time (m:s)**	**Tests**	**More mutants killed**	**Coverage**
JsonTest MethodDuplicationAmplifier	9:37	2	3	52,1%
JsonTest MethodAdderOnExistingObjectsAmplifier	14:06	2	5	52,1%
TemplateTest MethodGeneratorAmplifier	2:47	166	1	52,4%
TemplateTest MethodAdderOnExistingObjectsAmplifier + MethodDuplicationAmplifier	5:18	201	4	52,4%
XmlTest MethodDuplicationAmplifier	3:22	2	3	52,1%
XmlTest MethodGeneratorAmplifier	4:10	4	6	52,2%
TelluLibAll MethodDuplicationAmplifier	5h 21m	-	-	-

Using a few of the most effective amplifiers from the initial experiments on other test classes, DSpot was able to generate tests and increase coverage in all cases when limited to amplify a single test class. See Table 37. We see that something strange is going on in the TemplateTest experiments, resulting in a huge amount of tests compared to other experiments and the provided coverage and mutants killed. Amplifying the whole test suite in one run failed with what appears to be a memory/garbage collection issue. As this also takes a very long time to run, we prefer to limit DSpot to one test class at a time.

Table 38 Experiments on project Core

Experiment & Amplifiers	**Time (m:s)**	**Tests**	**More mutants killed**	**Coverage**
JsonUtilTests MethodAdderOnExistingObjectsAmplifier	17:04	0	0	-
JsonUtilTests MethodDuplicationAmplifier	16:54	0	0	-
RuleTemplateSubstituterTest MethodAdderOnExistingObjectsAmplifier	20:10	1	9	26,2%

RuleTemplateSubstituterTest MethodGeneratorAmplifier	25:36	1	9	26,2%
DatabaseServiceImplTest MethodAdderOnExistingObjectsAmplifier	3h 5m	3	39	26,5%
DatabaseServiceImplTest MethodDuplicationAmplifier	-	6	109	26,6%
AddressLookupServiceTest MethodAdderOnExistingObjectsAmplifier	19:59	3 (1 working)	4	26,2%

Table 38 shows experiments on project Core. Baseline is 83 tests and a Clover coverage score of 26,2%. Using a few of the most effective amplifiers on four test classes in the Core project, DSpot was able to amplify three of them. The tests in this project are more complex than in TelluLib, and some or not true unit tests, relying on external components and taking some time to execute. We therefore have longer execution times. One experiment is missing time, as the process had to be paused and later resumed, but it is on the same level as the other experiment on the same class (3 hours). AddressLookupServiceTest uses online address lookup services, and only one of the three generated tests worked correctly. It is hard to see an increase of the overall project coverage from a single test class amplification here, as this is a larger project, but we see below we got a real increase in coverage from this limited set of amplifications.

Tests generated in the various experiments were added to the production test suits of the projects. For most test classes we ran DSpot with different arguments, producing different sets of tests; we selected those with the best coverage increase. The resulting improvements to the test suites are summarized in Table 39, showing the final KPI results for DSpot.

Table 39 Final KPI results for DSpot

Project	**Baseline tests**	**Basel. coverage**	**Resulting tests**	**Resulting coverage**	**Improvement**
TelluLib	77	51,9%	306	53,3%	**2,7%**
Core	83	26,2%	95	26,7%	**1,9%**

Comparing the results on coverage automatically amplified by DSpot with that resulting from the Descartes experiments in Table 39, we see that we got a significantly larger coverage improvement with DSpot. We see the difference mainly in TelluLib, as we did more DSpot experiments there, but even the few experiments in Core produced real results. Since the Descartes improvement comes from manual fixing of Descartes issues, a process very much dependent on manual work by the developer, this is a strong result for DSpot. We continue the comparison in section K03.

Table 40 Comparison of Descartes and DSpot coverage improvements

Project	**Descartes + manual fixes** **Coverage improvement**	**DSpot** **Coverage improvement**
TelluLib	1,5%	2,7%
Core	1,5%	1,9%

Table 9.15: Comparison of Descartes and DSpot coverage improvements

RAMP

Table 41 shows the code coverage results of RAMP test generation. See *K09 - More production level tests* for details on the tests generated in these experiments.

Table 41 RAMP coverage results

Project	**NCLOC**	**Baseline coverage**	**Resulting coverage**	**Improvement over baseline**
TelluLib	8 245	51,9%	64,6%	24,5%
Core	27 462	26,2%	34,9%	33,2%
FilterStore	1 777	39,4%	39,4%	0%
Total	37 484	32,5%	41,7%	**28,2%**

The totals are created by adding together the total and covered lines of code from each project, as described under *Overall code coverage and mutation score* in 9.B.3. It does not include any cross-project coverage we may get from Core and TelluLib code being called from FilterStore. The approach worked well on the two main projects, providing a very substantial increase in code coverage.

Test generation did not work on FilterStore, but it is a high-level project without much code of its own, so this is no big surprise or loss. The total improvement in coverage of all the code taken together is still 28,2%, which must be said to be a very good result of a limited experiment. As the test generation was only done for a select few classes, it will be possible to further increase the coverage with these tools.

We compared the coverage of existing manually written tests with that provided by generated tests, for two classes. The classes DataModel and DataTemplate were selected for this, as they had a good baseline set of tests as well as getting a generated set of tests. Both classes are top-level classes in their respective packages, and should be able to execute much of the code in these packages. Results in Table 42.

Table 42 RAMP generated tests compared to manually written tests

Class	Test class	Tests	Coverage of package	Coverage of project
DataModel	ModelTest (manual)	12	47,4%	33,1%
	DataModel_ESTest (RAMP)	148	49,2%	21,9%
DataTemplate	TemplateTest (manual)	5	73,5%	7,8%
	DataTemplate_ESTest (RAMP)	23	56,9%	4,7%

While there are a lot more test methods in the generated suites, they tend to be much simpler than the manually written ones. The test coverage is comparable, although the manually written ones provide a bit more coverage, at least for the project as a whole.

Global coverage

We have tracked unit test coverage in the three use case projects throughout the STAMP project. Initial measurements were discarded as the code was refactored, but has thereafter been very stable, except for FilterStore having parts removed, reducing the project size. We therefore have numbers from December 2017 for TelluLib and from April 2018 for Core and FilterStore.

TelluLib - Table 43: TelluLib is the project which has seen the most work and experiments in STAMP, and the coverage has improved throughout. The first improvement is mainly due to manual writing of tests to provide a good basis for amplification experiments. Thereafter, the improvements come from the STAMP tools Descartes, DSpot and RAMP, with the last having the most significant impact. The total result is nearly a doubling of coverage, from 34 to more than 65 percent. The main KPI value is defined as the decrease in uncovered code, which gives a lower relative value since we started with a low coverage. Reducing the uncovered code by nearly half is still better than the KPI objective of 40%.

Table 43 Table 9.18: Global coverage in TelluLib throughout STAMP

	20.12.2017	24.04.2018	20.09.2018	25.10.2019	Change
NCLOC	7912	8196	7638	8245	
Unit tests	49	75	76	1062	
Coverage	34%	51,9%	54,4%	65,3%	**92,1%**
Non-covered lines	5222	3942	3483	2861	**-45,2%**

Core - Table 44: Core is a larger project which we found harder to work with, as many of the tests rely on external components and are not true unit tests. The experiments are more limited relative to the code size, but we still had good results. The improvements here are all directly due to experiments with STAMP tools. The increase in coverage is more than 30%, and the corresponding decrease in uncovered lines is 10%.

Table 44 Global coverage in Core throughout STAMP

	25.04.2018	03.10.2018	06.11.2019	Change
NCLOC	26905	27006	27450	
Unit tests	80	95	754	
Coverage	26,5%	26,9%	35,2%	**32,8%**
Non-covered lines	19775	19741	17788	**-10,1%**

FilterStore - Table 45: We include FilterStore for completeness. As it is not suited to unit testing, especially after the removal of some of its code before the final phase of validation, no STAMP effect was achieved on this little project. The changes recorded are only related to code changes and not relevant to STAMP.

Table 45 Global coverage in FilterStore throughout STAMP

	25.04.2018	21.09.2018	28.10.2019	Change
NCLOC	4572	4662	1777	
Unit tests	3	4	2	
Coverage	35,8%	35,8%	39,4%	10,1%
Non-covered lines	2935	2993	1077	-63,3%

Combined - Table 46: As discussed in 9.B.3, we wanted to create a total K01 sum across the three use case projects. Together they had a baseline of 39389 NCLOC and a final 37484 NCLOC, so a slight decrease coming from FilterStore. We include FilterStore in the total as it was part of the use case - it has had no STAMP improvements and some code was removed, but it is also very small and therefore has a small effect on the total.

Table 46 Combined coverage results

	Baseline coverage	Final coverage	Baseline uncovered	Final uncovered
TelluLib	34,0%	65,3%	5222	2861
Core	26,5%	35,2%	19775	17788
FilterStore	35,8%	39,4%	2935	1077
Total	29,1%	42,0%	27932	21725

For the total use case, we have a coverage increase from 29 to 42 percent, a relative improvement of **44,8%**. The decrease in uncovered code is **22,2%**. While this does not meet the very ambitious goal of a 40% decrease, it is still a strong result for this use case.

K02 - More flaky tests identified and handled

The flaky test result was described in D5.6. Flaky tests are rare in Tellu's test suites, and were only found in a small system test suite. Final results replicated here for completion:

- 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%

K03 - Better test quality

Descartes

The main Descartes experiments were reported in D5.6. Descartes was applied to the three use-case projects, calculating mutation score and reporting issues with pseudo- and partially-tested methods. The issues were categorized as relevant issues were fixed by manually improving the tests. Table 47 shows a summary of the results, with mutation scores produced by Descartes. See D5.6 for details.

Table 47 Descartes mutation score result summary

Project	**Baseline mut. score**	**Resulting mut. score**	**Improvement (mut. score)**	**Total issues found**	**Issues fixed**
TelluLib	48%	50%	4,2%	34 (4 partial, 30 pseudo)	15
Core	14%	16%	14%	125 (15 partial, 110 pseudo)	19
FilterStore	54%	54%	0%	4 (2 partial, 2 pseudo)	0

DSpot

DSpot experiment details are reported in section K01. Descartes was used to compute the mutation score, for the baseline and for the final result after the generated test improvements were added to the production test suites. Results in Table 48. DSpot produced a mutation score increase of 6,3% in TelluLib. Comparing the issue reports before and after the treatment, we find that DSpot fixed 3 issues of pseudo-testing in TelluLib. The new test coverage added 3 new issues of partially-testing. In Core the mutation score increase was 14%. The new tests have some issues of pseudo and partial testing, but we found 12 pseudo-tested methods in the baseline which has become partially tested and 1 partially-tested method fixed.

Table 48 DSpot mutation score results

Project	**Baseline mut. score**	**Baseline issues**	**Resulting mut. score**	**Resulting issues**	**Improvement (mut. score)**
TelluLib	48%	30 (3 partial, 27 pseudo)	51%	30 (6 partial, 24 pseudo)	6,3%
Core	14%	125 (20 partial,	16%	132 (35 partial, 97	14%

		105 pseudo)		pseudo)	

Table 49 compares the DSpot result with the manual fixing of Descartes issues reported in D5.6. Note that the baseline is not exactly the same for the two tools, as DSpot was used on a newer revision of the projects, but the differences are small. In TelluLib tests we fixed a total of 15 issues, some of them appearing as a result of initial fixes. 3 of the issues were fixed automatically by DSpot. Even so, the total mutation score improvement from DSpot was larger than that achieved in the Descartes process. In Core the mutation score improvement was the same with both tools. The manual fixes addressed more issues, but DSpot also improved a good number of issues (13 in total). Considering the large amount of manual work involved in the Descartes result, this is a good result for DSpot.

Table 49 Comparison of Descartes and DSpot test quality results

Project	**Descartes + manual fixes**		**DSpot**	
	Mutation improvement	**Issues fixed**	**Mutation improvement**	**Issues fixed**
TelluLib	4,2%	15	6,3%	3
Core	14%	19	14%	1 partial was fixed, 12 pseudo to partial

RAMP

Table 50 shows the mutation score improvements resulting from adding the tests generated in the model seeding experiments. See *K09 - More production level tests* for details on the tests generated in these experiments, as well as the code coverage improvements reported for K01.

Table 50 RAMP mutation score results

Project	**Baseline mut. score**	**Baseline issues**	**Resulting mut. score**	**Resulting issues**	**Improvement (mut. score)**
TelluLib	49%	30 (3 partial, 27 pseudo)	58%	52 (13 partial, 39 pseudo)	18,4%
Core	14%	105 (0 partial, 105 pseudo)	26%	Issue generation failed	85,7%
FilterStore	54%	4 (2 partial, 2 pseudo)	54%	-	0%
Weighted average	23,6%		34,4%		**45,7%**

Table 9.25: RAMP mutation score results

We used mutation scores computed by Descartes. For the resulting mutation score, we added the tests generated by RAMP to the baseline (manually written) tests. We see that we get a mutation score increase in line with the code coverage increase on the two substantial projects, indicating that the new tests really do test additional code. The relative improvement is very large for Core, which had a low baseline score, and which also has the majority of the code of the three use case projects. There is some increase in pseudo/partially tested methods. Unfortunately Descartes failed to generate issue reports for the extended test suite of Core, due to some extremely long method signatures being covered by the new tests. This bug had already been reported, and was being worked on. The important point is that we got a significant increase in mutation score.

As we have seen, the test generation did not work on the high-level FilterStore project, but this small project has little code of its own. To compute a total improvement across all three projects, we compute an average of the scores weighted by the number of lines of code in each project (see NCLOC in K01 table). The resulting improvement from 23,6% to 34,5% constitutes a 45,7% relative improvement.

Global mutation score

We have tracked mutation score for the three use case projects throughout the STAMP project, measured on the same dates as the coverage reported under K01. We include coverage here for comparison with mutation score.

TelluLib - Table 51: As discussed for K01, TelluLib has seen the most work and experiments in STAMP. The coverage has nearly doubled, and the mutation score has had a similar development, from 35% to 60%, a relative increase of more than 70%. And as with K01, the improvements are a combination of manual test writing, Descartes, DSpot and RAMP.

Table 51 Global mutation score in TelluLib throughout STAMP

	20.12.2017	24.04.2018	20.09.2018	25.10.2019	Change
NCLOC	7912	8196	7638	8245	
Coverage (K01)	34%	51,9%	54,4%	65,3%	92,1%
Mutation score	35%	46%	50%	60%	**71,4%**
Descartes runtime	46 sec	78 sec	69 sec	110 sec	

Core - Table 52: Core shows the same strong result for mutation score, from 17% to 28%, a relative increase of almost 65%. This all comes from the STAMP tool validation experiments.

Table 52 Global mutation score in Core throughout STAMP

	25.04.2018	03.10.2018	06.11.2019	Change
NCLOC	26905	27006	27450	
Coverage	26,5%	26,9%	35,2%	32,8%
Mutation score	17%	16%	28%	**64,7%**
Descartes runtime	21 min	15:30 min	13:04 min	

FilterStore - Table 53: As with K01, FilterStore has had no STAMP-related changes but is included for completeness. The reduction in code which is not related to STAMP has caused a fall in mutation score.

Table 53 Global mutation score in FilterStore throughout STAMP

	25.04.2018	21.09.2018	28.10.2019	Change
NCLOC	4572	4662	1777	
Coverage	35,8%	35,8%	39,4%	
Mutation score	52%	54%	46%	-11,5%
Descartes runtime	7:30 min	7:21 min	2:40 min	

Combined - Table 54: We calculate a total mutation score across all three projects as an average weighted by the NCLOC of each project. Together they had a baseline of 39389 NCLOC and a final 37484 NCLOC. Again, we include FilterStore in the total as it was part of the use case and too small to make much of an impact.

Table 54 Combined mutation score results

	Baseline score	Final score	Change
TelluLib	35%	60%	71,4%
Core	17%	28%	64,7%
FilterStore	52%	46%	-11,5%
Total	24,7%	35,9%	**45,5%**

Even with the untreated FilterStore included, the weighted average use case result is a mutation score improvement from 24,7% to 35,9%. The relative increase of 45,5% is more than twice the KPI goal of a 20% increase.

K04 - More unique invocation traces

As described in 9.B.3, we take invocation traces in the TelluCloud system of micro-services to mean message paths through the system. We measure the time taken through the system, the sequence of outputs given a specific sequence of inputs and the deviation from the input sequence seen in the resulting output sequence. This is done as part of CAMP experiments for performance optimization. The test system is deployed based on a configuration amplified by CAMP, and 1000 alarm messages are sent to it, resulting in 1000 alarms as output. In the baseline, there is only one instance of the component producing alarms. Alarms are processed sequentially, so the output sequence is always the same as the input sequence.

In the first test, we tested 48 configurations created by CAMP for the component. The configurations altered:

- Number of replicas, i.e., the number of instances of the component
- Memory of the component in megabytes
- CPU power of the component in thousandths of one cpu. For instance, 200 is 20% of one cpu.

Of the 48 configurations tested it seems the memory limit is the factor that has the most significant impact on performance. CPU limit did not seem to have much impact. For some configurations with too low memory limit the components did not run.

