The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Bachelor Thesis

# Supporting Distributed Environments in COMET

presented by

**Kaloyan Todorov**

Aachen, April 26, 2018

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens

# Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Bachelor's thesis entitled

Supporting Distributed Environments in COMET

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

**Official Notification**

**Para. 156 StGB (German Criminal Code): False Statutory Declarations**
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

**Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence**
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

## Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Supporting Distributed Environments in COMET

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, April 26, 2018 (Kaloyan Todorov)

**Belehrung**

**§ 156 StGB: Falsche Versicherung an Eides Statt**
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

**§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, April 26, 2018 (Kaloyan Todorov)

# Acknowledgment

First, I want to thank my family for supporting me during my studies. Without their help, I wouldn't be writing this theses.

Second, I want to thank my supervisor, Andreas Steffens, who not only offered to supervise my work, but also provided me with a very valuable feedback. His knowledge and ideas influenced and helped improving this thesis.

And last, but not least, I want to thank Prof. Dr. rer. nat. Horst Lichter who gave me the opportunity to pursue this thesis at the Research Group Software Construction and who also was my first examiner. In addition to that, I want to thank Prof. Dr. rer. nat. Bernhard Rumpe, who was the second examiner for this thesis.

*Kaloyan Todorov*

# Abstract

Software testing is a crucial part of modern software development. Thanks to the increasing number of users, the demand for more features, better security, redundancy, etc. software testing becomes more and more important and complex. That's why in addition to the regular functional tests [Fun], which test the behaviour of the software itself, non-functional tests [Non] are also needed.

Compliance Testing and Compliance as Code are two very interesting non-functional testing techniques. They focus on testing system configuration and their main goal is to increase the overall system and application security. Unfortunately, not many tools for Compliance Testing exist. That's why creating one was the main priority in [Mos17]. And indeed, such a tool was created. It is called the Compliance Management Tooling (COMET) and with its help one can create, manage and execute compliance tests. However, the tool is not perfect and has some limitations and missing functionality.

That's why in this work we are going to present some of the limitations that COMET has, propose a solution for them and finally implement it. We will be focusing on making COMET portable, i.e. make it possible to execute tests on distributed environments that are not remotely accessible. We are also going to implement the possibility to test complex and distributed systems and to make the testing of multiple Systems Under Test (SUT) at once possible.

# Contents

# List of Figures

# 1 Introduction

Real programmers don't comment their code. It was hard to write, it should be hard to understand.

ANONYMOUS

Contents

The inspiration for this thesis came from [Mos17]. Moscher took the big challenge to not only create a process model blueprint for Continuous Compliance, but to also implement a tool for Compliance Testing, which they ended up calling Compliance Management Tooling (COMET). Although their work was a success and they've accomplished the things they've planned, the *COMET* tool that resulted from their work is far from ideal, because it has some considerable limitations and missing functionality. That's why in this work we want to take *COMET* as a basis, further work on it and at the end, hopefully, improve it.

## 1.1 Contributions

The two main contributions that we want to make to the tool is to make it portable and to enable the possibility to not only test trivial systems (ones that basically consist from 1 computer), but also more complex and distributed ones.

Portability is quite a broad term, so at a first glance it might not be clear what we want to achieve here, but we will try to narrow it down and explain exactly what we mean. Currently *COMET* can only test systems which are accessible through remote connection. We want to remove this limitation. Our blueprint solution for this is to create a smaller tool, COMET-Portable (COMET-P) that can be distributed and deployed independently from *COMET* and that can enable the testing for systems that are not remotely accessible.

Our second contribution will enable the testing of complex and distributed systems, i.e. such that have more than 1 computer. Although that is theoretically possible at the moment, one doesn't have the possibility to define a complex/distributed system in *COMET* and test it as a whole. That's why we want to remove that limitation too. This task also goes hand in hand with the first one, since our portable tool will help us when testing distributed systems that have nodes, which are not meant to be remotely accessible.

Before reading this thesis, we strongly recommend to also read [Mos17], since all of our work will be based on *COMET* and without any background on it, this thesis will be very unclear and hard to read.

## 1.2 Structure of this Thesis

As for the structure of this work, in the next chapter we are going to make a small introduction to *COMET*. After that in chapter 3, we are going to describe some of the problems that *COMET* currently has and also explain why we picked out exactly these 2 problems to work on out of all the available ones. Then, in chapter 4 we are going to present some of the Related Work in the field of Compliance Testing and Distributed Systems.

After we are done with all of that, we are going to lay out the concept for our solution in chapter 5. In it we are going to present our solutions only on conceptional level and keep the technical details for chapter 6.

When we are ready with the implementation of our new tool, *COMET-P*, and also with the improvements to *COMET*, we are going to test everything and evaluate our results in chapter 7.

At the end, we are going to make a short conclusion in which we are going to summarize what we've achieved in our work and also propose topics for possible future work.

# 2 Background

Any product that needs a manual to work is broken.

Elon Musk

In this chapter we want to make a short introduction to *COMET*. As we mentioned in the previous chapter, one should definitely read [Mos17] to get a good grasp over *COMET* and understand how it works. However, we will still try to briefly explain how *COMET* works and go over some of the more important components and things on which we will work on in this paper.

*COMET* is a web based application for Compliance Testing. It makes use of the JHIPSTER [Jhi] stack, which uses the Spring Boot Framework [Spra] for the Back-End application and the Angular [Ang] framework for the Front-End application. All of the data is persisted in a MySQL [Mys] database, because the test data is structured in a relational way. The *JHIPSTER* stack was used as a base for *COMET*, because it provides a lot of functionality out-of-the-box without the need for any configuration whatsoever. In addition to speeding up the entire development process, it also helps make *COMET* a cross-platform application. Since *JHIPSTER* uses the *Spring Boot* framework for the Back-End application and *Spring Boot* is *Java* Web Framework, this makes it possible to run *COMET* on pretty much any Operating System, for which the Java Virtual Machine (JVM) is available.

The actual testing however is performed with the help of the *InSpec* [Ins] Testing Framework. With *InSpec*, the code for the tests can be written either in the *InSpec* Domain Specific Language (DSL) or in the *Ruby* programming language. The real magic of this testing framework however, is in the way it executes tests. It can not only test the system on which it is currently running on, but can actually connect via SSH or Windows Remote to a target System Under Test (SUT) and perform the actual testing without the need to install any additional software. In addition to that, *InSpec* offers Profiles, which are a way to structure our testing files. One can create such Profiles to basically test different components of a system. For example, a popular Profile is the *Linux Baseline Profile*, which tests the security configurations of a Linux Distribution.

The ability of *InSpec* to run tests on a system without the need to install any additional software, the Profiles, with the help of which the actual compliance tests are generated and structured and also the very good and easy to use *DSL* were the 3 main reason for choosing *InSpec* as a testing framework for *COMET*. To this date, there is still no better alternative to *InSpec* and that's why in is still used in *COMET*.