Total execution time of creating 1000 alarms is 94 seconds in the baseline configuration. The best configuration provided by CAMP tweaked this down to 18.94 seconds for the best configuration.

Table 55 Test execution times for a selection of generated configurations

Configuration	**Attributes [replicas, memory, CPU]**	**Execution time**	**Improvement in execution time**
Tellu Regular (baseline)	[1,200,500]	93.99699	-
Config 32	[3,**500**,700]	29.96	68.12
Config 14	[3,500,300]	29.42	68.7
Config 18	[3, 500, 500]	28.52	69.65
Config 41	[5, 500, 900]	27.88	70.34
Config 39	[5, 500, 700]	22.28	76.29
Config 11	[5, 500, 300]	18.94	79.85

As we can see in Table 55, the increase in CPU does not notably affect the execution time. Nor does the number of replicas in any significance when exceeding three. 500 mb is the highest memory configuration, and it also gives the best results.

The workload (number of messages) is always scheduled the same for the running pods, based on the number of replicas. This is done by a distribution policy which distributes the messages to queues and the queues to pods. For configurations with 5 replicas, 3 pods get 250 alarms, and two get 125 alarms. For 3 replicas the numbers are 375, 375 and 250. These numbers were consistent by pod, but the execution time is not. For the first alarms, the workload seems to be done quite evenly. One of the tests is referred to in Table 56.

Table 56 Test results for configuration 12

	POD	Number of Alarms Created	First Alarm indexes (0 - 999)	Last Alarm Indexes	Execution time	First five alarms	Last five alarms
Configuration 12 [5,200,	actions-56f5577d55-2zmml	250	2,7,8,9,13	820,824,825,827,833	15024	4,7,12,15,20	988,983,996,991,999

300] Test 3	actions-56f5577d55-dsr74	250	0,1,3,10,12	995,996,997,998,999	21170	0,5,8,13,16	981,984,989,992,997
	actions-56f5577d55-jc8bm	125	5,19,33,39,46	822,829,831,834,838	15012	2,10,18,26,34	962,970,978,986,994
	actions-56f5577d55-k9qjd	250	6,18,22,29,30	948,952,956,960,963	18878	1,6,9,14,17	982,985,990,993,998
	actions-56f5577d55-r5rvn	125	4,11,17,26,36	828,835,840,841,842	15123	3,11,19,27,35	963,971,979,987,995

All tested configurations are tested at least three times. The execution times improve after the initial test, but flats out over time - see Figure 16.

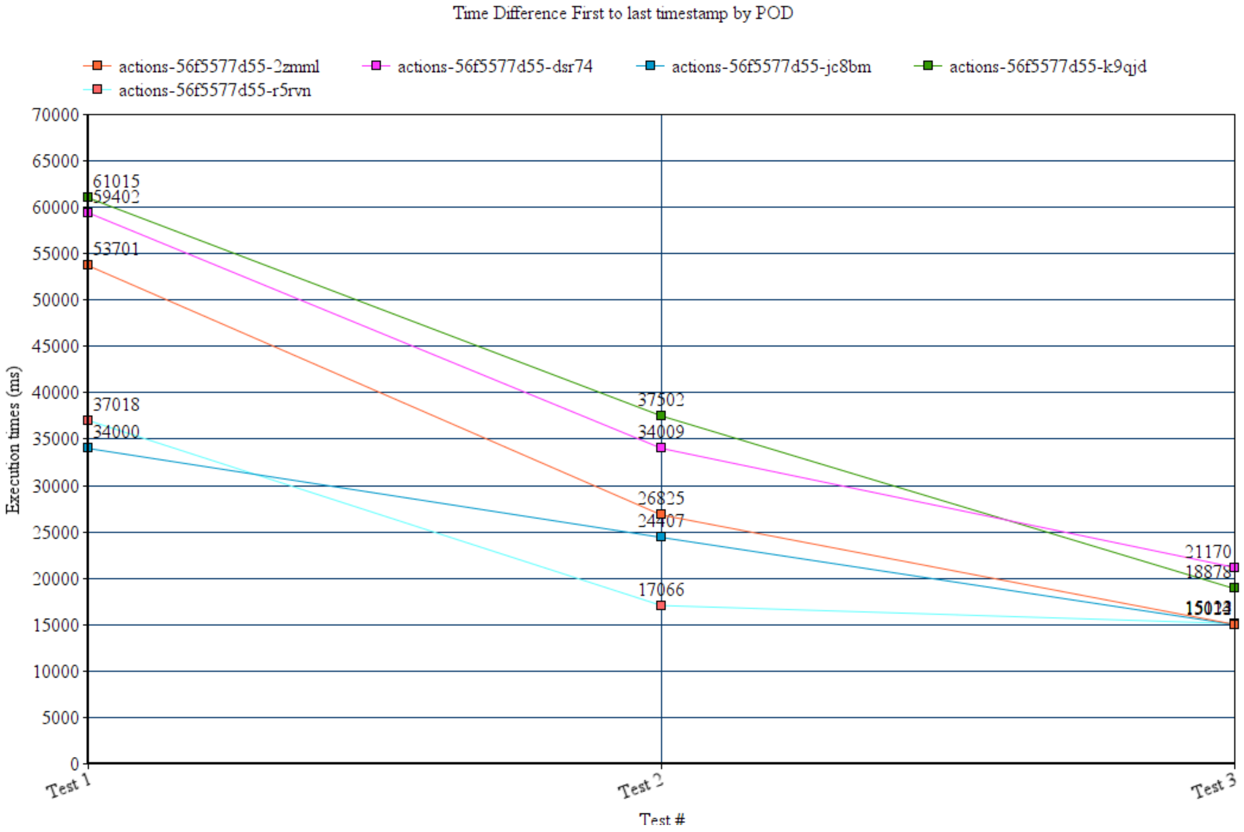
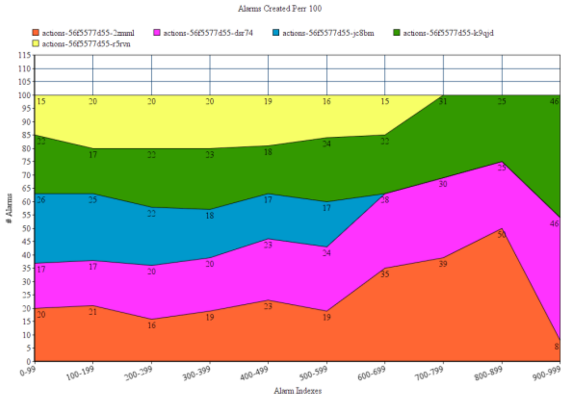
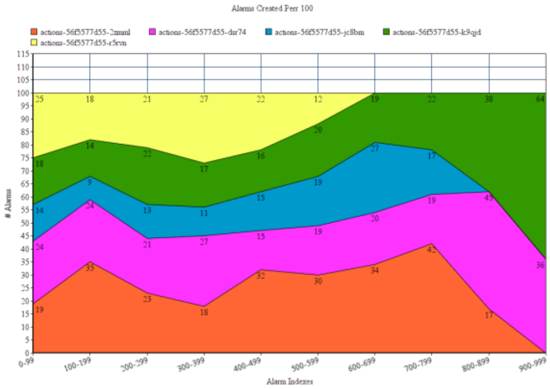


Figure 16 Execution time over three test runs

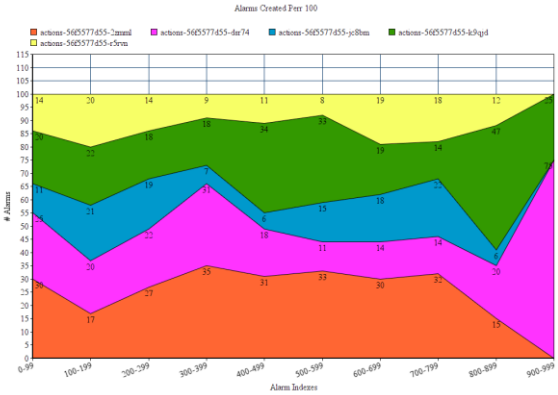
The pods always create the same amount of alarms every test. Sometimes one pod creates more than the 100 final alarms alone. Although they start out with roughly even distribution of the alarms, in the end one pod may end up creating the 100 final alarms or so. The Area Charts in Figure 17 shows number of alarms created by each pod per 100 alarms.



Test 1



Test 2



Test 3

Figure 17 Number of alarms created by each pod per 100 alarms

It turned out that the resulting alarm sequence is different each time for both three and five replicas. There is no clear relation between the configuration and the sequence, other than it having a random element whenever there is more than one replica. The plan to measure the number of different sequences produced by CAMP as an indication of invocation traces turned out to be infeasible - the number of sequences produced by CAMP is as many test runs as we care to do.

The tests calculated the average deviation of each alarm from its sequence. The deviation of one alarm is the absolute value of the difference between the input message's place in the input sequence and the alarm's place in the output sequence. For instance, for message 801 giving the 905th alarm, the difference will be 104. This number was calculated for all alarms and we measured the average. Table 57 shows the deviations in a selection of tests.

Table 57 Average deviation of each alarm from its sequence

Configuration	**Test 1**	**Test 2**	**Test 3**
Tellu Regular (baseline)	0	0	0
Config 32	46.762	33.236	53.256
Config 14	65.29	56.188	94.748
Config 18	69.288	56.658	119.508
Config 41	100.794	84.5	58.238
Config 39	135.986	154.62	106.258
Config 11	96.372	70.93	35.346

As with the sequences themselves, the deviations seem quite random, differing significantly between different runs of the same configuration. The only relationship between configuration and sequence deviation is that certain configurations, like Config 39, always produce more deviation than certain others, like Config 32. The main finding is that message deviation is always significant whenever there is more than one replica.

It is clear that the amplified configurations have produced considerable variation from the single message path of the baseline. Quantifying this for K04 is less clear. What we can say is that the baseline has **one** message path, and the amplified configurations provide up to **five** message paths, as the message can go to one of five different instances of the processing component before the result goes to a single web component to be shown. We could manually produce a configuration with five replicas - the real strength in CAMP is that it has let us explore the configuration space with a large number of different configurations, finding one which provides a large performance increase, with the processing time reduced from **94** to **19** seconds, almost five times (495%) as fast.

K05 - System-specific bugs

This was measured in the docker-compose amplification validation, reported in D5.6. The result was:

- Baseline (existing configuration-related bugs discovered): 10
- New configuration-related bugs discovered: 2
- Improvement (K05): **20%**

K06 - More configurations / Faster tests

Results for docker-compose amplification was reported in D5.6. Baseline was one configuration, and we tested two more thanks to CAMP. The major benefit of using CAMP with Docker is that CAMP takes care of getting the system up and running in a given configuration, greatly speeding up deployment time:

- Baseline time to deploy TelluCloud: 4 hours
- Time to deploy software using CAMP: 10 minutes
- Improvement: **2300%**

In the Kubernetes configuration amplification of the final phase, CAMP produced 48 configurations by varying three attributes (see K04). With a baseline of one, this gives us a KPI for number of configurations tested of **4800%**.

K08 - More crash-replicating tests

As described in the Validation method, we have created stack traces by introducing errors in the code. In this way, 12 stack traces from exceptions throughout the code were created, and we tried to replicate them with Botsing. Table 58 shows the results of each experiment.

Table 58 Botsing replication of introduced errors

Introduced error	Botsing results	Replicated frame
no.tellu.lib.DataValueNode line 317. DataNotFoundException normally thrown by get-method on missing data, now always thrown.	Frame 2: Recreated, 80 seconds. This one is easy, as any call to the second frame will cause a call to the first frame and throw the exception. Frame 3: Not recreated, 30 minutes. Depends on reading a valid configuration string to instantiate a class.	2
no.tellu.lib.model.DataModel line 543. Removed a null-check.	Frame 3: Recreated, 84 seconds. Frame 4: Botsing completed immediately, with code to reproduce the crash. However, the test code contains two errors preventing it from running (see Exception02Test.java, test02_f4). It calls Object.registerNatives(), which is private, and it instantiates the abstract class no.tellu.cloud.mediator.ServiceComponent. Running	3

	Botsing again produced another test with the same errors.	
no.tellu.lib.serialize.JsonSerializer line 400. Throw ParserEscape when creating value node from JSON, breaking our model parsing. This Exception should always be caught, but it will create other errors.	Frame 2: Not recreated, 30 minutes. Depends on reading a valid configuration file.	0
no.tellu.lib.util.ResourceFinder line 119. Dropped check for file.exists, always returning file. Application still runs, but logs the exception.	Frame 2: Not recreated. Logged timeout exceptions. Stopped after no new output for 15 minutes. Frame 3: Not recreated, 25 minutes. Ran slow and produced timeout exceptions. Eventually giving another exception and appeared inactive.	0
no.tellu.lib.util.StateMachine line 86. Removed a null-check.	Frame 3: Not recreated, 30 minutes.	0
no.tellu.findit.cmd.hibernate.CommandHandlerImpl line 29. Return null for DAOFactory. Application runs, but logs a number of Exceptions. We select the first trace.	Frame 2: Not recreated, 30 minutes. Frame 3: Not recreated, 30 minutes.	0
no.tellu.findit.domain.AssetImpl line 328. Remove object instatiation, creating null-pointer. Using root cause exception trace.	Frame 2: Not recreated, 30 minutes. It is targeting a complex method, where a lot of conditions must be met. Frame 3: Not recreated, 30 minutes.	0
no.tellu.findit.services.AddressLookupServiceImpl line 176. Throw NullPointerException. Application runs but logs the exception.	Frame 2: Not recreated, 30 minutes.	0
no.tellu.findit.util.ListToStringUtil line 63. Throw NullPointerException instead of returning serialized strings.	Frame 2: Recreated, 1 second. Test includes EvoSuite class, still works correctly after removing this. Frame 3: Recreated, 1 second. Test includes a statement "PositionImpl.__STATIC_RESET()", this method does not exist. Test works correctly after removing this line. Frame 4: Recreated, 86 seconds.	4
no.tellu.cloud.filterstore.ObservationEncoder line 49. Removed null-check.	Frame 3: Not recreated, 30 minutes.	0

no.tellu.cloud.filterstore.CircularPositionBuffer line 50. Error in Hibernate query string. This is passed to the Hibernate persistence layer, propagating Exceptions up through the layers of our app. We use the root cause exception trace, which involves many frames of Hibernate code. Tellu code starts on frame 17.	Frame 18 (2nd of Tellu code): Not recreated, 30 minutes. The path to the exception is simple, but it probably depends on running in a context where Hibernate is correctly set up. Frame 19 (3rd of Tellu code): Not recreated, 30 minutes.	0
no.tellu.cloud.filterstore.cmd.UpdateSensorPositionCmd line 475. Error in Hibernate query string	Frame 4 (2nd of Tellu code): Not recreated, 30 minutes.	0

Replicated crashes: **3**

Total crashes: **12**

Success rate: **25%**

Average frame replicated: 0,75 (3 among the replications)

We see that when it works, it completes in less than 90 seconds.

We did find a few remaining issues:

- Botsing tries to compile the test it generates, but always fails this step. It is unable to find the JDK, suggesting to set JAVA_HOME and PATH. These were correctly set and other tools work correctly, but it was a similar issue which stopped DSpot from working on Windows 10 with later versions of the JDK. Botsing does in any case print out the test code right before stopping with the error, so we were always able to get the code, but it was without any include statements, and adding the right includes was a bit of manual labour.
- The generated test code sometimes contained statements which could not be executed. We got a private method call, a non-existing method, instantiation of an abstract class and an EvoSuite class. Some of these were not needed for the crash reproduction and could be removed. Perhaps some of these would have been removed if compilation had worked as intended, as at least some of these would have caused a compilation error.

The process of crash replication involves some manual work. We have manually selected which frames to try to replicate, based on our knowledge of the code and previous experience. If every failed attempt takes the default 30 minutes it would also be expensive to automate this by trying a range of frames for each trace, although from our experience this timeout could be reduced. If the code is better suited to crash replication than TelluCloud is, automating the process and consistently producing crash-replicating tests should be feasible.

K09 - More production level tests

Use case project TelluLib

We generated models for two sub-packages, data and util, as this is where most of the code is. This generated a total of **115** model files. **59** of these are for classes in the project, while the rest are from dependencies.

We selected 8 high-level classes, which use many of the other classes in the data package, for test generation. Test generation worked for all, though a small fraction of the generated tests fail. Table 59 shows the results.

Table 59 RAMP test generation results for TelluLib classes

Class	**Generated**	**Working**	**Failed tests**
DataModel	152	148	test030 and test133 fails on IO (creating temp file). test075 fails expecting a nullpointer. test110 runs out of memory because it attempts to allocate a very large structure with a constructor argument.
DataHandleImpl	158	156	Two tests (045 and 048) fail with stack overflow.
DataModelHandle	103	103	
DataTemplate	24	23	test19 fails with stack overflow
MultiLevelHandle	27	27	
MergeHandle	57	53	Four tests (05, 19, 31 and 51) fail with stack overflow.
DataGroupNode	115	114	test005 fails with stack overflow
DataValueNode	134	131	test100 fails trying an invalid operation. 086 and 109 fails its assertion.
Total	**770**	**755**	

Test generation for a whole package (no.tellu.lib.util) with the prefix argument did not work.

Baseline tests (manually written): **77**

Production level tests generated by RAMP: **755**

Generated / baseline: **981%**

Use case project Core

We generated models for the four main sub-packages of no.tellu.findit in this project: cmd, domain, services, and util. This generated a total of **220** model files. **133** of these are for classes in the project, while the rest are from dependencies.

Looking through the modeled packages, we selected classes with a substantial amount of code and no existing test class for test generation, trying to get classes from all packages. Classes in cmd were not suitable as these are just wrapping a few statements in a command pattern. The domain package has a lot of data model classes with mostly simple set/get methods. We could generate a lot more tests here, but limited the selection to three classes as the rest are much the same and probably not benefitting from model generation. A total of 13 classes were selected for test generation. Table 60 shows the results.

Table 60 RAMP test generation results for Core classes

Class	**Generated**	**Working**
no.tellu.findit.services.AsyncServiceImpl	6	4
no.tellu.findit.services.BillingLogServiceImpl	39	26
no.tellu.findit.services.cassandra.ActivityLogItem	48	48
no.tellu.findit.services.cassandra.ActivityLogServiceImpl	24	24
no.tellu.findit.services.cassandra.DeviceHistoryServiceImpl	8	8
no.tellu.findit.services.cassandra.EventHistoryServiceImpl	11	10
no.tellu.findit.services.cassandra.HistoryDatabaseServiceImpl	26	25
no.tellu.findit.services.cassandra.StatisticsCQLServiceImpl	19	16
no.tellu.findit.util.ConvertToJson	34	33
no.tellu.findit.util.TrackingUtilities	36	36
no.tellu.findit.domain.AssetImpl	68	68
no.tellu.findit.domain.CustomerImpl	225	220
no.tellu.findit.domain.ServiceProviderImpl	134	134
Total	**678**	**652**

As before, some tests fail, but here we had more classes where all tests worked. We did not look in detail on the failed tests here, as we covered that for TelluLib, and simply discarded the failing tests when moving tests to the production test suit. As with TelluLib, we also tried to generate tests for a whole package with the prefix argument. We tried this with no.tellu.findit.util and no.tellu.findit.domain. It did not work, only producing errors.

Baseline tests (manually written): **83**

Production level tests generated by RAMP: **652**

Generated / baseline: **786%**

Use case project FilterStore

We generate models for no.tellu.cloud.filterstore, which contains all classes in this project. A total of **95** model files were generated, **14** for classes in this package. We tried generating tests for three classes:

- no.tellu.cloud.filterstore.ObservationEncoder
- no.tellu.cloud.filterstore.FilterStoreService
- no.tellu.cloud.filterstore.devicecommand.HandleDeviceCommandResponseCmd

It did not work, with the same behaviour in all attempts. After starting with Progress 0% there is no output for 10 minutes. Then we get the following error:

```
[MASTER]   10:43:00.093   [main]   ERROR   ExternalProcessGroupHandler   -   Class
no.tellu.cloud.filterstore.ObservationEncoder.  Clients  have  not  finished  yet,
although a timeout occurred.
```

At this point the Java process hangs. We tried doubling the amount of memory for the Java heap, but got the same results. Based on experiments with other tools on FilterStore, we know this project is problematic. It is a high-level project with a very large dependency tree, and little code of its own. The classpath has 214 jar files with a total size of 87,5 megabytes. We are therefore not surprised that test generation fails, and did not pursue the issue further as this project is in any case not a good target for unit tests.

Total KPI result

All working tests generated with model seeding have been copied to the production test suites of the projects. We get the following total results, presented in Table 61.

Table 61 Summary of K09 results

Project	**Baseline tests**	**Generated tests**	**K09 (generated / baseline)**
TelluLib	77	755	981%
Core	83	652	786%
FilterStore	2	0	0
Total	162	1407	**869%**

The total K09 value of 869% far exceeds the 10% goal for this KPI. The KPI have some caveats to be aware of. It is relative to the number of existing tests, so the very high number can partially be explained by the fact that we had relatively few tests in the baseline, making the relative increase large. We must also note that generated tests are small and simple, while at least some of the manually written tests are larger, covering more functionality per method. The coverage increase gives a better indication of the value of the tools, and this is also very good, but much smaller than the increase in test methods (a coverage increase of 28,2% was found in K01).

9.C.2 Qualitative Evaluation and Recommendations

Descartes

The main Qualitative Evaluation of Descartes was reported in D5.6, and is still valid. Descartes was found to be a useful tool for checking test quality and finding issues to fix for improving tests. Tellu decided to include it in the DevOps toolchain to check new tests as they are written. Later versions of Descartes has been used in the final validation phase to compute mutation score and find pseudo-tested methods in the validation of other tools. It has worked without any problems.

DSpot

After the move to Linux, it proved difficult for the new developer to get DSpot running correctly. Experience is needed to tweak parameters and interpret the output from the tool. With the help of the DSpot developer, issues were resolved and the experiments were completed successfully. Once these startup issues were resolved, DSpot worked very well. For almost all test classes DSpot was applied to, it was able to amplify the tests, resulting in both coverage and mutation score improvements.

It is interesting to compare the results of the Descartes and DSpot experiments. We saw that DSpot produced better results, both in terms of coverage increase and mutation score, than the manual fixing of Descartes issues. Manually fixing Descartes issues is a labour-intensive process, so this clearly shows that the automatic amplification of DSpot has merit. It does not fix all the relevant cases of pseudo-testing, so manual fixes will still be beneficial. Based on the experiments, we believe DSpot should be applied first. This will fix some issues but some of the new tests will also do pseudo and partial testing to some degree. Running Descartes, the developer can then check the mutation score and remaining issues, and decide if it is worth doing manual fixes.

CAMP

The two rounds of CAMP validation have explored different aspects of the tool and its usefulness on the TelluCloud use case. In the first, reported in D5.6, CAMP was used on the Docker level, amplifying the Docker configuration and deploying and running the resulting system. This eventually worked well, and proved useful for testing, as CAMP greatly speeds up the process of deploying and testing a Docker image. However, there is little variability to explore in Docker configurations in this use case, so the amplification part of CAMP has limited usefulness here.

In the final phase we focused on amplification of Kubernetes configuration files, which is basically the level above Docker, dealing with the deployment of multiple services and replicas, with resource allocation in a real cloud infrastructure. CAMP has not previously worked on such files, but the CAMP approach proved general and flexible enough that this could be implemented. CAMP generated a relevant set of configurations. This allowed us to explore the configuration space efficiently, giving interesting results.

Botsing

An initial evaluation was given in D5.6. We had two issues which together meant that Botsing crash replication was of low usefulness to the TelluCloud use case. The first is due to the nature of stack traces we see from running TelluCloud instances. Collecting such stack traces, we found that none were suitable for crash replication. There was no outright crashes, as all Exceptions were handled in some way and were part of service operation, dealing with such things as temporary connection issues and wrongly formatted input. The other issue deals with the nature of the TelluCloud code itself, and how reproducible stack traces from this code are. Initial experiments and analysis of the code led us to believe that many traces would be impossible to reproduce in an automated way due to the complexity of the code and the many dependencies to external context, such as the contents of input messages and the database. It is this which we have explored in a more systematic way in the latest experiments, finding that Botsing could reproduce 25% of the stack traces resulting from introducing errors into the code. This number is not directly applicable to real errors, but it gives some idea of the ratio.

The latest experiments showed us that Botsing has matured considerably, now being a complete tool which is documented and relatively straight-forward to use. It also mostly works as intended, replicating crashes as long as the relevant code has a manageable complexity and no implicit external dependencies which is outside the scope of automated mocking.

RAMP

The experiments with RAMP test generation gave us very good results. The main hurdle for applying the tools was finding a good work flow, with the right commands and tools, as the process involves a number of steps and depends to some extent on the nature of the project and the development environment. The provided tutorial is invaluable in this respect.

Generating models for a package works, as does generating tests for a specific class. We did not manage to generate tests for a whole package in one go (running RAMP with the prefix argument instead of class). This means each class to generate for must be selected manually, running RAMP once for each. To improve coverage, it is a good idea to calculate coverage first and look for classes with low coverage. Model generation takes hardly any time. Test generation for one class takes up to a few minutes, depending on the size of the class.

A few things must be done to bring generated tests into the test suite. Some tests have dependencies for mocking, so in a Maven project we need to add a test dependency. As in the tutorial, not all generated tests work, so we must run them to check. There will often be a few failing tests, either throwing an exception or failing an assertion. In most cases we simply delete the failing tests. We get so many working tests, so this is only a small inconvenience.

The generated tests are basic, but they do what they need, exercising many methods with arbitrary inputs. They should provide some protection against regressions, and we saw in the experiments that the mutation score increase was keeping track with the coverage increase, so the coverage appears to be solid. We did get an increase in pseudo- and partially-tested methods, but not bad considering the amount of tests generated. If test coverage is already good, this process will probably not provide much gain, but where coverage is low it is an easy way to get a quick boost. We suspect that some of the generated tests are redundant. For TelluLib, we went from 77 to 832 tests, with an increase in coverage from 51,9% to 64,6%. But these tests take hardly any time to run, and running the complete test suit still takes less than 4 seconds, so we are happy with the result. We have introduced the generated tests to the test suites of our projects, but we take care to keep them distinct from manually written tests.

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?

Large improvements were made in test amplification for the Tellu use case in the final phase. Both DSpot and RAMP are technologies which has provided automatic generation of new test code which reach previously untested code. With some additional manual fixes for STAMP experiments, we saw a total increase in code coverage of 44,5% across the use case projects. Although the decrease in uncovered code did not meet the very ambitious goal of 40%, a very significant amount of new coverage was produced by STAMP technologies, leading us to answer ***Yes*** to VQ1. The amplification tools gave good results in two of the three use case projects - the projects which have unit tests to amplify. Not all code will benefit equally, but the two projects are different in nature and the tools provided improvements for nearly all code we tried them on, so we believe we have a solid basis for a positive conclusion.

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

Yes. Descartes had already proved very useful for this purpose before the final phase of validation. It works well for both efficiently computing mutation scores and in finding pseudo-tested methods. The mutation score gives a good indication of test quality, and the issue report shows exactly which methods are not properly covered. Important issues can be fixed. This will provide small improvements to the mutation score, but most importantly fix critical issues. Tellu sees Descartes as a useful addition to the DevOps toolbox, and will continue to use the tool. The automatic test amplification done by DSpot and RAMP has provided large increases in mutation score for the Tellu use case, more than double the K03 goal of a 20% increase. 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, CAMP helps us to explore the configuration space, generating new configurations based on a combination of attribute ranges. CAMP now works both on the Docker (single machine) and Kubernetes (cloud) levels, with Kubernetes configuration being the main way for Tellu to test in cloud environments.

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

Yes. Automatic test amplification done by DSpot and RAMP provide clear speed benefits in the Tellu use case. These tools generate additional code coverage and mutation score improvements with minimal human effort. Some learning and setup is needed, but using the tools is thereafter easy. DSpot takes some time to run, but this is only done when there are significant new code to generate tests for. It is hard to automate the test amplification fully, running it from the CI, as some manual effort is still needed to check the tests and select which to include in the build. CAMP greatly speeds up the process of deploying and testing a Docker image, so it speeds up component and system tests which can run on Docker deployments. Generating a new set of configurations to explore the configuration space is efficient and easy thanks to CAMP.

9.D Global Takeaway

STAMP has been an interesting learning experience for Tellu. During the project period, the refactoring of the TelluCloud platform into a microservice architecture was completed, and the new TelluCloud was put into production. Tellu has been slowly moving towards DevOps, with the objective of quicker and more automated cycles of development and release. We have learned how important testing is to automation of DevOps. We started the project with a weak test suite. Test coverage was low, and there was no concept of test quality. It has also been a very busy period for the company, and there has been little time to work on improving the state of the tests. Participation in the STAMP project has therefore been a very good match for Tellu. This participation has brought us much insight into testing and test quality. It has given us a chance to experiment and learn. We have carried out many experiments with STAMP tools, especially in the final phase, documented here. The three use case projects selected for unit test experiments are different enough to give different experiences with the tools, and the two main projects have been stable throughout the test phases, allowing us to track the test coverage and mutation score over a longer period.

The STAMP tools have matured throughout the project, until they were able to deliver good results in the final phase. When summarizing the effects of the tools, we see a good increase in test coverage and mutation score. We had a goal for the final validation phase of being able to generate unit tests and see significant improvements in test coverage, and this was achieved. Descartes has proved a good way to measure mutation score, and finding pseudo- and partially-tested methods has provided insight and helped us improve the test quality, raising trust in the tests. Tellu continues to use Descartes, and has included it in our DevOps toolset. The DSpot and RAMP experiments provided a good deal of new test coverage. The two main use case projects are core projects in TelluCloud, and the new tests have been committed and helps test TelluCloud. The results are still fresh so these tools have not yet been included in Tellu's development process, but the next step is to use them on more projects and introduce them to more developers. Another goal was to be able to amplify Kubernetes configuration files, and this was achieved with CAMP. This allowed us to explore the configuration space for performance in cloud deployments. With CAMP we also learned how to deploy the whole system of micro-services in a Docker container with Docker compose, a good approach to running integration tests which does not need cloud infrastructure.

Looking at the KPI results, they vary from far below to far above the objectives. These numbers do not provide much value on their own, they need context and interpretation. Their relative nature means that they depend very much on the baseline state of the use case, and some tools and KPIs are better suited to the TelluCloud use case than others. The whole Tellu report is needed to understand the result. That result looks good to Tellu, especially the automatic amplification. We have learned that there is no free lunch here, as some effort is always needed to set up a tool correctly and interpret the results, but the right effort gives good results. One takeaway for Tellu is that test amplification is indeed possible. Testing is more important than ever for Tellu, and STAMP has helped ensure the quality of the TelluCloud software as we bring new versions into production.

10. XWiki Use Case Validation

The highlights mentioned below only concern the D5.7 period and doesn't include highlights for previous periods, see D5.6 for earlier ones.

XWiki Use Case Highlights

- Overall test coverage has kept increasing steadily even though the XWiki code base analyzed is live and is seeing more than 10 commits per day (over 50K lines of new code added during this last period, and lots of lines removed too).
- Multiple configuration testing has seen an explosive growth of configurations and tests, totalling over 30 configurations tested with hundreds of functional tests. This resulted in less issues raised in the XWiki issue tracker as a consequence of developers fixing configuration-dependent bugs during development.
- At a technical level, the XWiki project was able to measure all KPIs during the period, including K04 and K09 which couldn't be measured in the past period.

10.A. Use Case Description

No change from D5.6.

10.A.1 Target Software

No change from D5.6, except that XWiki is in active development and thus code has been added, modified or removed since D.5.6 was written. All measures and experiments are conducted on the real live source code. This makes it harder to achieve improvements in the various KPIs, but we also believe that it represents a real life experimentation from which the most feedback and learnings can be derived from.

10.A.2 Experimentation Environment

No change from D5.6.

10.A.3 Expected Improvements

No change from D5.6.

10.A.4 Business Relevance

No change from D5.6.

10.B. Validation Experimental Method

10.B.1 Validation Treatments

See D5.6. We're presenting only the changes in the following table, which 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.

Important note related to K04: The way to measure K04 has changed since D5.6 since the defined way from D5.6 has led to 0% additional coverage, after several attempts. The reason is because the XWiki code is generic and is using frameworks that isolate XWiki from its environment/configuration (Hibernate isolates from the database, the Servlet API isolates from the Servlet Container).

Table 62 Controls and Treatments for KPIs for XWiki Software

KPI	Metric	Control and Treatment
K04	More unique invocation traces (microservices) or coverage increase (monolithic)	**Control**: The baseline is the one from the beginning of the STAMP project, i.e. from December 2016, when there was a single configuration tested. Count the number of functional tests. This can be done by listing tests in the XWiki Git repositories: ● For `xwiki-commons`: No functional tests in XWiki Commons by design. ● For `xwiki-rendering`: No functional tests in XWiki Commons by design. ● For `xwiki-platform`: `find . \( -name "*Test.java" -o -name "*IT.java" \) ! -name "AllIT.java" -path "*-test/*-tests/*" -exec grep -H "@Test" {} \; | wc -l` ● For `xwiki-enterprise`: `find . \( -name "*Test.java" -o -name "*IT.java" \) ! -name "AllIT.java" -exec grep -H "@Test" {} \; | wc -l` The number of configurations is 1. The metric is computed with: `number of UI tests that are not Docker tests + number of UI Docker tests * number of configurations`. Note that only Docker-based tests are executed on several configurations. **Treatment**: Compute the increased configuration paths by taking into account the various configurations tests and the new Docker-based tests added and executed on these new configurations. The metric is computed with: `number of UI tests that are not Docker tests + number of UI tests * number of configurations`.