But how does *COMET* actually work, one might ask? That's exactly what we want to explain now. In *COMET* a user can define the following things: *Compliance Tests*, *Compliance Rules*, *Compliance Rule Sets*, *Compliance Profiles*, *Software Components*, *Software Landscapes*, *Customer Projects* and *Environments*.

The *Compliance Tests* contain the actual test code written in the *InSpec DSL* or in Ruby. These tests can be reused in different *Compliance Rules*, as the user can use Placeholders for variables, which might change depending on where the test is used. These Placeholders can then be overwritten at 4 different stages: First in the *Compliance Rule*, then in the *Software Component*, after that in the *Software Landscape* and finally the *Customer Project*, with each overwriting having a higher priority than the last one, i.e. overwriting a placeholder in the *Customer Project* will always have a higher priority than any other. To give an example for a test written in the *InSpec DSL* and containing a Placeholder, we present the following code which tests if a given file exists:

```
describe file('<% fileName %>') do
  it {should exist}
end
```

As one can see, we have a Placeholder for the name and actually the path of the file, denoted with *<% fileName %>*, which is the standard notation for placeholders. By using a Placeholder Value for our filename, the test can be reused as often as we need, without the need to change the actual code of the test.

After we have our *Compliance Tests*, we have to combine them in *Compliance Rules*. Each *Compliance Rule* can have multiple *Compliance Tests*, but should test only one thing, or as the name suggests, one rule. For example, a *Compliance Rule* can check if a given file exists and if the user has permissions to modify this file. This can be achieved with the help of 2 *Compliance Tests*: one that checks if the files exists and one that checks if the user has the permissions. Similar *Compliance Rules* can then be combined in *Compliance Rule Sets* and further into *Compliance Profiles*.

One can then group these *Compliance Rules, Rule Sets and Profiles* into *Software Components*, or as called in the *COMET* Front-End, *Solutions*. These *Software Components* can be further combined together to form *Software Landscapes* or, as again, called in the *COMET* Front-End, *Complex Solutions*. A good example for a *Software Landscape* is the *LAMP (Linux, Apache, MySQL, PHP)* stack, which consists of the *Software Components Linux*, *Apache*, *MySQL* and *PHP*. Each of these components, can then have multiple *Compliance Rules, Rule Sets and Profiles*. For example, there might exist a *Compliance Profile* that checks if the *Linux* Distribution is configured securely and can run the web server and there might exist a *Compliance Profile* that checks the configuration of the *Apache* web server.

Finally, one has to group the desired *Compliance Rules*, *Compliance Rule Sets*, *Compliance Profiles*, *Software Components* and *Software Landscapes* in one *Customer Project*, which pretty much describes the entire SUT, or more precisely, the things that have to be tested for compliance in this system. The address of this system, the username and the password (or the SSH keys) with which the system can be accessed, are defined in an *Environment* instance, which is then also added to the *Customer Project*. Further in this work we might use the terms *Environment*, *Node* and *Machine* interchangeably, because in the context of *COMET* they mean the same thing: 1 computer. A *System* however, might consist from more than 1 computer, so it shouldn't be mistaken with an *Environment*, a *Node* or a *Machine*.

Another thing worth pointing out, is that a *Job* in the context of *COMET*, is when we start the testing for a given *Customer Project*. *Jobs* are represented by special entities, which have an ID, contain the *Customer Project* that is being executed, have a state, and also a *Job Result*. This means that, when we say that we are running a *Customer Project*, we mean that we start a *Job* for it.

This is pretty much a short summary of everything that one should know about *COMET*. It is not by any means a comprehensive introduction, but we will explain in more detail other things that one should know, when we need them. For now, we want to finish the introduction by presenting the current domain model of *COMET* in figure 2.1, because we are going to reference it and change it in order to solve the problems listed in the next chapter.

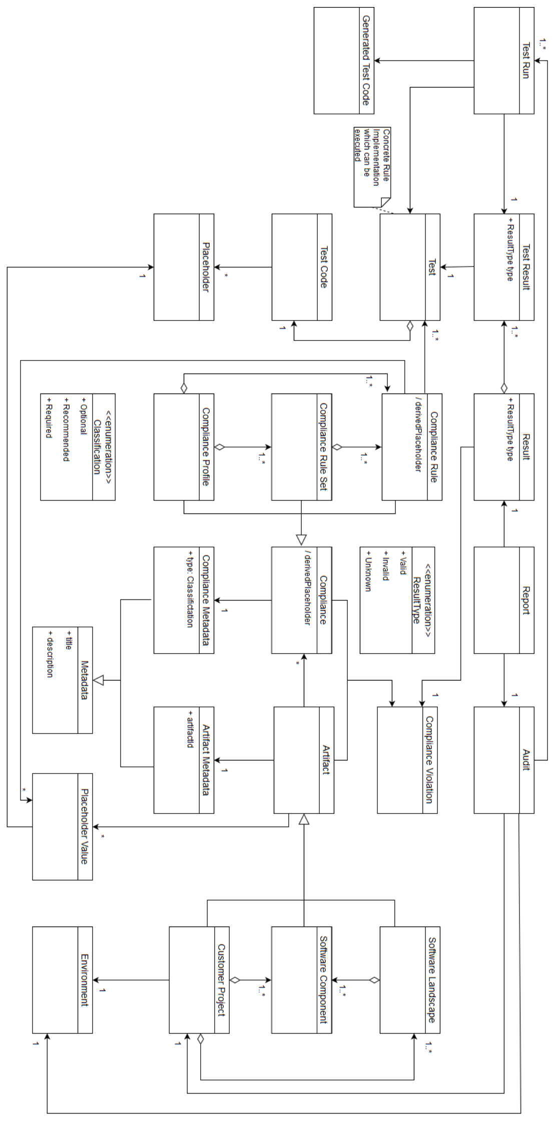


Figure 2.1: Current Domain Model of COMET

# 3 Problem Statement

Don't panic!

DOUGLAS ADAMS

While *COMET* might be an innovative system for Compliance Testing, it is certainly not perfect. It was created by a single person, for a limited amount of time, which combined with the complexity of *COMET* leaves a lot of things to desired. That's why in this chapter we want to lay out some of the problems and limitations that *COMET* currently has. Of course, we cannot cover everything, but we will try to present some of the major problems and limitations and then explain why we picked 2 of them to work on in this paper.

The first limitation that *COMET* currently has is that there is currenly only 1 supported testing framework: *InSpec*. The reason for that is however rather simple: There aren't many testing frameworks available that can do the things that *InSpec* can. The other popular framework that does pretty much the same things is ServerSpec. And as one might guess from the name, *ServerSpec* and *InSpec* are related with *InSpec* being based on *ServerSpec*. This makes *InSpec* an improved version of ServerSpec and eliminates the need for the latter. A more detailed explanation of why *InSpec* was chosen for the task and what other options are available, can be found in [Mos17].