K09	More production-level test cases	**Control**: Count the number of tests at the start of the STAMP project, i.e. from December 2016. This can be done by listing tests in the XWiki Git repositories: ● For `xwiki-commons`: `find . \( -name "*Test.java" -o -name "*IT.java" \) ! -name "AllIT.java" -exec grep -H "@Test" {} \; | wc -l.` ● For `xwiki-rendering`: `find . \( -name "*Test.java" -o -name "*IT.java" \) ! -name "AllIT.java" -exec grep -H "@Test" {} \; | wc -l.` ● For `xwiki-platform`: `find . \( -name "*Test.java" -o -name "*IT.java" \) ! -name "AllIT.java" -exec grep -H "@Test" {} \; | wc -l` ● For `xwiki-enterprise`: `find . \( -name "*Test.java" -o -name "*IT.java" \) ! -name "AllIT.java" -exec grep -H "@Test" {} \; | wc -l` **Treatment**: Compute again the number of functional tests at the time of measure (since XWiki keeps adding new tests as we're measuring on a live system) and count the number of crash-reproducing tests generated by Botsing. The metric is: `number of tests / number of tests generated by Botsing`.

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 KPI K04.

10.B.2 Validation Target Objects and Tasks

See D5.6. We are presenting the changes for K09 because it wasn't measured in D5.6 as the tool wasn't ready at that time.

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

Table 63 Validation Target Objects for XWiki Software

KPI	Metric	Target objects
K09	(All metrics)	Full XWiki Standard (xwiki-commons, xwiki-rendering and xwiki-platform GiHub repository sources). In practice, it depends on the ability of RAMP to generate tests for some XWiki modules (for example, one limitation is that RAMP doesn't support JUnit5 and most XWiki modules have tests written in JUnit5. Thus dynamic model seeding won't be achievable on those modules).

10.B.3 Validation Method

See D5.6. Changes for K04 and K09.

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.

Table 64 Validation/Measurements Methods for XWiki

KPI	Metric	Method/Process
K04	(all metrics)	Process: ● Implement configuration testing using CAMP/TestContainers ○ Convert existing Selenium Tests to Docker-based tests so that they can be executed by CAMP/TestContainers with various configurations ○ Write new tests directly as Docker-based tests ○ Configure the CI to run all the Docker-based tests under all the supported XWiki configurations ● Git checkout the sha1 of the repos corresponding to 2016-12-20 for the baseline. ● Do the same on master. ● Count the number of UI tests that are not Docker tests (i.e. configuration tests). See the control section of 10.B.1 for how to perform this. ● Count the number of Docker tests. See the control section of 10.B.1 for how to perform this.

		○ More precisely they are “*.IT” java classes located in “*-test-docker” directories. ● Count the number of configurations by checking the Jenkins pipeline of CI where the tests are executed on all configurations. ○ The CI job page will report how many configurations were executed (they corresponds to the number of executed tests). ● Compare and compute the KPI as defined in D5.3 for K04.
K09	(all metrics)	Process: ● Git checkout the sha1 of the repos corresponding to 2016-12-20 for the baseline. ● Count the number of tests. See the control section of 10.B.1 for how to perform this. ● Execute RAMP on various XWiki modules (where it succeeds). ● Do the same on master. ● Count the number of tests. See the control section of 10.B.1 for how to perform this. ● Count the number of tests generated by RAMP. ○ RAMP reports the number of tests generated so this figure can be used for the count. In practice what we did was move these tests inside the Maven test directory for the related modules and executed them. We discarded all tests that were either not compiling or not running fine. ● Compute the percentage increase by dividing the number of tests generated by RAMP by the total number of tests. See D5.3 for K09 for more details

10.B.4 Validation data collection and Measurement method

No change from D5.6.

10.C. Validation Results

See D5.6. The report below focuses on new KPI measures.

10.C.1 KPIs report

Table 65 shows a summary of the KPI measures on the XWiki use case, focusing on the obtained measures for the STAMP KPIs. Results and detailed analysis for each KPI are presented in the following paragraphs.

Table 65 Summary of XWiki KPI measurements

KPI	**Measure**			**Difference with objective**
	Baseline	**Treatment**	**Difference**	
K01-Execution Paths	**65.29%** (Code Coverage)	**71.33%** (Code Coverage)	**+9.25%**	**-7.85%** (40% reduction)
K02-Flaky Tests	**0.010%** (% flaky fixed vs total # tests)	**0.290%**	**+2890.5%**	**+2870.5%** (20%)
K03-Better Test Quality	**61%** (mutation score)	**68%** (mutation score)	**+11.47%**	**-8.53%** (20%)
K04-More unique traces	**194** (unique traces)	**1315** (unique traces)	**+577.83%**	**+537.83%** (40%)
K05-System specific bugs	**56** (config-related bugs)	**72** (config-related bugs)	**+28.57%**	**-1.43%** (30%)
K06-More config / Faster	**1** (#config)	**32** (#config)	**+3100%**	**+3050%** (50%)
K08-More crash tests	**9** (#issues reproduced by Botsing)	**13** (#issues reproduced by Botsing)	**+44.44%**	**- 25.56%** (70%)
K09-More prod tests	**40** (#tests)	**213** (#tests)	**+432%**	**+422%** (10%)

K01 - More Execution Paths

The synthetic results for coverage are shown in table 66.

Table 66 K01 Metric Measures for XWiki

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%
	2019-01-01	69.65%
	2019-02-15	69.75%
	2019-04-01	69.9%
	2019-05-05	70.35%
	2019-06-14	70.46%
	2019-08-02	70.7%
	2019-09-06	70.92%
	2019-09-22	71.02%
	2019-11-21	71.33%
Coverage thanks to Descartes	2018-10-02	37% total increase on modules having fixes done thanks to Descartes

	2018-10-03	40% total increase on modules having fixes done thanks to Descartes
	2019-04-01	50% total increase on modules having fixes done thanks to Descartes
	2019-09-17	84% 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
	2019-10-01	8.82% increase on modules where DSpot generated tests
Coverage thanks to CAMP	2018-10-02	0% increase
	2019-09-17	0% increase
Coverage thanks to Botsing	2018-10-02	N/A

Global Coverage

By looking at the global coverage values from the table above, we can see that the global coverage is progressing regularly and it has increased by 6.04% since the beginning of STAMP, which we consider as being a valuable achievement since XWiki had a relatively high coverage at the start of STAMP, and these measures are done on live code (new code being added or modified every day).

The KPI target is to have 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.

So far, we have reduced the non-covered code by (100 - 65.29) - (100 - 71.33) = 6.03% (out of 13.88%) and thus we're at 43.44% of the target.

As noted previously in D5.6, we believe what's even more important than the target value is that the XWiki project has set up some engineering strategies so that the global coverage cannot go down. Thus, the global coverage will continue to rise slowly but steadily even after STAMP is finished.

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 from 2018-10-03 till 2019-09-24. 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.

Table 67 Coverage and Mutation Score Contributions by Descartes Strategy

Date	Module	Coverage increase (%)	Mutation score increase (%)
2018-11-28	xwiki-commons-properties	1	3
2018-11-19	xwiki-platform-refactoring-api	2	2
2019-01-24	xwiki-commons-xml	1	2
2019-01-25	xwiki-platform-rendering-macro-rss	3	47
2019-02-04	xwiki-platform-eventstream-default	3	7
2019-03-28	xwiki-platform-office-importer	0	1
2019-04-23	xwiki-platform-csrf	9	3
2019-05-27	xwiki-commons-logging-logback	0	11
2019-06-05	xwiki-platform-xar-model	3	2
2019-06-07	xwiki-platform-xar-model	0	6
2019-06-07	xwiki-platform-store-transaction	1	4
2019-06-11	xwiki-platform-rest-server	12	9
2019-06-12	xwiki-platform-vfs-api	2	2
2019-06-12	xwiki-platform-store-filesystem-oldcore	1	4
2019-06-18	xwiki-platform-rest-server	0	1
2019-08-23	xwiki-commons-job (several commits)	6	N/A (Descartes failing)

2019-09-04	xwiki-commons-diff-display	0	13
2019-09-24	xwiki-platform-rendering-wikimacro-store	3	5
2019-09-24	xwiki-platform-search-solr-api	6	3
TOTAL		**53%**	**125%**
TOTAL from beginning		**93%**	**241%**

The data shows that by fixing issues raised by Descartes we were able to increase the coverage (53% increase over the period). We can't easily compute the contribution of Descartes to the global coverage. However, 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 6.03% over the period.

Globally, since the start of STAMP, the usage of Descartes has allowed XWiki to increase the coverage by 93% on the modules where corrections were performed. On average each module where the work was done has seen its coverage increases by 2.38%.

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 from 2018-10-02 till 2019-10-01. Note that all issues we found by running DSpot on XWiki modules were reported in the DSpot issue tracker.

Table 68 Coverage and Mutation Score Contributions by DSpot Strategy

Date	Module	Coverage increase (%)	Mutation score increase (%)	Number of generated tests
2019-02-04	commons-filter-api	0.734	0	3
2019-02-04	commons-filter-api	0.734	0	16
2019-02-04	commons-filter-api	1.223	0	74

2019-10-01	commons-component-api	3	3	1
2019-10-01	commons-component-default	2	0	3
TOTAL		**7.69%**	**3%**	**97**
TOTAL from beginning		**8.82%**	**3%**	**107**

It’s important to note that DSpot didn’t generate complete new tests (but it did add assertions to existing tests and thus generated new tests). Please see D5.6 for the analysis on this aspect. We found that DSpot has improved over this period compared to the previous one and that more tests were modified and more assertions were added. In addition, some mutation score could also be increased but the overall coverage increased more than the mutation score.

K02 - More flaky tests identified and handled

Results since D5.6 are shown in the following table:

Table 69 Flaky Test Measurements

Metric	**Date**	**Value**
Number of flaky tests identified and handled	2019-09-18	47 (source)
Number of flaky tests fixed	2019-09-18	31 (source)
Percentage of flaky tests fixed compared to the total number of tests	2019-09-18	0.2901535006% (10684 tests on 2019-09-18)
Improvement	**2019-09-18**	**+2890.5%**

The KPI target for K02 is an increase of 20% of fixed flaky tests compared to the total number of tests. Thus, we are well ahead of that number with 2890.5% that far. See D5.6 for more explanations.

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 +117% increase of mutation score thanks to Descartes over the period. Globally, since the beginning, we got a +233% increase. But, that's the sum of all the increases for all 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 there's no way to measure that value easily. See D5.6 for more details.

The objective for K03 is a 20% increase of mutation scores. What we see is that Descartes is increasing mutation scores by 1 to 47%, with an average of 6.85% 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 configurations are 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). We have defined 3 types of configurations because executing all the tests on all configurations would take too much time and too many CI agents. Thus, we have 3 Jenkins pipelines which are triggered at different times:

- "latest": Triggered once per day and executes all Docker-based tests on the latest versions of supported servers (DB, Servlet Container, Browser)
- "all": Triggered once per week and executes all Docker-based tests on all supported server versions (DB, Servlet Container, Browser)
- "unsupported": Triggered once per month and executes some Docker-based tests on currently unsupported configurations that we'd like to support in the future

Configurations of type "'latest":

Table 70 Floating Configurations Implemented for "latest"

DB Type	DB Version	JDBC Version	Servlet engine Type	Servlet engine Version	Browser Type & Version	Java Version
MySQL	5.7.x	5.1.45	Tomcat	8.5.x	Chrome latest	8.x

PostgreSQL	11.x	42.2.5	Jetty	9.2.x	Chrome latest	8.x
HSQLDB	Version from XWiki	Version from XWiki	Jetty Standalone	Version from XWiki	Firefox latest	8.x
Oracle	11g Release 2	12.2.0.1	Tomcat	8.5.x	Firefox latest	8.x

Configurations of type “all”:

Table 71 Floating Configurations Implemented for “all”

DB Type	**DB Version**	**JDBC Version**	**Servlet engine Type**	**Servlet engine Version**	**Browser Type & Version**	**Java Version**
MySQL	5.7.x	5.1.45	Tomcat	8.x	Chrome latest	8.x
MySQL	5.5.x	5.1.45	Tomcat	8.x	Firefox latest	8.x
PostgreSQL	11.x	42.2.5	Jetty	9.x	Chrome latest	8.x
PostgreSQL	9.4.x	42.2.5	Jetty	9.x	Firefox latest	8.x
PostgreSQL	9.6.x	42.2.5	Jetty	9.x	Chrome latest	8.x
HSQLDB	Version from XWiki	Version from XWiki	Jetty Standalone	Versionfrom XWiki	Firefox latest	8.x

MySQL	5.7.x with utf8mb4	5.1.45	Tomcat	8.x	Chrome latest	8.x
MySQL	5.7.x	5.1.45	Tomcat	8.x	Firefox latest	11.x

Configurations of type “unsupported”:

Table 72 Floating Configurations Implemented for “unsupported”

DB Type	DB Version	JDBC Version	Servlet engine Type	Servlet engine Version	Browser Type & Version	Java Version
MySQL	8.x	5.1.45	Tomcat	8.x	Chrome latest	8.x
MySQL	5.x	5.1.45	Tomcat	9.x	Chrome latest	11.x

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

In total there are 14 configurations on which XWiki tests are executed by the XWiki CI. The XWiki project has hundreds of functional tests (and about 10K tests in total) and several of them have been converted to Docker-based tests (86 as of 2019-08-22, allowing them to be executed on all the mentioned configurations (see table below).

Table 73 Docker-based tests

Date	Number of Docker-based tests
2016-12-20	0
2019-08-22	86

The goal of K04 is to measure all the paths taken by the tests and due to the different configurations. We approximate this by counting the number of different tests executed on the various configurations. Note that several tests can have common execution paths but they will all have some specific paths that are not in other tests (otherwise they’d be the same test) and it’s enough to have a single different execution path to count.

Table 74 Unique Traces Metrics

Date	Metric	Value
2016-12-20	Number of functional UI test methods (including TestContainers-based tests)	194
2016-12-20	Number of TestContainers-based tests	0
2016-12-20	Number of configurations	1
2016-12-20	Number of unique traces	= 194 - 0 + 1 * 194 = 194
2016-08-22	Number of functional UI test methods (including TestContainers-based tests)	369
2016-08-22	Number of TestContainers-based tests	86
2016-08-22	Number of configurations	12
2016-08-22	Number of unique traces	= 369 - 86 + 12 * 86 = 1315
2016-08-22	Increase	= (1315 / 194 - 1) * 100 = 577.83%

At the date of 2019-08-22, the XWiki project has increased the number of unique traces by 577.83% compared to the situation at the beginning of the STAMP project. The goal was to increase this metric by 40% and thus, it's been achieved. The main success factors for achieving it are:

- The XWiki project was executing its tests on a single configuration at the start of STAMP so there was a lot of potential improvement (but it wasn't easy since it required writing Docker images, writing a testing framework based on TestContainers, stabilizing it, creating pipelines for Jenkins, and converting existing tests to this new Docker-based framework.
- A lot of new functional tests have been added in XWiki since the start of STAMP.

K05 - System-specific bugs

Results are shown in the following table for the measures after D5.6.

Table 75 System-Specific Bugs Found Thanks to Configuration Testing

Metric	**Measure date**	**Value**
Number of new configuration-related bugs discovered	2019-02-22	8 (source)
Percentage improvement vs baseline	2019-02-22	= 8 * 100 / 56 = 14.28%
Number of new configuration-related bugs discovered	2019-04-01	9 (source)
Percentage improvement vs baseline	2019-04-01	= 9 * 100 / 56 = 16.07%
Number of new configuration-related bugs discovered	2019-09-18	16 (source)
Percentage improvement vs baseline	2019-09-18	= 16 * 100 / 56 = 28.57%

The target for this KPI is to increase by 30% the new configuration-related bugs discovered. As of 2019-09-18, XWiki is at 28.57%. However, it should be noted that in reality the increase is much higher than this. We have put in place automated tests with the execution of these Docker-based tests automatically by our CI. Thus, whenever some code is added by developers that results in a failing test, the build will fail and the code won't be released. The developer then fixes the introduced bug before it goes in production. As a consequence, these bugs never make it to our issue tracker!

K06 - More configurations / Faster tests

Results are shown in the following table (showing differences since D5.6).

Table 76 Number of New Configurations Tested

Metric	**Measure Date**	**Value**

Number of configurations tested	2018-11-13	26
Number of configurations tested	2018-12-16	29 (+1 new test (OfficeImporterTest) * N configs. Added LibreOffice config
Number of configurations tested	2019-04-01	31 (2 new configs: Added test with utf8mb4 + test with Java11)
Number of configurations tested	2019-09-18	32 (1 new config= Oracle)
Increase (baseline was 1)	2019-09-18	= 3200%

The target for the number of new configurations was 50% more configurations but since the XWiki project was testing automatically on only one configuration before STAMP, we now have 32 times more configuration, i.e. 3200% increase.

K08 - More crash-replicating tests

New results for the period are shown in the following table.

Table 77 Percentage of Crash-Replicating Test Cases

Metrics	**Measure Date**	**Value**
Number of fixed production crash issues not reproduced thanks to Botsing and which have test cases, since the beginning of STAMP	2019-10-07	15 (source)
Number of issues reproduced by Botsing (overall)	2019-10-07	13 (source)

Number of issues reproduced by Evocrash since the beginning of STAMP	2019-10-07	5 (source)
% of crash replicating test cases	2019-10-07	(((15+5)/15)–1)*100 = 33.33%

In order to reach the objective of 70% more crash replicating test cases, we should have had 8 replicating tests. We had 5 for the period since the beginning of STAMP but if we include the ones that were successfully reproduced but that happened before the start of STAMP, it raises the reached percentage to 86.6%, which is above the objective. However, we're not considering this number since we want to count only cases after the start of STAMP.