But why not create our own COMET DSL and a COMET Testing Framework someone might ask? The answer for that is, again, rather simple: It's just not worth it at this point. Creating a custom DSL and a custom testing framework will require a lot of time and effort just in order to get to the current state of *InSpec*. And the thing is that *InSpec* is not currently limiting the functionality of *COMET* in any way. Still, it will be nice to have a custom DSL and a custom testing framework, because they can be optimized and adapted more easily to the needs of *COMET*. The task of creating these is however not a one man job and is beyond the scope of this theses. That's why we are leaving this problem open for now.

The second limitation of *COMET* is that it is currently only a Compliance Testing Tool. Other non-functional types of testing, like Performance Testing, are currently not supported. And while the original idea for *COMET* was for it to be exactly that, turning *COMET* in to a multi-purpose tool that can perform multiple types of testing is something to be desired. The integration of performance testing tools like Apache Jmeter [Apa], Gatling [Gat] and also the integration with Security Scanners like OpenVAS [Ope] is another open topic. These features however are all nice to have and there is no real urgent need to add them to *COMET*.

Another open problem is the fact that *COMET* can curretly only test remotely accessible systems. This is actually the first problem on which we are going to work on in this thesis: Making *COMET* portable. As we mentioned in the previous chapter chapter 1, *COMET* currently makes use of *InSpec* for the testing, which connects to the target SUT via remote connection and performs the testing. If, however the target SUT is not accessible via remote connection, *COMET* has to be deployed on the target SUT itself in order for the testing to be possible. This is not an ideal solution, because although *COMET* is a cross-platform application, it is a relatively big one, requires a database connection and is also relatively heavy on system resources. This can easily turn in to a problem and a solution for it is needed. That's why we are going to create a small tool, *COMET-P*, which is going to solve that problem. We are going to discuss all of the requirements for this tool in chapter 5.

The second problem that we want to solve is the lack of option to test complex and distributed systems. Currently *COMET* can only execute 1 *Customer Project* on 1 machine at a time. That means, that if we want to execute the same test on 10 systems, that have to be configured in exactly the same way, e.g. the computers at the checkout in a supermarket, we have to run 10 different Jobs with the same tests. This problem however goes even further, because 1 Customer Project can actually contain many *Software Landscapes* or *Software Components*, the tests for which in reality might need to run on different nodes of the system. What that means is that, if we have, for example, a typical web application with a micro-service [Mic] architecture, i.e. the Front-End runs on 1 machine, the Back-End on a second and the database on a third, with each one represented by a *Software Landscape*, and we want to test the system as a whole, because ultimately that is how the system must run, we can't simply do this. While the definition of different *Software Landscapes* and *Software Components* in *COMET* is possible, we cannot distribute these to different nodes/machines. This is a serious limitation, because while *COMET* can test systems that need complex configuration, it can currently test only trivial systems, i.e. ones that basically have 1 node. Again, how we plan to solve this problem and the exact requirements for this it, will be considered in chapter 5.

We think that these two problems go together very well, because both the portable tool and the ability to test distributed systems, will make *COMET* more versatile and a lot more capable. There are a lot of systems that need to be tested for compliance, but are not accessible via remote connection and there are even more distributed systems, some of which might also not be accessible remotely. By solving these two problems, *COMET* will be able to run Compliance Test on basically every system for which the *InSpec* testing framework and the *JVM* are available.

# 4 Related Work

Good artists copy. Great artists steal.

VARIOUS

Contents

Saying that finding work directly related to ours was easy, wouldn't be true. Although *Compliance Testing*, which is also known as *Conformance Testing*, is not a new idea, information and publications about it are limited and software products that perform *Compliance Testing* are even less common. [Mos17] did a good job finding related work for *Compliance Testing*, but we had more difficult time, because we had to focus on *Compliance Testing of Distributed Systems*, which seriously limits our search results. Nevertheless, here we present all the relevant work that we managed to find.

Because we are solving two different problems in our thesis, we will try to present related work for both of them separately. The reason for that is that our portable tool, *COMET-P*, is not conceptually new and doesn't have more functionality compared to *COMET*. That's why, we are going to present only design patterns for it, with the help of which our new tool can be integrated with *COMET*. For our second problem, we will present other software products that do similar things to *COMET*.

## 4.1 COMET-Portable

Starting with the related work for our portable tool, we had to decide how to integrate it with *COMET*. The first option is to use a *Shared Database*, [HW03], in order to transfer the information about the *Customer Projects* and the *Jobs* between *COMET* and *COMET-P*. This, however, would require for *COMET-P* to have access to the same database as *COMET*, which would be an issue, if we want to test a system, which doesn't have an Internet connection. Also, if we choose this option, *COMET-P* should also support the *Placeholder Resolvers, Artifacts* and so on, because it has to generate the test files on it's own. Generating the test files and *COMET* and using a *Shared Database* wouldn't simply make sense. That's why, the only possible and reasonable way to transfer information between *COMET* and *COMET-P*, is to use a *File Transfer* for the application integration [HW03], which would allow us to only transfer the generated test files.

In order for the *File Transfer* to work properly, we have to specify a *Data Format* for these files. Fortunately, since we are working with *Java Objects* for our *Customer Projects, Jobs* and *Environments*, we could make use of *Serialization* and *Deserialization* of *Objects*. This would allow us to have a consistent *Data Format* without the need to do some extra conversion.

Finally, we would like to reuse as much from *COMET* as possible and basically have a tight coupling for the shared components. The reason for that is that we want our *COMET-P* tool to have as much of the functionality of *COMET* as possible.

## 4.2 Compliance Testing of Complex and Distributed Systems

For our second problem, we present the following software products that solve similar problems to *COMET*.

### Sentinel

*Sentinel* is a language and framework for policy built to be embedded in existing software to enable fine-grained, logic-based policy decisions. A policy describes under what circumstances certain behaviors are allowed [Sen]. It also offers its own, easy to use DSL, the *Sentinel DSL*, just like *InSpec*, in which the policies have to be written. And although the idea behind *Sentinel* is more or less to embed policies into software, it can be also used to test system behaviour, which is what Compliance Testing does. However, as far as we know, *Sentinel* doesn't provide the option to test distributed systems in the way in which we plan to do it with *COMET*. Distributing the *rules*, that one defines in a *Sentinel* policy, to different machines might be theoretically possible, but there is no clear information on how to do that. In addition to that, *Sentinel* only offers a CLI tool without any graphical interface. These problems might be solved in the future, because *Sentinel* is a relatively new product and, at the time of writing this, is less than 1 year old, but for now, in its current state, it doesn't fully fulfill our needs.

### UpGuard Core

*UpGuard Core* simplifies security and compliance for your infrastructure in the cloud and on premises [Upg]. *UpGuard Core* really offers a lot of features and in addition to testing, it can also monitor different nodes. It can also show the differences in system configuration over time and with that make it easier to find the source of a problem. *UpGuard Core* is designed with the Cloud in mind, so it solves our problem with the compliance testing of distributed systems. Unfortunately, *UpGuard Core* is not free, not open source and doesn't offer a free demo or a trial. The idea behind this licensing is that primarily enterprises with a lot of servers with a lot of nodes are going to use such a product, because it doesn't make sense to use such a complicated piece of software for a small system. And since *UpGuard* lists *NASA* and the *New York Stock Exchange (NYSE)* as some of their customers, we believe that *UpGuard Core* is great. Due to the issue with the license however, it simply doesn't help us to solve our problems.

### Chef Automate

Chef Automate helps overcome complexity to build, manage, and deploy better, faster, and safer [Che]. *Chef Automate* includes the *InSpec* framework, which we also use in *COMET*. However, as explained in [Mos17], *Chef Automate* and *InSpec* alone don't solve two of the problems that *COMET* does, *Information Management* and *Management Inexperience*. They only help us to test for *System Misconfiguration*, which is not sufficient, because the idea behind *COMET* was to solve all 3 problems [Mos17]. In addition to that, *Chef Automate* cannot distribute different tests to different nodes they way we plan to do it. This makes *Chef Automate* also unsuitable for our needs.

# 5 Concept

There are no facts,
only interpretations.

FRIEDRICH NIETZSCHE

Contents

Since we are working on two different problems in this thesis, this chapter is also going to be split in two parts. First, we are going to consider all of the problems and the requirements for the portable version of *COMET* and then in the second part, we explain all the things that have to be extended and modified in order to make the testing of complex and distributed systems possible.

## 5.1 Portability

At first glance creating a smaller and portable version of *COMET* might not look like a big challenge. And that's exactly the case, if we only want to create a very simple version that can only execute the tests on a different system and then import back the results. For such a tool, one can probably even use Bash or some other type of Shell scripts, since once all of the test files are available on the targeted *SUT*, one has to only execute the InSpec command. But we are not interested in such a solution. We want a more sophisticated tool that doesn't only have this basic functionality. Our tool should be able to grow and provide more functionality with time. For example, the portable tool should be able to also test distributed systems that are not accessible via remote connection at all. This means that an instance of it should run on every node of the distributed system. Such a scenario will require flexible configuration and the ability for each running instance to be controlled remotely from *COMET*. With all of that in mind, let's take a look at the requirements for this new tool, which we are going to call COMET-Portable (COMET-P).

### 5.1.1 Adapting COMET

The first thing that we have to consider is how are we going to adapt *COMET* in order for it to start supporting the export of tests and also the import of test results. In addition to that, at some stage *COMET* should be able to “control” multiple instances of *COMET-P*. These instances of *COMET-P* will basically be Workers. In order to explain the current testing work-flow, we present the following diagram in figure 5.1.

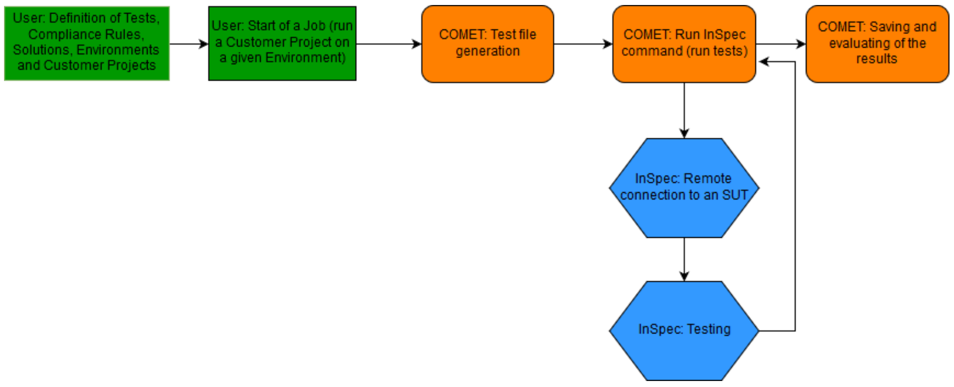

User: Definition of Tests, Compliance Rules, Solutions, Environments and Customer Projects
User: Start of a Job (run a Customer Project on a given Environment)
COMET: Test file generation
COMET: Run InSpec command (run tests)
COMET: Saving and evaluating of the results
InSpec: Remote connection to an SUT
InSpec: Testing

Figure 5.1: Current Test Execution Flow in COMET

As one can see, the first stage is, of course, the definition of *Compliance Tests, Compliance Rules, Rules Sets* and *Profiles*, *Software Components* and *Software Landscapes*, which are also called *Solutions* and *Complex Solutions*, the target SUTs/Environments and finally *Customer Projects* by the user. After all of these are defined, one can execute the *Customer Project* as a *Job* to basically test the target system. The things that currently happen when the user presses the “Start Job” button in the *COMET* Front-End, are as follows:

- A REST Request with the information describing for which *Customer Project* a *Job* has to be created is made to the *COMET* Back-End.
- A *Job* is created for this *Customer Project*, marked as *QUEQED* and saved in the Database.
- The now created *Job* is started asynchronously in a new Thread, so it doesn’t block any other parts of the application, and is marked as *RUNNING*. This makes it possible to run multiple *Jobs* at a time.
- After the *Job* is started, the *Job* directory which is going to contain the InSpec test profile is created in the */jobs* directory.
- Then, all of the test files are generated. Before the actual file creation, a Placeholder Resolving is performed. Currently *COMET* provides Interfaces with methods for Placeholder Derivation and Resolving, which have to be implemented by every test

plug-in. This is exactly the case with InSpec. So, after all of the Placeholders are substituted, the actual files are created with the help of templates.

- After generating the files, *COMET* performs a connection test to see if the SUT is reachable and if so, it executes the InSpec's *check* and *exec* commands.
- The first command, *check*, checks if the InSpec test profile is valid. If so, the process continues to the next step, which is the actual testing.
- When the *exec* command is executed, InSpec connects via SSH or WinRM to the target system, performs the testing and then returns the result in either JSON or XML. In our case, it was opted for the results to be in JSON.
- Then, the results are saved in the database. *COMET* currently has 2 Entities which are designed for the saving of the results. These are *Job Result* and *Job Result Item.* The *Job* entity has a One-To-One relationship to *Job Result* and *Job Result* has a One-To-Many relationship to *Job Result Item.* At the end, for each *Job* we currently have exactly 3 results or *Job Resul Items*: 1 from the connection test, 1 from the check of the InSpec profile and 1 from the actual test. After the results are saved, the *Job* is marked as finished and the results from it can be seen by the user in the Front-End.

As one can see, if we want to support the export of tests, one should simply stop the execution process directly after the files are generated. The *Job* should be then marked as *EXPORTED* until the test results are imported.