It should be noted that in the previous period we had 60% crash-replicating test cases produced by Botsing. In practice this means that we had more issues reported in the XWiki issue tracker with stack traces (and tests) than Botsing was able to reproduce. Also, in the previous period the numbers were low and not statistically very relevant. The reality is that the XWiki software is complex and it's difficult for Botsing to succeed in generating tests that match the stack traces. We consider it already as a very interesting achievement that Botsing was able to generate tests for 13 issues. It should also be noted that Botsing has improved recently thanks to adding the Model Seeding approach (static and dynamic analysis of the code base) as input to RAMP. This has allowed to produce more reproduction than we had before (4 more that couldn't be reproduced before). It should also be noted that on the XWiki code base, only the static analysis could be performed since XWiki tests are using JUnit5 and RAMP only supports JUnit4 at this stage. Once it starts supporting JUnit5, it could mean that more tests could be generated.

K09 - More production level tests

Note: This metric could only be measured during this period as the tool wasn't ready before.

Table 78 Percentage of generated tests

Metrics	Measure Date	Value
Number of manual unit tests (for module xwiki-commons-xml)	2019-10-08	57
Number of tests generated by RAMP (for module xwiki-commons-xml)	2019-10-08	247
Percentage of tests added (for module xwiki-commons-xml)	2019-10-08	433.33%

Increased coverage thanks to generated tests (for module xwiki-commons-xml)	2019-10-08	6.9%
Increased mutation score thanks to generated tests (for module xwiki-commons-xml)	2019-10-08	11%
Number of manual unit tests (for module xwiki-commons-context)	2019-10-08	23
Number of tests generated by RAMP (for module xwiki-commons-context)	2019-10-08	100
Percentage of tests added (for module xwiki-commons-context)	2019-10-08	434%
Increased coverage thanks to generated tests (for module xwiki-commons-context)	2019-10-08	5%
Increased mutation score thanks to generated tests (for module xwiki-commons-context)	2019-10-08	3% (cannot go higher since it reached 100% mutation score!)

We found that the generated tests did increase coverage and mutation score a lot. Note that in the table above we're reporting the increases for 2 modules: one that had a relatively low coverage and mutation score (60%+) and one that had a high coverage and mutation score (87+). In both cases the coverage and mutation scores were increased significantly.

This KPI objective was to increase the percentage of tests by 10% and our experiments have shown an average increase of over 400%.

The RAMP tool was developed in the last few months of the STAMP project to generate new unit tests after doing both a static and dynamic analysis of the code. We were able to test it and found that it generated a lot of new tests, which was great. However, we found that the tool is not yet mature enough to be used in production for the following reasons:

- Several generated tests do not compile or execute. They need to be removed.
- There's no guarantee that the mutation score is going to stay stable or increase. The only guarantee (by design) is that the test coverage is going to increase. However, in practice, all our experiments have shown an increase in the mutation score.
- Generated tests sometimes don't increase the coverage. We suggest that in these cases, the tests should be discarded since they consume CPU and execution time without tangibly improving the test suite (although it's possible that it's testing the code with different inputs which could be seen as useful in some cases).
- The generated tests prove the current behavior and thus they need to be reviewed by a human before being committed. In the case of XWiki we found that a lot of the tests were setting edge values for method parameter (such as null) and verifying that the code was throwing a NullPointerException.
- The generated tests don't have any intent and they're sometimes hard to read and shouldn't be committed as is.

10.C.2 Qualitative evaluation and Recommendations

Descartes

Descartes is very stable and mature and generates lots of good results when executed on the XWiki code base. When its recommendations are followed, it increased the quality of the tests (and incidentally also the code coverage).

It requires manual intervention to be used and specific tests sessions to be planned. Thus if you don't make an explicit effort to use it, you won't get any benefit from it. This is why on the XWiki project we've decided to adopt a different approach to using Descartes in order to automate its usage and benefit the most from it.

We've included it in our build and CI so that each XWiki source module has a current mutation score threshold defined, and whenever a developer modifies any code in that module, Descartes is automatically executed in the CI to verify that the quality of the tests doesn't go down (i.e. that the mutation score doesn't go down). If it does, we fail the build and this prevents the code from being releasable. Developers are notified and need to fix the code to increase the mutation score at least to the value of the threshold or higher (the new value then becomes the new threshold). This guarantees that the XWiki tests can only improve in quality. We apply the same principle for test coverage using Jacoco.

After close to one year in production, this system is working well and has allowed to increase the quality of XWiki tests by more than 6% on average, even though the XWiki code base is modified tens of times every single day! We consider this being a major achievement, and Descartes is instrumental in it.

DSpot

Dspot builds on the promise of Descartes but adds the ability to automatically generate new tests, thus freeing time from developers. In the same manner as for Descartes, on the XWiki project we wanted to automate this in the CI. We wanted to write some Jenkins pipeline to simply execute DSpot either at certain times of the day/week or whenever there was a commit (but in this case, applying it on the touched code only), to generate new tests and commit/push them automatically in our CI. We did this manually (by configuring our Maven build to support it and not mix manually-written tests with DSpot-generated ones) and it worked fine. However, we didn't put it in production for one simple reason: we were not able to generate enough tests fast enough when using DSpot on the XWiki code base. We believe that the XWiki code base is quite complex and isn't well-suited for the moment for DSpot to be able to generate any significant number of tests. In other words, the ratio results vs time to execute (and thus the cost of CI agents) was too low. We believe DSpot is a great idea and will have a great future, if it can be further developed to generate more tests on codebases such as the

XWiki one.

CAMP/TestContainers

There are two parts in CAMP: automatic generation of configurations and execution of tests on various configurations. On the XWiki use case, we were not interested in the first part since we support only a fixed set of configurations (fixed set of Databases and versions, fixed set of Servlet Containers and versions, fixed set of Browsers and versions, etc). Thus we were not interested in finding out what new configuration would break our tests since we wouldn't support it and thus wouldn't fix it. We consider it's already hard enough to support all the combinations we support (over 30). On the other side, we were extremely interested by the ability to automatically execute our functional tests on various configurations since this is a domain we were quite bad at and that was generating substantial bug reports by users and reducing the overall perceived quality of the XWiki product. Before the start of STAMP we were only able to test automatically on a single configuration! Thanks to CAMP/TestContainers and all the work that was required (creating docker images for XWiki, creating the TestContainer-based testing framework, converting Selenium tests to be Docker-based), we've now put in production in our CI the automatic execution of our functional tests on all the supported configurations. Thanks to this we've been able to reduce a lot the number of configuration-related issues reported by users. There's still work to be done such as creating even more configurations (for more exotic use cases) and continuing the conversion of the thousand of functional tests we have to be Docker-based, but it's already been tremendously useful since we set it up about 6 months to 1 year ago, and we will definitely continue progressing on it even after STAMP is over.

Botsing/RAMP

The promise of Botsing is to facilitate the analysis of production bugs by generating tests cases fromJava stack traces. During STAMP all the XWiki issues containing stack traces have been collected (over 300 issues) to try to generate tests for them. So far 13 issues have had tests generated for them, reproducing the stack trace at different frame levels. This is already a great achievement, validating the concept and that it works on a real life project. The reproduction level can seem low (4%) but it should be noted that XWiki is a complex project. Our idea for using Botsing was to automatically discover new issues when stack traces are included in them (through a JIRA plugin), execute Botsing and attach the generated tests to the issue if successful. The tool to perform this was developed in STAMP as part of work package 4. However, we didn't put it in production at this stage since reproduction with Botsing currently has a too low rate (less than 4%) and the ratio of cost (CPU used to execute Botsing) vs interest is not favorable. When Botsing is able to increase that number substantially then it'll become viable to start using it in production.

Initially, we wanted to apply the strategy we used with DSpot on RAMP, which is to automatically commit the generated tests but we found that this wasn't currently possible due to limitations above. The main use case is to consider these generated tests as very interesting hints on what new tests to write to get a better coverage. They do half of the work for developers, by providing a strategy to improve coverage. Then it's up to the developer to write a human-readable test based on it and commit the result. These tests are thus very good for increasing developer's productivity.

10.C.3 Answer to Validation Questions

This section provides XWiki 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?

This is definitely the case, especially with the following tools:

- Descartes: when increasing mutation score we've found that code coverage is also increased
- DSpot: generated tests often increase code coverage
- CAMP/TestContainers: even though XWiki uses technologies that abstract the environment (Servlet API, Hibernate, etc) there are still some portions of code that depend on the configuration used and being able to execute all the tests on various configurations allow increasing the code coverage, even if y a small measure.

- RAMP: the generated tests increase a lot code coverage

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

This is also the case especially thanks to Descartes which measure the quality of tests through mutation scores. The RAMP tool also generates tests that increase mutation score a lot and thus participate to the increased confidence.

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

For the XWiki use case this was the biggest benefit. Thanks to the TestContainers integration and the development of the XWiki Docker images, we were able to offer developers the ability to execute, on their own machines, the tests on various production environments, thus verifying that their code works in these environments but also speeding up the turnaround time for these verifications. Moreover, the CI integration also increased the confidence a lot since we can now be sure that all our tests are executed on the supported configurations and a release won't be done until all tests pass.

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

There were clear wins, such as the development of Docker images for XWiki and execution of tests on developer's local machines. This was a huge boost in productivity.

Then there are 3 tools which were successful at generating tests but which didn't have the necessary maturity for the XWiki code base to be able to be put in production: DSpot, Botsing and RAMP. All the 3 hold great promises and it's already a major achievement that we were able to get results with them on a complex code base as XWiki, but they still require some improvements before they can be used in earnest in production.

10.D Global Takeaway

The STAMP research project has been a boon for the XWiki open source project. It has allowed the project to gain a lot in terms of quality. There have been lots of areas of improvements but the main domains that have substantially benefited the quality are:

- Increase of the test coverage. At the start of STAMP the XWiki project already had a substantial automated suite of tests covering 65.29% of the whole case base. Not only the project was able to increase it to over 71% (a major achievement for a large code base of a million lines of code such as XWiki) but also improve the quality of these tests themselves thanks to increasing the mutation score for them by using PIT/Descartes.
- Addition of configuration testing. At the start of STAMP the XWiki project was only testing automatically a single configuration (latest HSQLDB + latest Jetty + latest Firefox). From time to time some testers were doing manual tests on different configurations but this was a very intensive process and very random and ad hoc. We were having a substantial number of issues raised in the XWiki issue tracker about configuration-related bugs. Thanks to STAMP the XWiki project has been able to cover all the configurations it supports (about 31 at the moment of writing) and to execute its functional UI tests on all of them (every day, every week and every month based on different criteria). This is leading to a huge improvement in quality and in developer productivity since developers don't need to manually setup multiple environments on their machine to test their new code (that was very time-consuming and difficult when onboarding new developers).

XWiki has spread its own engineering practices to the other STAMP project members and has benefitted from the interactions with the students, researchers and project members to accrue and firm up its own practices.

Last but not least, the STAMP research project has allowed the XWiki project to get time to work on testing in general, on its CI/CD pipeline, on adding new distribution packagings (such as the new Docker-based distribution now) which were prerequisites for STAMP-developed tools, but which have tremendous benefits by themselves even outside of pure testing. Globally, this has raised the bar for the project, placing it even higher in the category of projects with a strong engineering practice with controlled quality. The net result is a better and more stable product with an increased development productivity. Globally STAMP has allowed a European software editor (XWiki SAS) and open source software (XWiki) to match and possibly even surpass non-European (American, etc) software editors in terms of engineering practices.

11. OW2 Use Case Validation

OW2 Use Case Highlights

- STAMP tools were used as AI-enabled tools in semi-automated mode: they generated tests and helped us understand issues and write new tests by hand (generated tests provide accurate hints and new ideas about how to fix issues). This is a very promising approach, as it provides different views on problems, some of them being difficult to guess for a human developer.
- The use case projects saw very significant increases in code coverage (31% increase) and mutation score (36% increase) thanks to STAMP work and tools (Descartes, DSpot, Botsing and RAMP). The number of test cases was increased by 43%.
- We have developed CI tools to integrate STAMP with our issue tracking system (Gitlab), and provided methodology hints to our community concerning low-intrusive STAMP integration in the CI process (Maven build with monitoring of mutation score).

11.A. Use Case Description

11.A.1 Target Software

Previous experiments were focused on a quite simple and monolithic project: SAT4J. They were useful to validate the usability and limits of STAMP tools, prior to applying them to more complex environments.

They have now been extended to more complex projects:

- Authzforce (https://authzforce.ow2.org/), focused on web-services access control, was addressed without any help from the developer community, to check the usability of STAMP tools with no skill at all regarding the project itself.
- Joram (https://joram.ow2.org/), a JMS-compliant messaging system, was the main target for Descartes, DSpot and Botsing. The experiment received pro-bono help from the Joram team, and was focused on technical, development-related improvements.
- Lutece (http://dev.lutece.paris.fr/), a portal engine and modular CMS, was the main target for CAMP. The experiment received pro-bono help from the Lutece team, and was focused on continuous integration and configuration-related improvements.

Note: another OW2 project, XWiki, is present in the STAMP use-case list (see chapter 10 above for details). As the use-case was conducted by the XWiki team, it provides us with very detailed information and fruitful user experience, that can be of great help for other OW2 projects.

11.A.2 Experimentation environment

Authzforce project

Authforce is the simplest of the three other projects involved. Written in Java, it has a modern-style build system based on maven, and JUnit tests.

Joram project

Joram is a historical project: developed in Java, it has a modular maven build system, but few JUnit tests: most tests rely on a specific test framework, JUnit-like, activated by ant scripts. Many tests are in fact integration tests (more or less : if most tests run Joram to exchange messages, few are “heavy” integration tests, for example focused on recovery, transaction, server restart…).

Applying STAMP test amplification tools on Joram required some refactoring and/or porting of existing tests.

Lutece project

Lutece, as a portal/CMS project, integrates loosely coupled components with multiple dependencies: it is the ideal target for configuration testing (the field for CAMP), because there are several configuration options to tune such as databases, application server, platform, and optimization-related configurations.

11.A.3 Expected Improvements

OW2 has two main goals:

- Determine which STAMP tools are mature enough, i.e. ready to be proposed to project leaders in the community and improve OW2's "market readiness level" indicators for projects.
- Determine which tool is more suitable for which kind of project, in order to refine the value proposal.

In the previous period (covered by deliverable D5.6), both tools and process were evaluated against one project, to evaluate feasibility.

In the current period, the test was generalized and applied to 3 additional use-cases (Joram, Lutece, Authzforce).

Project leaders and teams involved (pro-bono help from both Joram and Lutece teams) wish to enhance their test coverage and code quality, in a win-win approach (helping us with STAMP can provide their added value for free or at low cost).

OW2 has also started integrating STAMP tools in a global production chain: DSpot, Descartes and Botsing can be linked to our Gitlab instance and the maven build.

- Descartes, triggered from the maven build, can generate issues in Gitlab, that suggest DSpot command-line scripts for pseudo-tested code.
- Botsing can react upon submission of a Gitlab issue that contains an exception trace (as a Gitlab web hook), and suggest a test to reproduce it as an issue comment.

The figure 18 below illustrates the tool chain:

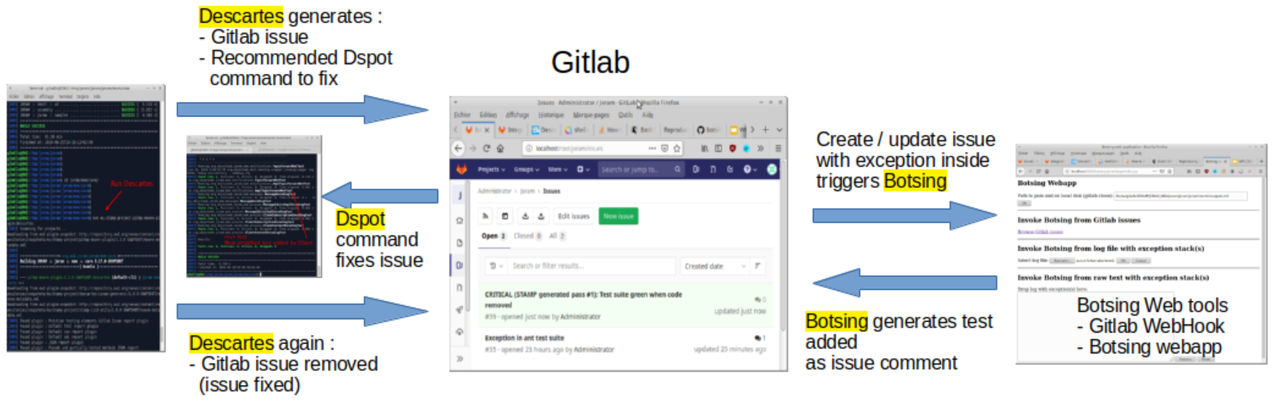


Figure 18 Tool chain

Tools described in figure 18 are available on the STAMP github:

- Descartes issue generator is included in the stamp-ci repository (https://github.com/STAMP-project/stamp-ci/tree/master/descartes-issue-generator).
- Botsing Gitlab web-hook and webapp are available at https://github.com/STAMP-project/botsing-git-webapp .
- Libraries used in these projects are embedded in stamp-ci (https://github.com/STAMP-project/stamp-ci/tree/master/stamp-cicd-utils).