In addition to the InSpec test profile files, we will also need to export all of the information about the *Job*, including the information about the *Environments* that have to be tested, and save it in a file in the *Job* directory. Since some *Environments* are accessible via SSH only with the help of a public and private key pair, *COMET* currently generates these keys when an *Environment* is defined and then offers an authentication-free link from which the public key can be downloaded on the target SUT. This is a very useful feature, but if we want *COMET-P* to be able to access the systems on which this public key is downloaded, we have to transfer the private key between the systems on which *COMET* and *COMET-P* are running. Private SSH keys however can be a very sensitive information and because of that we won't provide a method for transferring them. We leave this operation entirely to the user of *COMET-P*. He/she must generate it's own public and private key pair and configure the target SUT for remote connection.

After all of these actions are performed, we will need a way to transfer the test files between *COMET* and *COMET-P*. The 2 possible options are:

- Manually Copy & Paste the files from the system on which *COMET* is running to the system on which *COMET-P* is running.
- Create a REST Endpoint from which the files can be downloaded in the form of a simple ZIP file. This is actually our preferred way for exporting the test files, since the overhead for it is very minimal.

Finally, the test results have to be imported to *COMET*. For this we need 2 different methods which will actually work in exactly the same way. The first one is via a REST request. Since we already have entities for saving the test results, with the help of JSON serialization and deserialization we can directly make a REST request with the results and save them. The second method is to import everything manually through the front-end. This will require for us to extend the front-end slightly. The nice thing here is that a manual import of the test results will also use the same REST endpoint and won't create a big overhead.

### COMET-Portable

Now we want to present all of the requirements for *COMET-P*.

Since *COMET* is already a cross-platform application, *COMET-P* has to also be able to run on most modern systems. It has to be small, lightweight and portable, which means that we need a command line tool (CLI). While a Graphical User Interface (GUI) will be a nice addition, a *GUI* only application won't be a good choice since there are a lot of systems that don't have a user interface, e.g. Linux Servers.

Furthermore, *COMET-P* should be at least able to make REST requests to at least one instance of *COMET* in order to download the test files and information and also to upload the test results. Ideally, it should also be able to accept REST request for easier configuration and control. This will basically turn *COMET-P* into a web application. However, the ability to configure and more importantly control *COMET-P* from *COMET* should be also available even if the SUT doesn't allow incoming HTTP request, i.e. ports 80, 443 or any others are not open and running a web server is not possible.

The configuration problem can be easily solved by using a simple configuration or *ini* file to store the settings. This file can then be modified by the user with the help of a simple text editor.

The control of *COMET-P* from *COMET* however, will be a bit more complex. If *COMET-P* cannot be accessed at all by *COMET*, a control with HTTP GET requests in a typical *Polling* fashion should be possible. While this might not be the most efficient way to control an instance of *COMET-P*, because it will create an additional overhead, since we need to implement this functionality for both *COMET* and *COMET-P*, it will give us a lot more flexibility and will enable the full functionality of *COMET* on a lot more systems.

As for the actual functionality that *COMET-P* needs, it has to implement at least the following features:

- It should be able to list all of the *Jobs* that are marked as *EXPORTED* by one or more instances of *COMET* to which *COMET-P* is currently connected. If the portable tool is connected to more than one *COMET* instances and there are *Jobs* with the same Id, the listing should clearly visualize that. An example command for that will be a simple *list* with an additional parameter with the help of which the user can specify a concrete instance of *COMET*, from which the *exported Jobs* have to be listed.

- It should be able to download the ZIP files with the *exported Jobs* and unzip them automatically. In the case of matching *Job* Ids from different instances of *COMET*, the user should be able to choose which one to download. An example command for that will be a simple *download* with an additional parameter for the *COMET* instance.
- It should be able to run the *Jobs* exactly like *COMET* does. An example command for that will be a simple *run*.
- It should be able to generate a JSON file with the test results, which can be imported back manually to *COMET*. An option to import the results via REST request to the instance of *COMET*, from which the *Job* was downloaded, should also exist. In addition to that, it should be possible to print the test results on the screen, so the user can see them. For these features we need both an additional command and also optional parameters for our *run* command, so the results can be saved to a file, printed and/or imported to *COMET* directly after running the test. An example for the additional command that we will need, so the user can see the results, which are saved in a JSON file and also to be able to import them to *COMET* after the test has finished, is a simple *res*. A few optional parameters like -*m* for printing the raw message that InSpec returned after running the test and -i for importing the test results are also going to be needed.
- Finally, an *authenticate* or *login* command will also be needed, since *COMET* is RESTful web application with enabled security. This means that a user needs to login first, before he/she can access the REST endpoints for listing and downloading *Jobs* and importing of test results.

This sums up pretty much the entire functionality that *COMET-P* needs for its first version. Implementing these features will gives us exactly the tool that we need and will help us to solve the described problem. With our new tool, the entire test execution flow will look as visualized in figure 5.2.

## 5.2 Compliance Testing of Complex and Distributed Systems

The second problem that we want to solve in our work is to extend *COMET* and make it possible to test complex and distributed systems for compliance. Before we go further, we want to clarify what our definition for a “complex” system is. A complex system in our case is a system that has more than 1 node/computers and multiple *Software Landscapes* and/or multiple *Software Components*. If a system has more than 1 nodes and only 1 *Software Landscape/Software Component*, we still consider it to be a “complex” system, because it doesn’t fit our previous 1 *Job*, 1 *Job Result* model.

In order to make such a testing with *COMET* possible, we have to solve, yet again, 2 problems. The first one is to simply implement the possibility to test a system with multiple nodes. For this part, we don’t need any advanced testing options. We are

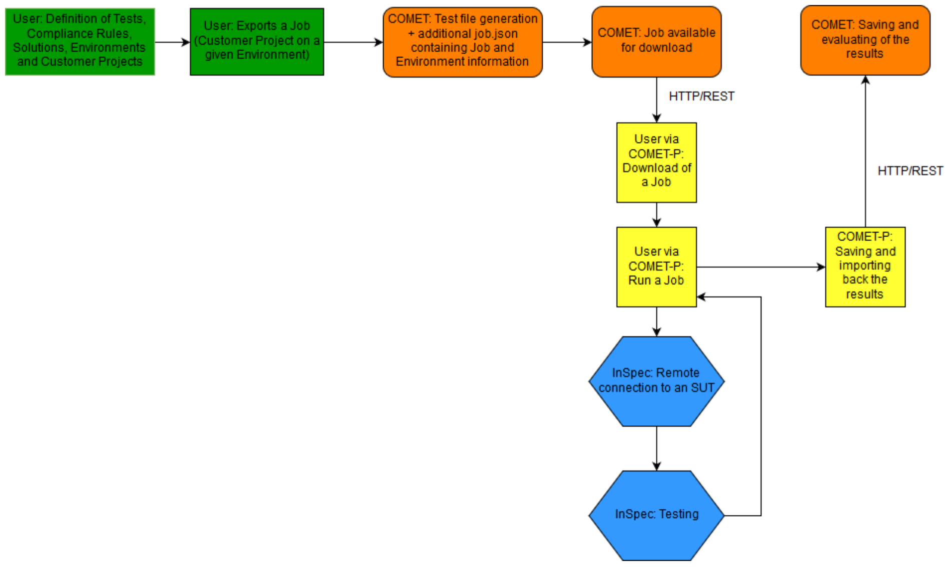

User: Definition of Tests, Compliance Rules, Solutions, Environments and Customer Projects
User: Exports a Job (Customer Project on a given Environment)
COMET: Test file generation + additional job.json containing Job and Environment information
COMET: Job available for download
COMET: Saving and evaluating of the results
HTTP/REST
User via COMET-P: Download of a Job
HTTP/REST
User via COMET-P: Run a Job
COMET-P: Saving and importing back the results
InSpec: Remote connection to an SUT
InSpec: Testing