CAMP has also been tested as a tool embedded in the Continuous Integration chain (Jenkins), on Lutece project.

11.A.4 Business Relevance

OW2 provides infrastructure (hosting and quality assessment resources) to open-source projects.

STAMP tools are expected to:

- Enhance OW2's capability to evaluate and classify projects, by adding value to our MRL (Market Readiness Level) methodology.
- Offer OW2 project leaders ready-to-use test amplification tools, that can be activated in the project build and/or continuous integration process.

11.B. Validation Experimental Method

11.B.1 Validation Treatments

Since the last version (D5.6), OW2 has tested STAMP tools on three new projects: Authzforce, Joram and Lutece.

- Authzforce has been used as a testbed and a benchmark for tools usability and use-cases feasibility, without specific help from the development team. Two issues have been posted in the bug tracker, one following a test generated with Botsing.
- Joram and Lutece were our main focus, as we obtained a lot of pro-bono help from the development teams, so we could go far deeper in the use-cases.

KPI measurements presented here, focused on Joram and Lutece, can be summarized as follows (table 79 - this does not mean only a subset of tools has been tested: it highlights where and how we were able to measure KPIs).

Table 79 KPI measurements

	Joram	**Lutece**
KPI measurements	1, 2, 3, 8, 9 (+ 10)	4, 5, 6
Main tools involved	Descartes, DSpot, Botsing, RAMP	CAMP

Note: the focus on tools and KPIs per project was driven by the project's problematic (Joram lacks tests and has many flaky ones, Lutece has complex and multiple configuration schemes) and the profile of the teams that could help us (for Joram, core developers, and for Lutece, integration experts).

11.B.2 Validation Target Objects and Tasks

Table 80 OW2 summary of validation tasks

KPI #	Tools	OW2 projects	Tasks
1-Execution Paths	Descartes DSpot	Joram	Use Descartes to detect where to focus DSpot
2-Flaky tests	Botsing RAMP DSpot Descartes	Joram	Most flaky tests produce exceptions when they fail. - Run botsing to reproduce stack trace / RAMP on classes producing exceptions - Focus DSpot on specific cases - Descartes for measurements
3-Better test quality	DSpot Botsing RAMP Descartes	Joram	Increase mutation score by adding tests - Generate tests when possible (DSpot / Botsing / RAMP) - Add / fix tests when necessary - Descartes for measurements
4-More unique traces	CAMP	Lutece	Compare system traces during executions among configurations and evaluate differences.
5-System specific bugs	CAMP	Lutece	Compare execution behaviour using stress tests to evaluate performance bugs among configurations.
6-More configuration s	CAMP	Lutece	Generate automatically different configurations and evaluate their standard behaviour.
8-More crash tests	Botsing RAMP Descartes	Joram	Integration test suite provides lots of crash scenarii - Apply Botsing / RAMP on them individually - Descartes for measurements
9-More production tests	Descartes Botsing RAMP DSpot	Joram	Descartes for measurements - Add tests, generated (DSpot / Botsing / RAMP) when applicable
10-Three STAMP services in 2 different toolchains	Descartes Botsing DSpot	Joram	Integrate tools in ow2 process - Gitlab + maven build process - Jenkins CI
11-Three use-cases for each STAMP	Descartes Botsing DSpot	Sat4j Joram Lutece	Provide cross-testing table with evaluations of each tool (see 11.C.2)

service	CAMP	Authzforce	
12-Two in-lab workshops with comparative studies	Descartes Botsing RAMP DSpot CAMP	XWiki Sat4j Joram Lutece Authzforce	5 OW2 projects involved, with the collaboration of all partners: - XWiki with full STAMP suite, by development team. - Joram with heavy testing on Descartes, DSpot and Botsing/RAMP, by OW2 with pro-bono help of development team. - Lutece with heavy testing on CAMP, by OW2 with pro-bono help of development team. - Authzforce and SAT4J, by OW2, all STAMP tools evaluation.

11.B.3 Validation Method

We can distinguish two categories of test amplifications provided by STAMP:

- Technical test suite: part of the development process, STAMP is used by the development team to detect and fix bugs, then measure enhancements.
- Configuration tests: part of the integration and validation test process, STAMP is used on alpha or beta releases to validate software compatibility with target deployment environments.

The technical testing uses Descartes for measurements, and a mix of DSpot / Botsing to amplify tests.

The configuration testing uses CAMP to amplify tests.

11.B.4 Validation data collection and Measurement method

Technical test suite amplification

Based on Descartes, DSpot and Botsing, used on an everyday basis by the development team.

The process is iterative, as follows:

1) Issue detection

- Descartes measurements (pseudo-tests, to fix)
- Exception reports (flaky tests, issues reported by users…)

2) Test amplification and/or enhancements

- Focus DSpot and/or Botsing on detected issues
- Focus development efforts on weaknesses pointed out by Descartes (review/fix/integrate STAMP-generated tests when any, and/or develop new tests)

3) Progress detection: back to step #1 (loop forever).

As stated in part 11.A.3 above, this can be integrated with OW2 Gitlab, the Maven build, and the global CI chain.

Configuration tests amplification

Based on CAMP, the process is based on the tool automatically generating new configurations from an input file.

The process is iterative, as follows:

1) Configuration definition

- We set the parameter we want to test in our configuration.
- Lutece application server configuration: in our experimentations, we tested three different versions of Apache Tomcat, version 7, version 8 and version 9.
- Database configuration: in our experimentations, we tested two different versions of MySQL, version 5.6 and version 5.7.
- Java implementation: in our experimentations, we tested two different Java implementations, Hotspot and openJ9.

2) Generation and execution of new configurations

- CAMP generates Docker configurations
- CAMP executes each configuration
- CAMP generates one global report and for each configuration one output folder with detailed logs.

3) Executions analysis

- Analyse global report
- If any difference is detected among configurations, inspect more thoroughly logs and analyse what is the reason of that difference.

All OW2 CAMP experimentations are available on Github:

https://github.com/STAMP-project/camp-samples/tree/master/ow2

11.C. Validation Results

11.C.1 KPIs report

This section shows a summary of experiment results, focusing on the obtained measures for the STAMP KPIs. Results and detailed analysis for each KPI are presented in dedicated paragraphs.

KPI table

Table 81 Summary of KPI metrics

KPI	**Measure**			**Difference with objective**
	Baseline	**Treatment**	**Difference**	
K01-Execution Paths (Joram)	**14.0%** (Code Coverage)	**18.4%** (Code Coverage)	**+31.3%**	**+11.3%** (20%)
K02-Flaky Tests (Joram)	**Not quantifiable** (multiple random exceptions in existing test suite).	**Flaky tests not fixed** (requires skilled human development from Joram team).	**0%**	-20% (20%) But great help for developers (generated tests provide hints).
K03-Better Test Quality (Joram)	**20%** (mutation score)	**27.2%** (mutation score)	**+36.2%**	**+16.2%** (20%)
K04-More unique traces (Lutece)	Initial configuration: config0	88 % difference between config0 and config2	**+88 %**	**+48%** (40%)
	Base for comparing traces.	Highest differences compared to config0.		
K05-System specific bugs (Lutece)	**0**	**0**	**0%**	**-30%** (30%)
K06-More config / Faster (Lutece)	**1** (#config)	**7**(#config)	**+600%**	**+550%** (50%)
	initial configuration	**2 variations on DB** **3 variations on application server** **2 variations on Java implementations**		
K08-More crash tests (Joram)	**0**	**7 exceptions out of 12 successful** (Result: 64 tests in 4 test cases).	**+58%**	-12% (70%)
K09-More prod tests (Joram)	**93 test cases**	**133 test cases**	**+43%**	**+33%** (10%)

K01 - More execution paths

Tests on Joram, using Botsing and DSpot for generation, and Descartes for measurements.

Highlight:

Using only test generation, we obtain a partial success (+13.9% line coverage). When adding tests manually (using Descartes output to know what to add and where), objectives are met (+ 31.3% line coverage).

We probably would gain much more if we had a real knowledge of Joram (the same information in the hands of the Joram team should at least double our performances).

In detail:

DSpot ran test by test on the whole JMS test suite. We developed tools that suggest DSpot command-line focused on missing tests (based on pseudo-testing detected by Descartes), and obtain better statistical results using that (better ratio of solved issues).

Botsing tests generated for flaky exceptions (as described for K02 below) were also included in the experiment.

Manual tests have been added, using Descartes output as hints : about half of the KPI objective being met automatically (DSpot + Botsing = RAMP), the other half manually, with the help of tools (this is not cheating: we had NO SKILL on Joram, as we are only the hosters, not the Joram team).

Global measurements and details available here:

https://gitlab.ow2.org/stamp/joram/wikis/OW2-STAMP-experiments-on-Joram-project-(Descartes,-DSpot,-Botsing,-model-seeding) .

K02: More flaky tests identified

Tests on Joram, using Botsing and RAMP. Considered successful when at least one test is generated to reproduce an exception.

Highlight:

Botsing (and RAMP) provide a methodology to handle flaky tests (and this is GOOD news).

- Flaky tests often produce random exceptions
- It is very difficult, for a developer, to circumvent them (so botsing-generated tests are very helpful).

In detail:

Joram's existing integration test suite (ant-based) produces many random exceptions, that reflect flaky behaviour (like delays on server startups, concurrent accesses to resources…, most of them being synchronization-related issues, either time or thread synchronization).

Botsing successful generation of tests, for about 30% of the exceptions (7 new tests for botsing itself, 56 tests for RAMP).

Our lack of skills concerning Joram (we are the hosters, not the developers !) doesn't allow us to FIX the issues (and even not COUNT issues, some causing multiple exceptions, and some exceptions not being real issues), only to PROVIDE TESTS that reproduce them: thus, we can't determine the percentage of issues fixed.

More details here: https://gitlab.ow2.org/stamp/joram/wikis/Botsing-experiments-(flaky-tests) .

The Joram resulting tests were submitted for review to the Joram team.

K03: Better test quality

Tests on Joram, using Botsing and DSpot for generation, and Descartes for measurements.

Highlight:

Using only test generation, objectives are met (+20.7% mutation score). When adding tests manually (using Descartes output to know what to add and where), we obtain higher results (+ 36.2% mutation score).

We probably would gain much more if we had a real knowledge of Joram (the same information in the hands of the Joram team should at least double our performances).

In detail: see KPI 01 above (same remarks, same tools).

K04: More unique invocation traces

Tests on Lutece, using CAMP and docker-trace-xp

Highlight:

The objective for Lutece project is to evaluate differences in system behaviours among different configurations: MySQL 5.6 and 5.7, Tomcat server 7, 8 and 9, and OpenJ9 and Hotspot Java implementations.

In detail:

Our baseline, config0, represent the recommended environment provided by the Lutece project.

We started the experimentations using perf tool, in order to obtain Flamegraphs outputs. Perf is able to monitor Java calls during execution. Unfortunately, after many workshops organized with Sintef team, we didn't manage to make it work.

We decided to work on strace[23], a lower level tool to monitor system calls. We succeeded in different results.

In this document: https://github.com/STAMP-project/camp-samples/blob/master/ow2/traces/output/strace/configs_test_plots/table_counts.pdf

We can see in detail all the system calls during executions depending on the configuration. For instance, we can see differences between read or write functions.

In this document: https://github.com/STAMP-project/camp-samples/blob/master/ow2/traces/output/strace/configs_test_plots/dissimilarity_graph.pdf

Which is the aggregation of the results of our experimentations, the higher the number and the greater are the differences.

The most differences we find was between two different Tomcat versions, version 8 recommended and version 7. This is the value in the table for K04.

All experimentations data are all available on Github:

https://github.com/STAMP-project/camp-samples/tree/master/ow2/traces

K05: System-specific bugs

[23] https://strace.io/

Tests on Lutece using CAMP.

Highlight:

The objective for Lutece is to evaluate the application response to workload depending on the configuration.

In detail:

At the very beginning of our experimentations with CAMP, we had several workshops to present STAMP and more specifically CAMP to the Lutece team. They presented us the project more in details and what we understood is that they were not interested in unit testing but on performance testing. They have a good knowledge of their environments and they are able to recommend optimal configuration. But they are not able to measure the difference of behaviours when stressing the application on different configurations. In order to gain more confidence, they were interested in that feature. With the help of Sintef team and Engineering, on the CI part, CAMP is now able to present Jmeter outputs.

We did many different experimentations, working on leveraging both configurations and Jmeter test plans charge.

We worked on five different configurations: MySQL 5.6 and 5.7, Tomcat server version 7, 8 and 9.

We used five different test plans, mixing on users input (1.10,100,1000) and test duration time (no time, 60s, 600s).

Those experimentations are all available on Github:

https://github.com/STAMP-project/camp-samples/tree/master/ow2/experimentations

After all those variations, we did not find any failure of the application. The feedback is very positive from the Lutece team.Thanks to CAMP, we can assume the application is able to respond in relatively the same way (response times are very close) under stress on those configurations. The Lutece team can be more confident about their application with that knowledge. This KPI is not contributing to the specific KPI designed at the beginning of the STAMP project, this is why our result is not meeting the initial expectations.

K06: More configurations / faster tests

Tests on Lutece using CAMP.

Highlight:

The goal for Lutece project is to be able to generate automatically relevant configurations

In detail:

As stated above, Lutece team knows the configurations that suit their application. They do not care to know, for instance, that Lutece application is facing issues with Tomcat version 5. They support version 7 and higher, lower versions are out of their perimeter.

The standard test we used for our experimentations was a simple access to the login page of the application.

We based our different configurations on those parameters:

- MySQL 5.6 and 5.7
- Tomcat version 7, 8, 9 and Jetty version 9

- OpenJDK8, OpenJ9 and Hotspot Java implementations

The seven test configurations we worked on are:

- Tomcat 7, MySQL 5.6 and OpenJDK8
- Tomcat 7, MySQL 5.7 and OpenJDK8
- Tomcat 8, MySQL 5.6 and OpenJDK8
- Tomcat 9, MySQL 5.6 and OpenJDK8
- Jetty 9, MySQL 5.6 and OpenJDK8
- Tomcat 7, MySQL 5.6 and OpenJ9
- Tomcat 7, MySQL 5.6 and HotspotJava

All configuration parameters are available on Github:
https://github.com/STAMP-project/camp-samples/tree/master/ow2/template

K08: More crash-replicating tests

Tests on Joram, using Botsing and RAMP.

The original formulation of this KPI implies a way to recognize “crash replicating test cases” in existing test suites. We have none, so the percentage of increase is not computable.

Instead, we can provide “crash replicating test cases” starting from crash reports : this is what we did, using Botsing + RAMP, thus starting from exceptions.

We mainly focused on flaky tests: not only because of KPI 02, but because we know no easy way or method to circumvent them, and Botsing provides at least a part of the answer (and a kind of methodology).

Indeed, many flaky tests randomly generate exceptions, either blocking or not (some are more logs than crashes), that make them visible along the test process: applying Botsing / RAMP to exceptions observed during tests appears as a good way of handling flakiness.

The experiment was conducted on 12 exceptions, with 7 successes and 5 failures (we consider as a success a test generated with Botsing). The exception list is available here:

https://gitlab.ow2.org/stamp/joram/blob/master/botsing-exceptions/flaky-exceptions.txt .

As a result, 64 new tests were generated, gathered in 4 test cases: details on https://gitlab.ow2.org/stamp/joram/wikis/Botsing-experiments-(flaky-tests) .

K09: More production-level tests

Tests on Joram, using Botsing / RAMP and DSpot.

We added 40 new test cases to Joram, representing 43% of the number of pre-existing test cases (93).

Among the added tests cases:

- 22 were generated (18 by DSpot, the others extracted from Botsing-generated classes, ported to plain JUnit, fixed when necessary, and gathered in 4 test cases);
- 13 were developed by hand, with guidance from STAMP tools (mainly Descartes: reports concerning pseudo-tested methods tell where to focus test development effort).

All is detailed on OW2 Gitlab (stamp section), with one commit by bunch of tests, and corresponding documentation and statistics:

https://gitlab.ow2.org/stamp/joram/wikis/OW2-STAMP-experiments-on-Joram-project-(Descartes,-DSpot,-Botsing,-model-seeding).

As stated before, a mix of automatic code generation and tool-guided development was used to reach the objectives. This is the methodology we at OW2 would recommend.

11.C.2 Qualitative Evaluation and Recommendations

OW2 experiments on 4 projects (Sat4j, Authforce, Joram and Lutece), 3 of them detailed herein (the 1st one, Sat4j, was the object of previous experiments as of D5.6), can be summarized as follows in table 82:

Note: the following table reflects global usability of STAMP tools, as perceived from a user point of view, for each OW2 project they have been tested on.