Figure 5.2: Test Execution Flow of Exported Jobs

leaving these for the second part in which we are going to implement the possibility to distribute *Software Components* and *Software Landscapes* on to different *Environments*. That means that we are going to provide the user with the option to map different *Software Component/Software Landscape* to different nodes.

### 5.2.1 Testing of multiple nodes at once

For the testing of multiple nodes in one *Job*, we actually don't have to do a lot of work. Literally all of the changes that we have to make are reflected in our updated domain model in figure 5.3.

These include only the following things:

- We have to add the option for a *Customer Project* to have more than one *Environment*, i.e. turn our One-To-One relationship to a One-To-Many relationship. This, unsurprisingly, was already implemented in *COMET*. The change was not reflected in the domain model in figure 2.1 however, because the rest of *COMET* didn't support the functionality to test multiple systems at once, so it was denoted as a One-To-One relationship in order to correspond to what *COMET* can actually do.

- The second change that has to be made is to add the option for a *Job*, or as denoted in the domain model, a *Test Run*, to have multiple *Job Results/Test Results*. This is of course a very obvious thing since the number of test results will be equal to the number of systems tested. In order to achieve that, we have to again turn the One-To-One relationship between *Test Run* and *Test Result* to a One-To-many relationship.

- The last change is also a change in a relationship. The One-To-One relationship from *Report* to *Result* has to become a One-To-Many relationship. This might not be as obvious as the other two changes, but the reason for it is quite simple: Currently a report is generated based on a test result. Since now we have multiple nodes on which we run the tests, we also have multiple results, so we need multiple reports. One might wonder why not create a single report for all systems, but that wouldn't be accurate, because running the same *Job* on multiple *Environments* at once, doesn't require the nodes to be related at all and to even belong to the same system, so creating a report based on the results of all systems doesn't make sense.

Of course, there are also other changes that have to be made in order for *COMET* to continue working after we change the domain model, but these all involve implementation specific details, which we will present in chapter 6.

### 5.2.2 Distribution of Software Landscapes and Software Components to different nodes

By being able to distribute *Software Landscapes* and *Software Components* to different nodes, we would be able to perform compliance testing on distributed systems and on systems with complex configuration. So, naturally we would like this feature to be possible. Currently, if we have a system with multiple nodes, which have different *Software Landscapes* and *Software Components* installed on them and also require different configuration, we basically have to create a *Customer Project* for each node and then run these as different *Jobs*. There is no option to define a *Customer Project* containing all *Software Landscapes* and *Software Components* and then to choose which ones belong to which node. This is a serious limitation which we want to remove.

To solve this problem, we came up with 2 possible solutions, one of which ended up being better than the other. Now we want to present them and discuss their advantages and disadvantages.

The first idea that we had, was to have *Customer Projects* defining the *Software Landscape* and *Software Components* for 1 or more nodes with the exact same configuration and then to group these *Customer Projects* into a *Customer Project Set*. This is basically the same idea as the grouping of *Compliance Rules* into *Compliance Rule Sets*. The advantages of this approach are that it is simple to implement and that it allows us to reuse *Customer Projects* in different systems. Unfortunately, the main and probably the only disadvantage of this approach is that it requires for all nodes in the *Customer Project* to have exactly the same configuration. This might not seem like a big problem at first, but can easily turn into one for systems with a lot of nodes with different configurations. For example, if we have a system with 20 nodes, which all have different *Software Landscapes* and *Software Components*, we would need to define 20 *Customer Projects* and then combine them in one *Customer Project Set*. This, as one can guess, is not ideal, because in reality it will work only for systems with low number of different nodes. The definition and possibly later modifications of *Customer Projects* will come out of control, because the user will have to keep track of all *Customer Projects* contained in the *Customer Project Set.*

That's why we needed a better solution for this problem. The second idea that we had and that we ended up using, is to create a *Mapping* between the *Software Landscapes* and *Software Components* defined in a *Customer Project* and the *Environments*/nodes of this *Customer Project.* With this *Mapping* we allow the user to choose which *Software Landscapes* and *Software Components* to run on which node. That means that now we can have one *Customer Project* combining all *Software Landscapes* and *Software Components* contained in the system and with that basically define and entire complex/distributed system with only one *Customer Project.* Then, for each node the user can pick only the landscapes and components that the node constains. This is exactly the functionality that we need and that we want. The only real downside of this approach is that it a lot harder to implement than the first one. We are going to provide a detailed explanation of how we implemented this solution in section 6.2.2, but before that we want to present the solution on a conceptional level.

The things to consider with this solution was how to define this mapping and where it makes sense to actually put the options for it. The latter one was, as one might guess, is a lot easier. We had pretty much 2 options for the places to which we can add the menu for the mapping: In the definition of the *Customer Project* and in the *Compliance Run* page. We ended up choosing the first option, because it allows us to persist the mapping and reuse it.

Deciding how to define this mapping was actually a lot harder. After some considerations, we figured out that it will make the most sense to implement the mapping with the help of 2 new entities: *Mapping* and *Mapping Entry.* The *Mapping Entry* entity is the one that is used to describe which *Software Landscapes* and *Software Components* should run on the environment/node. This means that each *MappingEntry* instance should be defined for 1 *Environment* and should have multiple *Software Landscapes* and/or multiple *Software Components.* These *Mapping Entries* are then combined into 1 *Mapping* via a One-To-Many relationship. After that, we introduced the constraint to have 1 *Mapping* per *Customer Project.* This was achieved with the help of a One-To-One relationship. What this means in reality is that the user can choose to define and have a *Mapping* for the *Customer Project*, which will consist of different *Mapping Entries.* Each *Mapping Entry* will then define which *Software Landscapes* and *Software Components* are contained on which node. If the user doesn't define a mapping, then the tests for all *Software Landscapes* and *Software Components* will be run on each *Environment.* If on the other hand, the user creates a *Mapping Entry* only for some of the *Environments* and leaves the others without one, the ones for which a *Mapping Entry* doesn't exist, will be tested with the tests for all *Software Landscapes* and *Software Components* defined in the *Customer Project.* Finally, we had to introduce one more constraint in the relationship between *Mapping Entry* and *Software Landscape* and in the relationship between *Mapping Entry* and *Software Component.* Both of these relationships have to also have a constraint to *Customer Project*, since a *Mapping Entry* can contain only the *Software Landscapes* and *Software Components*, which are defined in the *Customer Project.* One can see how these

2 new entities interact with the rest of the domain model in the updated domain model in figure 5.4.