Table 82 Qualitative evaluation of STAMP tools

OW2 project	Descartes	Dspot	Camp	Botsing/RAMP
Sat4J	Good candidate : Descartes spotted relevant issues. Contributed to several commits.	Easy to apply, but amplified test suites bring no added value to existing test suite.	Not applicable (no configuration)	Evocrash: generated tests not relevant.
Lutece	Difficult to apply. Tricky hand-made initializations (that we might even call fixes ?) are necessary, and the pass is very long (like 3 hours). Interesting results.	Not applicable. Too specific build/test environment, requires a database and specific initialization by hand, not independent nor stateless. Lot of effort required, for limited and poor results.	Good candidate : Configurations generated, integration with CI chain.	Little relevance (most tests empty)
Authzforce	Good candidate. Descartes relevant and fast enough (around 10 min).	Easy to apply, but test suites not adequate for DSpot (finds quite nothing to amplify...)	Not tested, but little value expected (few configuration issues)	Minimal test successful (few exceptions available for test)
Joram	Good candidate. provides relevant information to enhance tests. Contributed to several commits and/or tests enhancements.	Good candidate. Shows little initial coverage, but good enhancement rate.	Not tested, but likely to be valuable (multiple configurations make sense)	Good candidate: Helpful for flaky tests (random exceptions in integration tests flow).

Concerning the tools and their applicability:

Descartes

Descartes is a stable tool with industry-grade maturity. It is highly recommendable to project managers, either in the build process (break the build upon mutation score breach), the development process (detect poor testing patterns like pseudo-tests), or even the CI (when fast enough: depends on projects and the way tests are coded). Only poorly structured projects are not Descartes-friendly.

Moreover, Descartes has configurable outputs, either dedicated for human readers (HTML, issues list), machine processing (JSON), or automation (we developed one to generate Gitlab issues, which proves Descartes is also extensible).

Among the STAMP tools, Descartes is our best pick and we recommend it warmly.

Botsing / RAMP

Botsing is a way to understand weird exceptions: it provides accurate crash-reproduction code, that can often be considered original in a developer point of view (something one might not have guessed, like a new idea).

RAMP, when used in addition, provides important increases of mutation and line coverage, at little cost.

The limitation is that Botsing does not always succeed, and Botsing / RAMP require skilled post-processing (generated code requires some effort to be ported to JUnit and cleaned up).

Botsing / RAMP should be focused on specific issues, by developers or maintainers, when necessary: they can be very helpful as companion tools to bring new ideas and help find innovative fixes, but are difficult to use massively and automatically (long processing time and medium success rate).

As a conclusion, Botsing / RAMP should be recommended to skilled people for specific hard debugging (like flaky behaviour). We were surprised by the value added, although the tool is not obvious to use.

CAMP

CAMP can be recommended for projects with complex configuration schemes (eg. multiple external components: servers, databases…), but has little value for monolithic or simple projects.

CAMP remains complex to use, and more suitable for validation / deployment tests on releases than for everyday use by development teams. It clearly has to be used alongside with Docker: standalone use may have unexpected side-effects.

As a conclusion, we could recommend CAMP, but only to complex multi-configuration projects (like Lutece) and when more basic solutions (like plain Dockerfiles) appear insufficient.

DSpot

Although DSpot provided some increase in mutation and line coverage, it seldom succeeded to generate any test, and when so, the result was really difficult to interpret for a human developer.

One can help DSpot succeed more frequently, but it requires to change many parameters and retry: it is time and resource consuming, with no warranty of success.

DSpot punctually can prove efficient to fix something (particularly when mixed with Descartes: we used Descartes output to suggest automatically DSpot commands). It happened for us on a real Joram bug: but the Joram team preferred to rewrite the test by hand, and discard the generated code, considered unlegible.

As a conclusion, we would not recommend DSpot for an industrial usage, except when no other solution can be found.

11.C.3 Answers to Validation Questions

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

Yes, and this is what STAMP is really good at.

On the technical side (development / CI), STAMP tools can be mixed so they help plug holes (thus meeting KPIs 1 and 8):

- Descartes can detect holes and tell developers how they can be plugged using DSpot (where to focus DSpot to make it likely to increase test coverage)
- Botsing can suggest new ideas about possible causes of random behaviour (by generating tests to reproduce such behaviours, a non-trivial task for a developer), and RAMP significantly increases mutation coverage.

The Descartes / DSpot / Botsing / RAMP mix is a kind of “augmented reality” environment for developers: not full automation, but assisted way of amplifying tests. Using STAMP this way is of great help to discover what’s not visible at first sight, and increase coverage dramatically at low cost.

At OW2, this process was initiated on Authzforce project (to conduct experiments and refine it), then implemented on Joram with good improvements of KPIs.

CAMP, combined with tools such as strace or perf, can spot potential weaknesses in a specific configuration. This process requires though manual investigation. Automatic reports can give us the difference in the behaviour, this is how KPI 4 is calculated by the way. A human intervention is needed to analyse the logs and investigate time to understand the reasons for such difference.

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

Yes, definitely:

- STAMP provides ways to measure test coverage, in terms of mutation or more classical coverage.
- STAMP provides ways of amplifying tests suites (by generating tests).

Mixing the two approaches allows to quantify progress due to test amplification.

Moreover, Descartes can be inserted in the maven build and/or the CI process, to break the build when the mutation score is too low: STAMP can enforce high levels of mutation resilience for the code.

This approach, fruitful for the Joram project (KPI improvements), and implemented in production by the XWiki development team, will be promoted by OW2 toward project managers and included in OW2’s MRL (“market-readiness level”) evaluation process.

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

Yes, but dependent on use-cases (more or less relevant, depending on project).

In our experiments, we were able to stress the Lutece application under different test plans using Jmeter with CAMP. To be able to examine different results, under different configurations, and see that everything is going fine, gives more confidence to the project team in their application.

In the eventuality that new exceptions are thrown during the CAMP testing process, Botsing can help provide dedicated tests to cover them (such errors being related to environment changes).

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

Yes, very helpful when properly focused (alongside with human review, not pure automation).

Some STAMP tools can generate tests: Botsing / RAMP and DSpot. Botsing being the most likely to speed up test development, because:

- It is clearly focused on an identified problem (crash replication), allowing to react upon event (eg. Gitlab issue from production environment)
- It generates human-readable code, that can be quickly integrated with existing test suites
- It can provide the developer with new ideas concerning crash causes.

Botsing was used this way in OW2 use-cases:

- On Authzforce, it helped discover lack of validation of values passed to some methods (causing exceptions when wrong or empty values were used)
- On Joram, it provided both new tests, and new insights of flaky tests causes (likely related to thread isolation issues, as suggested by the botsing-generated test code).

11.D Global takeaway

Over those three years, thanks to STAMP, there have been many achievements in the OW2 community regarding testing processes. The knowledge acquired by the management team helped to spread among members new approaches and better understanding of what is at stake regarding testing.

Experience returns from OW2 projects that adopting STAMP in production (done for XWiki, in progress for Joram and Lutece) will refine the way we involve them in OW2's MRL process (Market Readiness Level, the in-house process we use to evaluate project maturity: see https://oscar.ow2.org/view/MRL/). When STAMP tools are used on a project, we consider its code quality and market readiness score higher.

Next step in the short term is providing recommendations and tools to exploit Descartes in production. We already organized workshops for our members and demonstrate STAMP tools. Among them, Orange is strongly interested in Descartes and intends to use it. They expect us to help them and share our knowledge accumulated during our experimentations.

12. Threats to Validity

In D5.6, we reported and discussed potential threats to the validation validity due to the choice of the validation design, namely:

- 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)

These threats to validity are still relevant in this evaluation period, and have been considered during the experimentation, the analysis of results and the drawing of conclusions.

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 might be not homogeneous as the evaluation and validation of STAMP tools across the different use cases may differ from one use case to another. In the following paragraphs, we focus on results that may be shared by all use cases. More details are provided in the “Answers to Validation Questions” section of each use case.

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

There is a positive consensus among use case partners with regard to this question, confirmed by the results obtained in the related KPIs (except for K08, which offers less concluding results). Partners confirm significant improvements in test amplification (and its related code coverage metric), emphasizing that DSpot, Botsing and RAMP are the tools most contributing to this improvement. Descartes is also mentioned as having a positive influence (lower than the others, though) on covering areas of untested code. The ability of CAMP to tests more unique execution paths is also mentioned by some use case partners as an indicator leading to a positive answer to this question.

The UC partners report some limitations, including the high number of similar amplified tests produced by the tools, and the encountered limitations to incorporate some of these tools to the partners’ CI processes.

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, and it is also reflected in the metrics (for most use cases). STAMP tools help to identify weaknesses in test suites. Descartes is particularly useful here. Use case partners find that Descartes helps to efficiently compute mutation scores and significantly reduce the number of pseudo and partially tested methods. By tackling these issues, some partners also report improvements in the mutation score. Use case partners find Descartes’ analysis and report useful to increase the level of confidence on test cases, once the required test/code refactoring that can be derived from the report was applied. Most of the partners confirm their interest in using Descartes in their CI processes, not only for getting reports about their test quality, but also to mark the CI build as failed on the circumstances where the mutation score, reported by Descartes, diminishes.

Partners also report the positive effect of DSpot and Botsing tools in increasing the mutation score, and thus, their confidence in their test cases.

A related impact of the STAMP tools is that they help to generally improve testing culture, behavior and processes, contributing to a higher level of confidence in test cases.

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

There is another positive consensus here. Use case partners report that, since CAMP greatly simplified the generation of alternative configurations, STAMP could help to identify optimal configurations, and to determine the functional compatibility of the SUT under these environments. Concretely, CAMP assists users to classify configurations between those that reliable (e.g. free of bugs) and those are not.

CAMP increases the level of confidence of partners’ teams during the delivery and maintenance operations. Several partners report their interest to adopt CAMP within their CI processes.

A key business advantage of CAMP is that it helps to test different configurations, which contributes to the improvement of software quality as perceived by end-users.

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

There is almost full consensus for a positive answer to this question. The most significant positive result with regard to this question is the huge reduction in the time it takes for developers to test a SUT under a different configuration thanks to CAMP. CAMP facilitates testing the SUT for several aspects (e.g. performance, functional equivalence, etc.) under different environment settings. Before CAMP, this process was manual and time consuming. This improvement is perceived as a huge boost in productivity.

Use case partners also acknowledge the contributions of DSpot and Botsing and RAMP to speed up the test development process thanks to their ability to automatically generate new test cases. However, some partners report some encountered difficulties, including: the low readability for humans observed in some generated tests, the limitations of Botsing to reproduce production runtime tests, the simplicity of some generated tests, the need to check generated tests before being accepted in the CI build, etc. Botsing helps to understand why code fails, but it would be more effective when it automatically generates regression tests that can be integrated in test suites.

13.A Validation Questions Synthesis

This section summarizes answers provided by use cases to the validation questions.

13.B. KPI Synthesis

KPI metrics, consolidated from all use cases, show a significant overall improvement, over this third evaluation period, in the fulfilment of the KPI objectives, for them all. In all KPIs (with the exception of K08), at least one UC has achieved the objective. K06 and K09 objectives were achieved in all UCs. K03 and K04 objectives were achieved in all UCs with the exception of XWIKI (K03) and Atos (K04). K01 and K02 objectives were achieved in three UCs. K05 objective was achieved in two UCs.

13.B.1 K01 - More execution Paths

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

Table 83 K01 Progress Overview

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

Activeeon obtained a 21% increment on code coverage (1% over threshold) for the Catalog component. Atos obtained a 65.68% increment (44.68% over the threshold) combining DSpot, Descartes and RAMP treatments, noticing however that Descartes had a negative effect on coverage (-1.5%) while RAMP had the best one (25%). DSpot effect on coverage was 7.9%. Tellu achieved (in the global TellU system) a coverage relative increment of 44.8%, that corresponds to 22.2% decrement of the uncovered code (17.8% below the threshold). XWiki obtained 8.77% code coverage increment (8.15% below the threshold); this is a very remarkable achievement considering the baseline coverage (65.29%). Both Descartes (2.38%) and DSpot (8.82%)

contributed to this increment. OW2 reported a 31.3% code coverage increase (11.3% over the threshold) for the Joram project.

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

Objective: Difference with baseline > 20 %

Table 84 K02 Progress Overview

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

Activeeon detected 46 flaky tests and fixed 23 (50%, 20% over threshold) since the beginning of the project. 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 (100% of detected flaky test, 80% over the threshold). XWiki reports an outstanding result of +2890.5% for the objective. OW2 identified several flaky tests in the Joram project that could not be fixed because of their limited knowledge on the target project.

13.B.3 K03 - Better test quality

Objective: 20% increase in mutation score

Table 85 K03 Progress Overview

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

Activeeon reported a 23% (3% over threshold) increase in Catalog project. Atos measured a 69.23% increase (49.23% over the threshold) in mutation score, 90% reduction on the number of pseudo tested methods and 90% reduction in the number of partial tested methods. Tellu reported a 45.5% increment in mutation score (25.5% over the threshold), thanks to the combined treatment with Descartes, DSpot and RAMP (45.7% increment). XWiki measured an average of 11.47% of mutation score increase on affected modules (8.53%

below the threshold). OW2 reported 36.2% (16.2% over the threshold) increment in mutation score for Joram project, thanks to the combined treatment of DSpot and RAMP.

13.B.4 K04 - More unique invocation traces

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

Table 86 K04 Progress Overview

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

Activeeon reported an 82% (42% over threshold) increase in the PWS system. Atos reports an increment of 10.86% (29.13% below the threshold) on the number of unique external inter-service invocation paths (service base monolithic system). Tellu reported fewer concluding results, but claiming that they increased the number of unique message paths from 1 to 5 in their tests (400%, 460% over the threshold). XWiki reported an outstanding 577.83% increment despite the monolithic architecture of the application. OW2 reported an increment of 88% (48% over the threshold) for the Lutece project.

13.B.5 K05 - System specific bugs

Objective:30% more such bugs discovered

Table 87 K05 Progress Overview

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

Activeeon reported 5 new detected system specific bugs. Atos detected 5 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, 3 new detections account for a 20% improvement (10% below the threshold). XWiki reported 28.57% increment in the number of system specific bugs (1.43% below the threshold). OW2 did not detect any system configuration bug after testing Lutece system under 5 more configurations.

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

Objective: 50% increase

Table 88 K06 Progress Overview

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

Activeeon obtained a 233% (183% over threshold) increase in the number of configurations. Atos obtained up to 10 new configurations generated compared to the default configuration that was managed before, for their performance scenario; and up to 32 new configurations for their functional scenario (3150% over the threshold). Tellu went from 1 to 48 configurations (4750% over the threshold). XWiki had 1 config before and now has 32 (3050% over the threshold). OW2 increased the number of configurations for Lutece project from 1 to 7 (550% over the threshold).

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

Table 89 K08 Progress Overview

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

Activeeon obtained 1 test replicated over 16 targeted (6.3% success that is 64.7% below the threshold). Atos obtained 3/6: crash replicating tests were generated in 50% of the cases (20% below the threshold) treated by Botsing. Tellu artificially introduced 12 crashes in runtime code, obtaining 3 crash replicating tests (25%

success, 45% below the threshold). XWiki however reported a current increase of 44.44% (25.56% below the threshold). OW2 reported that 7 (of 12) crashes of Joram project were reproduced in tests cases by Botsing (58% increment, 12% below the threshold). Due to different reasons reported by partners (e.g. difficulties to mock inputs, the generation of not compilable tests, etc.), Botsing could not generate tests that reproduce crashes, in a significant percentage of the runtime exceptions it was run against. Nonetheless, it is worth noting that in three of five use cases, Botsing succeeded for more than 45% of the production crashes it was executed against (this is a remarkable achievement). In Tellu use case, many crashes could not be reproduced because the complexity of the code and the many dependencies on the external context. Partners believe that by addressing some reported issues related to the compilability of the generated tests, the inclusion of non-executable code statements, or the mocking of complex input types, the success rate of Botsing will significantly increase, resulting in better K09 metrics.

13.B.10 K09 - More production-level test cases

Objective: 10% increase of existing test suites with production-level test cases

Table 90 K09 Progress Overview

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

Activeeon increased the number of production-level tests by 81% (71% over the threshold). Atos increased 1012% the number of production-level tests (1002% over the threshold). Tellu increased 869% the number of production-level tests (859% over the threshold). XWIKI increased 432% the number of production-level tests (422% over the threshold). OW2 increased 43% the number of production-level tests (33% over the threshold).

13.C. Quality Assessment

This section provides an overview of the tools' quality assessment provided by the use cases. The assessment criteria uses a quality model that was introduced in D5.4 and D5.6.

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. In this last evaluation period, all STAMP tools have been qualitatively evaluated by all industrial partners. The results demonstrate a very good progress during this period in *installability*[24] (>4.5) for DSpot, Descartes and CAMP; *maturity*, specially high for Descartes (~5.0) but with significant progress for the other tools (>3.0); *operability*

[24] The definition of installability and other quality assessment criteria are defined in the quality model introduced in D5.2 and D5.4

(~4.0) with significant progress for DSpot, Descartes and Botsing; *functional completeness* (>4.0)*, correctness* (>3.5) *and appropriateness* (>3.0)*,* improved for all tools; and *learnability* (>3.0) that has been improved for all tools. The *time-behavior* factor has stepped back a bit for CAMP tool, caused by the time cost of camp-execute to run tests suites on the instantiated SUT for amplified configurations.

Overall Descartes and CAMP (in this order) are the STAMP tools with the highest rating by use case partners.

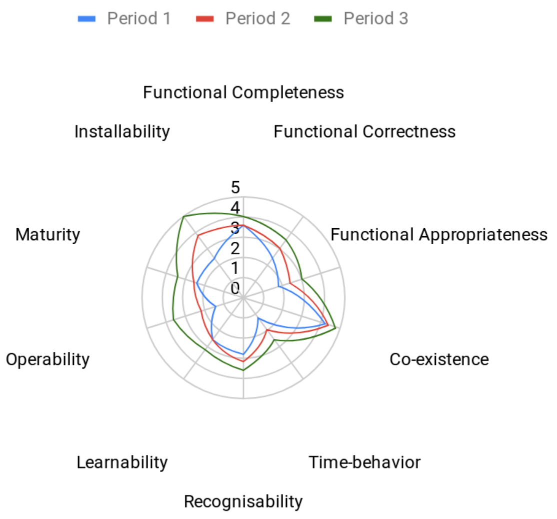


Figure 19 DSpot Quality Assessment Evolution

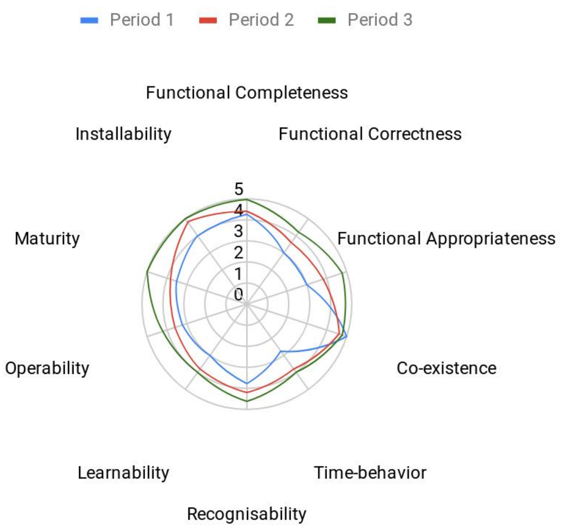


Figure 20 Descartes Quality Assessment Evolution

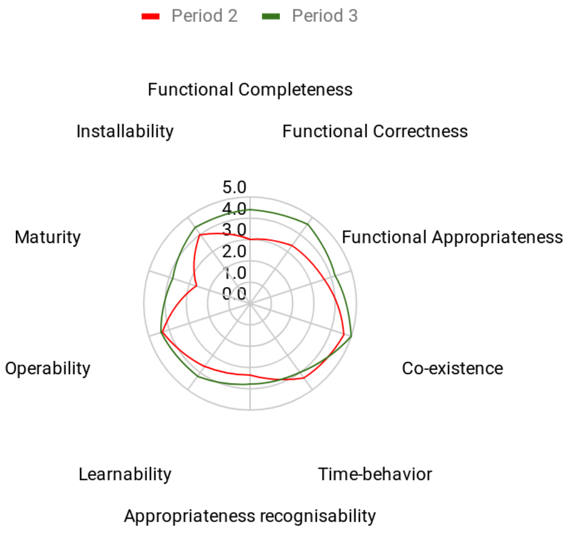


Figure 21 CAMP Quality Assessment Evolution

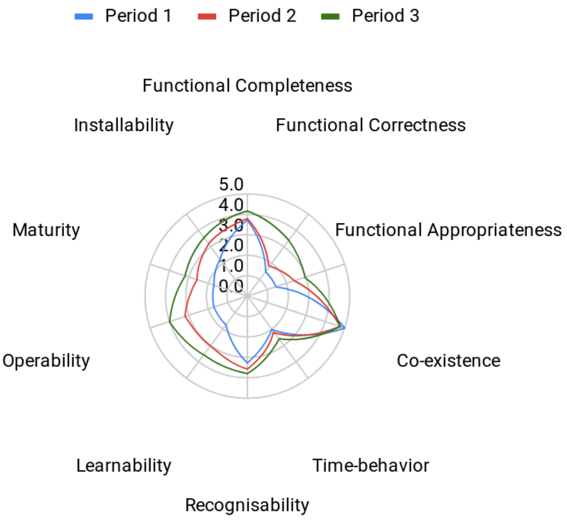


Figure 22 Botsing/RAMP Quality Assessment Evolution

DSpot

DSpot provided useful contributions to improve the test suite for a code base. The tool has great potential since automated test generation has a huge interest.

Significant progress has been accomplished on DSpot since the previous version tested in D5.6. Partners acknowledged DSpot has gained significant maturity in the last development phase, being more robust, offering more client interfaces and better CI integration. Partners also appreciated the large configurability of DSpot. Partners also recognized its improved capability to increase code coverage, while it reduced the number of generated tests (and their degree of duplication)

However, the tool is not yet completely ready for industrialization. Some validation comments focused on industrialization challenges and guidance for improvements. For example, i)DSpot configuration was perceived as complex by several partners, ii) DSpot was incompatible with Windows OS, iii) excessive execution time as to be included in CI processes.

The execution time of DSpot has drastically decreased since the previous release though. Yet, it remains one of the main concerns expressed by some partners. The reported execution time is expressed in hours instead of minutes. Using DSpot is still time consuming, especially when combining several amplifiers.

Figure 19 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 was recognized by all use case partners as a useful and very mature tool. It helped to fix issues related to pseudo and partially tested methods and thus concretely contributed to improve the quality of test suites. The benefits brought by Descartes were real and significant.

Partners agreed that Descartes was an easy-to-use tool (supported by very good documentation), both standalone and integrated in the CI: it produced good reports that were clear to understand. Use case partners appreciated the tool integration in Eclipse, Maven and Gradle.

The most important feature request from use case partners was the possibility to compare two reports in order to identify fixed, remaining and new issues between two runs. At this final stage, the usage Descartes remained implying important manual efforts to analyse its results and improve the tests.

All use case partners acknowledged that Descartes requires deep knowledge of the code that is tested so that its report may offer clear benefits. It should be used by the code and test developers, who understand the specifics of the methods being tested.

Partners have shown their interest to adopt Descartes as part of their tool chain, and CI processes.

The Figure 20 above reflects the average perception of Descartes by the use case partners. It is worthy to note that the overall users' perception has been improved for all the quality factors in this period.

CAMP

CAMP was improved compared to the version evaluated in D5.6. It was also extended by a new add-on called CAMP/TestContainers which can be used to execute various configurations directly from Java test cases. CAMP was easy to install and use. Its function, focused on configuration testing, had a huge potential. Use case partners assessed that CAMP helped them to run their applications under a variety of configurations. Moreover, CAMP helped them find environment-related issues. Partners reported a significant reduction on their number of configuration-related issues thanks to CAMP. It could be very useful to validate that a software can run correctly using different versions of back-end services or help explore new configuration environments.

The CAMP integration with CI processes was largely improved and several reusable generic CI pipelines were implemented by STAMP and adopted by industrial partners.

Despite CAMP having largely simplified the testing of SUT configurations, the analysis of testing results still remained a manual process as well as the exploration of the configuration space to look for optimal/compatible environments.

The performance of CAMP was largely improved, particularly after its integration with CI processes. Its documentation was also much better than for previous releases.

As a downside, the CAMP application to any system requires the specification of variability features in a YAML file that requires specialized CAMP consultancy.

Figure 21 above 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 and RAMP

Botsing is a reimplementation of the EvoCrash tool [D3.3]. Compared to EvoCrash, Botsing is more understandable with improved usability and accessibility. The documentation is much more complete and easier to understand in the current version and it can be run standalone on the CLI or in Eclipse.

Botsing was able to generate test cases that reproduce a number of production crashes described in a log file, although the success rate was low. Nonetheless, this feature was well-appreciated by most of developers (there were a few exceptions where runtime exceptions were not suitable for experimentation with Botsing). Botsing execution was configurable and fast enough (depending on the specified budget).

On the downside, some partners have reported a number of issues, including the generation of empty or non-compilable test cases or the inclusion in tests of non-executable sentences. Other limitations are related to the Botsing inability to mock complex objects.

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.

RAMP with behavioral model seeding was able to generate a huge number of new production-level test cases, although most of partners reported the simplicity of generated tests.

Figure 22 above reflects the average perception of Botsing by the use case partners. The overall users' perception on the tool has improved in this period for all the quality properties.

14. Conclusion

In this third and last evaluation period, STAMP use case partners and development teams have made a real effort to demonstrate the achievement of KPIs' objectives in the evaluation of 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.

Results vary greatly depending on the experimentation environment and the specifics of the target software, but in general, they have demonstrated that all (with the exception of one, K08) KPI's objectives have been achieved. STAMP tools contributed to execute more code paths, an increase that ranges from 8% to 66%. Mutation score was also increased by applying the STAMP techniques in a percentage that ranges from 11% to 69%. The number of runtime unique invocation traces were also increased (10%, 577%). STAMP helped industrial partners to detect a higher number of configuration related bugs, to generate and test more configuration settings, and to speed up this process. Last but not least, STAMP helped partners to reproduce more runtime crashes on their tests suites, with success rates for crash reproduction that ranges from 6% to 58%.

Besides, the experimentation has concluded that the answers to the validation questions were affirmative: i) STAMP tools assist developers to extend their test suites to execute code sentences not tested before, ii) STAMP tools definitively did help developers to improve the quality of their tests so that this increase their confidence on them, iii) STAMP tools did assist developers to test the SUT under different environment settings, and iv) STAMP tools largely speed up the test development process.

The STAMP tools have shown their potential albeit at different degrees given the difference in their development stages and maturity. Use case partners were 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) feedback.

All use cases have manifested their interest in adopting some of the STAMP technologies and tools within their software development processes in alignment with the specifics of their own business processes.

15. Appendix. Comparative STAMP Workshop

As recommended in [D5.4], Atos internally organized a workshop to conduct some experiments aimed to compare the results of applying some STAMP techniques with the usual procedures adopted by the participants. In this workshop, five Java software engineers participated, two of them involved in the STAMP project, and others not related at all.

Before the workshop, participants received detailed instructions about it, available online at:

STAMP Workshop

This documentation provided to the participants consisted of:

1. An introduction to the STAMP project and the techniques/tools that were the experimental subject of the workshop,
2. An introduction to the objectives of the workshop,
3. A list of software requirements needed to conduct the workshop tasks,
4. Instructions to setup the workshop artefacts,
5. A description of the workshop tasks, including instructions to conduct the tasks and a description of the task' objective.

The execution of the workshop proceeded as follows:

1. Participants were welcomed and introduced in the objectives of the workshop,
2. The STAMP project, its tools and technologies were introduced using a presentation whose link is included in the online instructions of the workshop. Only the techniques and tools subject of the workshop experiments were mentioned.
3. Information about the participants' expertise on Java test development technologies was collected.
4. Software requirements were commented. Participants were prompted to install them (if needed).
5. Participants locally cloned the *dhell* project from the GitHub repository. This project was the target of the comparative tasks.
6. The *dhell* project was introduced to the participants, with a focus on the structure of the code base and test suites, and the functionality of this project.
7. For each of the four tasks of the workshop:
   a. The task was introduced in detail. This included: i) the objective of the task, ii) the artefacts provided as input, iii) the tools/techniques to be applied, iv) the expected outputs
   b. Participants executed the task. Each task took 20 minutes
   c. Results were committed locally using Git
8. *Results were pushed to GitHub (in a dedicated dhell project branch for each participant)*
9. Feedback about the workshop was collected from participants.

The following comparative tasks were performed in this workshop:

1. Task 1: Amplify the mutation score of the *dhell* project by manually extending its test suite, addressing the issues reported by Descartes.
2. Task 2: Amplify the mutation score of the *dhell* project by manually extending its test suite, addressing the issues reported by PIT
3. Task 3: Amplify the code coverage of the *dhell* project by manually extending its test suite, using the Clover report as a guidance. Compare the code coverage achieved after this refactoring with the one obtained by applying STAMP DSpot technique.
4. Task 4: Create manually crash replicating test cases for some reported runtime exceptions found in the *dhell* project. Compare these crash replicating test cases with those obtained by applying STAMP Botsing technique with/without behavioral model seeding.

These tasks were designed to give qualitative answers to the following questions:

1. Question 1: Is the report of issues given by Descartes more useful than the PIT's report to help developers to increase the mutation score?
2. Question 2: What is the efficiency of DSpot to increase the coverage compared with the result obtained from developers that refactor the test suites addressing a code coverage report given by Clover (or another similar tool)?
3. Question 3: What is the efficiency of Botsing to generate crash replicating test compared with the developers' one?

Dhell was developed by STAMP as a target project to test DSpot and Descartes technologies for code coverage and mutation score amplification. This simple project was selected as the subject of experimentation for this workshop, instead of another more complex industrial project. The main reason for this decision was that adopting an industrial project would have required either to involve in this workshop participants with deep expertise on that project (e.g. team developers) or to give participants enough time to familiarize themselves with the project's code base and test suites[25]. Besides, *dhell* project is simple enough to offer easy to understand code coverage reports (e.g. Clover); mutation score reports (e.g. PIT) and quality issues report (e.g. Descartes)

Workshop results

The following reports and discusses the results of the workshop, by task.

Participants

All participants declared expertise in Java programming (ranging from 2 to 15 years of experience). Java was the main programming language for 4 of them. They all declared some experience in JUnit4 test development, but none of them claimed to be an expert on test development.

Task1

Table 91 collects the results of the task 1, per participant. It collects the mutation score (reported by PITest/Descartes) and the number of issues reported by Descartes after completing the task 1. The first row provides the reference value (i.e. baseline values collected for *dhell* project without applying the refactoring of any task).

Table 91 Task 1 results for each participant

	Mutation Score	**Remaining issues**
Baseline (Descartes)	*70%*	*5*
Participant-1	83%	1
Participant-2	80%	2
Participant-3	70%	5
Participant-4	73%	4
Participant-5	83%	1

[25] This would have significantly extended the workshop duration.

Two participants achieved a significant in the mutation score (from 70% to 83%) and a reduction on the number of issues (from 5 to 1). There was one participant that could neither improve the mutation score nor reduce the number of issues[26].

Task 2

Table 92 collects the results of the task 2, per participant. It collects the mutation score (reported by PITest) after completing the task 2. The first row provides the reference value.

Table 92 Task 2 results for each participant

Developer	**Mutation Score**
Baseline (PIT)	*37%*
Participant-1	42%
Participant-2	38%
Participant-3	37%
Participant-4	38%
Participant-5	37%

Two participants could not improve the mutation score by applying a test refactoring in reaction to the PIT report. Others achieved a very modest increase (1%), and one participant achieved a better increase (5%).

Comparing task 1 and task 2, we can provide a qualitative[27] answer to question 1. The report provided by Descartes is more useful for developers to introduce a refactoring in the test suites that increase the mutation score.

Task 3

Table 93 collects the results of the task 3, per participant. It collects the code coverage (reported by Clover) after completing the task 3. The first row provides the reference value. The last row provides the value obtained after applying STAMP DSpot.

Table 93 Task 3 results for each participant

Developer	**Code coverage**
Baseline (Clover)	*50.4%*

[26] This participant is the one with lowest Java experience

[27] These results cannot provide a quantitative answer as the number of participants have no statistical relevance. Moreover, this experiment should be conducted against an industrial software project.

Participant 1	55.6%
Participant 2	51%
Participant 3	50.4%
Participant 4	54.4%
Participant 5	53.6%
DSPOT	*64.89%*

Participants obtained increases on the code coverage in the range [0, 5.2%]. These values are significantly lower than the increase obtained by DSpot: 14.59%. Besides, DSpot completed this task in a minute, while participants spent 20 min.

Task 4

Table 94 collects the results of the task 4, per participant. It collects the time it took each participant to create a crash replicating test for each proposed exception[28]. Botsing could not generate any crash replicating test cases for any of these exceptions[29].

[28] Proposed exceptions trigger a *NullPointerException* and a *FileNotFoundException*. See above the link to the workshop documentation for additional information about them.

[29] Botsing was executed using behavioral model seeding (with different probabilities), 100 initial populations and a budget of 360

Table 94 Task 4 results for each participant

Participant	**Experiment time**		
	Exception 1	**Exception 2**	**Exception 3**
Participant 1	1’50’’	1’56’’	5’43’’
Participant 2	7’50’’	8’55’’	-
Participant 3	7’50’’	9’42’’	-
Participant 4	1’05’’	2’41’	16’14’’
Participant 5	1’03’’	3’19’’	10’26’’

Exception 1: all participants could generate a crash replicating test. Three participants completed this objective quite soon since they already pursued this objective when they conducted the task 3 to improve the code coverage, encouraged by the Clover report.

Exception 2: all participants could generate a crash replicating test. It was completed in a short time by the same participants that succeeded fast for the first exception.

Exception 3: only the participants that succeeded for the first and second exception in a short time had enough time to generate a crash replicating test for this exception.

Botsing (with behavioral model seeding) did not succeed to generate a crash replicating test for any proposed exception. Exceptions 1 and 2 look simple and the crash replicating tests created by participants seem to be straightforward. This is not the case of exception 3, which requires a very concrete test input to be generated.