As sparse as it might seem, this pretty much sums up the conceptional part for the solution of this problem. As we already mentioned, the implementation is quite complicated and we will describe it in full detail in the corresponding chapter.

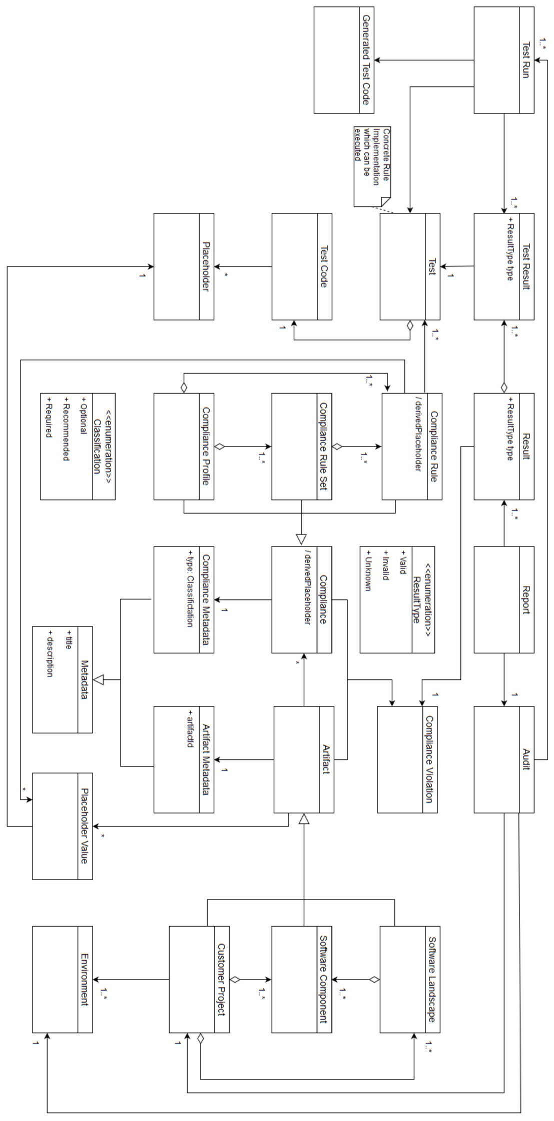


Figure 5.3: Domain Model of COMET supporting the simultaneous testing of multiple nodes in one Job

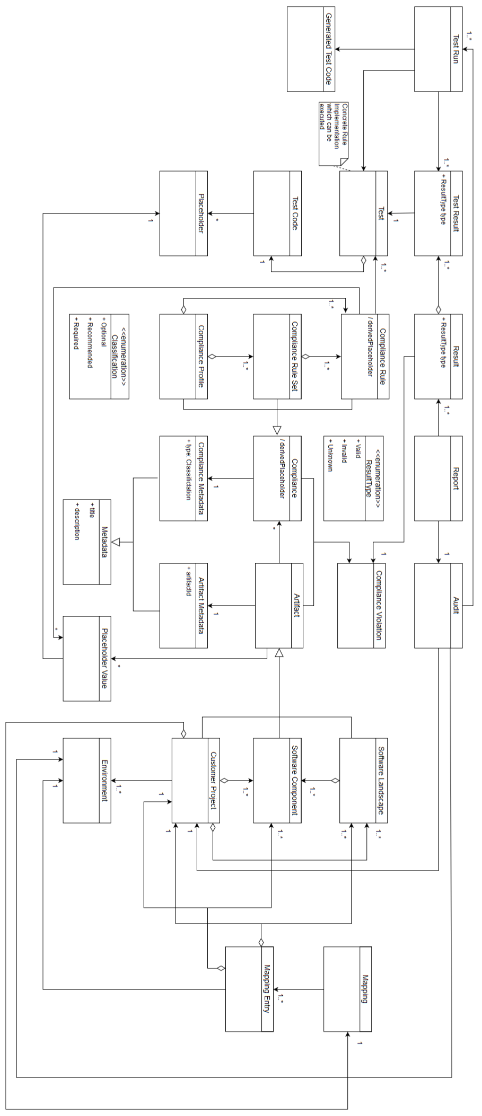


Figure 5.4: Domain Model of COMET supporting a Distribution (Mapping) of Software Landscapes and Software Components on Environments

# 6 Realization

I don't know
if it's what you want,
but it's what you get. :-)

Larry Wall

Contents

As with the previous chapter in which we conceptualized our ideas for all the features and improvements, we are going to also split this chapter in two parts. In the first one, we are going to take a look at all of the additions and small changes that we had to make to *COMET* in order to support the export of *Jobs* and the import of *Job Results*. After that we will explain how we have created our portable tool, *COMET-P*, and how it works. Then, in the second chapter, we present how we have extended and modified *COMET* and also *COMET-P*, so they can test complex and distributed systems.

## 6.1 Portability

To keep the same structure as in chapter 5 and also because this was the actual order in which we had to implement our solution, we first present how we've adapted *COMET* to support the export of *Jobs* and the import of *Job Results* and then explain how we've created our *COMET-P* tool.

### 6.1.1 Adapting COMET

Adding the support for these new features to *COMET* turned out to be a very simple job, which didn't require any major modifications to the system. This means, that we only had to add the new functionality without changing the old one at all.

So instead of changing the entire test execution flow from figure 5.1, we simply added the new one, which we visualized in figure 5.2. In order to do that, we gave the user the option to export the *Job* instead of running it, as one can see in figure 6.1. The new

"Export Job" button is the only visual difference, apart from the page for importing of the *Job Results*, that had to be made to the Front-End of *COMET*. After the addition of the button, a new REST Endpoint was created in the Back-End project for accepting of the new request types. Of course, the REST client in the Front-End was also extended to support this functionality.

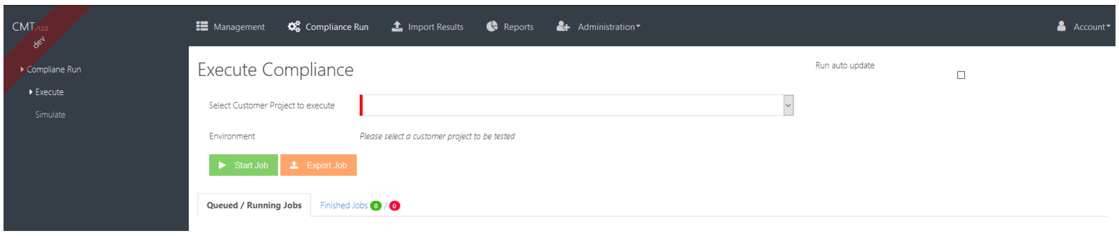


Figure 6.1: The new button for the export of Jobs

Then, we had to make sure to modify the Back-End further in order to not start the *Job*, but to only create it and generate the test files. *COMET* currently makes use of Java's Concurrency library in order to start *Jobs* asynchronously at the end of every REST request. To achieve that, there is an abstract *ComplianceRun* Task Executor, which implements the *Runnable* interface. This abstract task executor takes care of all standard procedures for *COMET* before starting a *Job*, like reading the entire information for the *Job* from the database. The *ComplianceRun* Executor is then extended by the *ComplianceExecution* one, which implements the needed for the *Runnable* interface *run* method. Here is how the *run* method of the *ComplianceExecution* class currently looks like:

```java
@Override
public void run() {

    this.setExecutionJob();
    this.job.setState(JobState.RUNNING);
    this.getJobRepository().save(this.job);

    File exportDestination = this.setExportFolder(this.job);
    TestRunner runner = TestRunnerFactory.getRunner();

    runner.setExportDestination(exportDestination)
        .setPlaceholderResolver(new PlaceholderResolveSimple());
    runner.execute(this.job);

    // update job properties
    this.job.setResult(runner.getResult());
    this.job.setState(JobState.FINISHED);

    this.getJobRepository().save(this.job);
}
```

Source Code 6.1: ComplianceExecution.java

As one can see, the things that happen in this method are as follows: the *Job State* is set to *RUNNING* and the *Job* is saved again to the database in order to update the state. After that, the export destination or the folder in which the test files are going to be generated, is configured and then the *Job* is started via the *TestRunner* interface and the *execute* method. In *COMET*, this interface has to be implemented by every test plug-in, as it is the case with *InSpec* and its *InspecTestRunner* class. The *execute* method is basically the entry point for the testing and it looks like this:

```java
@Override
public JobResult execute(Job job) {

    this.job = job;
    this.result = new JobResult();

    // check correct state of class
    if(this.exportDestination == null || this.resolver == null) {
        throw new IllegalStateException("Export Destination or
           PlaceholderResolver are missing, stopping execution");
    }

    this.generateTestCode();
    this.executeTests();

    return this.result;
}
```

Source Code 6.2: InspecTestRunner.java

Finally, after the *Job* is completed, i.e. the *InSpec* command has returned a result via the *execute* method, the results are saved in the database and the *Job* is marked as *FINISHED*.

In order to support the export of *Jobs*, we created a new Task Executor, *Compliance-Export*, which also extends the abstract *ComplianceRun* one, and which is called at the end of the REST request for exporting a *Job*. This new executor also has a *run* method, which currently looks like this:

```java
@Override
public void run() {

    this.setExecutionJob();
    this.job.setState(Job.JobState.EXPORTED);
    this.getJobRepository().save(this.job);

    File exportDestination = this.setExportFolder(this.job);
    TestRunner runner = TestRunnerFactory.getRunner();
    runner.setExportDestination(exportDestination)
        .setPlaceholderResolver(new PlaceholderResolveSimple());
    runner.export(this.job);

    // Generate the JSON with the Job and Environment Information
    this.generateJSONWithJobInformation(this.job);
}
```

Source Code 6.3: ComplianceExport.java

As one can see, we have made some changes in comparison to the *run* method in the *ComplianceExecution* class. The first one is quite obvious, the *Job State* is set to *EXPORTED*. In order to support this new state, we had to first add it to the *Job* entity. And since the *Job States* in *COMET* are listed as a basic *Enum*, we only had to add the new state as follows:

```java
public class Job {

    // ...

    public enum JobState {
        NEW,         // created and not yet queued
        QUEUED,      // queued in thread pool
        RUNNING,     // currently running
        FINISHED,    // job finished
        EXPORTED     // job is exported. waiting for the job results
    }

    // ...
}
```

Source Code 6.4: Job.java

The second change is that instead of the *execute* method of the *TestRunner* or actually the *InspecTestRunner*, we are calling a new *export* method, which looks like this:

```java
@Override
public void export(Job job) {

    this.job = job;
    this.result = new JobResult();

    // check correct state of class
    if(this.exportDestination == null || this.resolver == null) {
        throw new IllegalStateException(
        "Export Destination or PlaceholderResolver are missing,
            stopping export");
    }

    this.generateTestCode();
}
```

Source Code 6.5: InspecTestRunner.java

There is really no real magic in this method. We simply do not call the *executeTests* method here and that gives us exactly the desired behaviour: the test files are generated, but the *InSpec's check* and *exec* commands are not executed.

The third and final change in the *run* method is that we call an additional method, *generateJSONWithJobInformation*, which exports the entire information about the *Job* and the *Environments* from the database and saves it to a JSON file in the *Job* directory. We need this file, because our *COMET-P* tool also needs the *Job* information in order to able to execute the *Job*.

In addition to these extensions, we also had to create a few additional REST Endpoints, which provide information about which *Jobs* are currently marked as *EXPORTED*, offer the ability to download the *Job* as a ZIP file and to import back the *Job Results*. These Endpoints were basically added as a way for our *COMET-P* tool to communicate with *COMET*. They are all standard and pretty straight forward, so we omit further details about them in this document and encourage the reader to take a look at them himself/herself. The only thing that we want to point out, is that for the import of the test results, the *JobResult* POJO is deserialized to JSON by *COMET-P* and then serialized back to POJO by *COMET* with the help of the Jackson [Fas] library.

As far as the page in the *COMET* Front-End for the import of the test results goes, we didn't need anything more than a simple file upload form for it, which utilizes our existing REST Endpoint.

This sums up pretty much all of the changes that we had to make to *COMET* in order to support the export of *Jobs* and with that our new tool too.

### 6.1.2 COMET-Portable

Before we start with the details for the implementation of our tool, we want to say a few words about the technologies that we've used, why we've picked them and why we've created the tool this way.

As written in [Mos17], in addition to *COMET* there is also a *COMET-CLI* tool, which offers some of the functionality that the *COMET* Front-End has. Namely, it offers the 4 commands: *list*, *run*, *result* and *import*. The *run* command is, of course, used to start a new *Job* and the *result* command is used to show the results for a finished *Job*. The *list* command is used to list various resources, which is different to the *list* command that we've described in section 5.1.1. And finally, there is the *import* command, with the help of which one can import tests written in a *COMET-CLI* specific format to the *COMET* database. On first glance the *COMET-CLI* does almost everything that we've described and wanted for our *COMET-P* tool. So one might ask why not use the already existing tool and extend it, instead of creating a new one? There are however a couple of reasons for not working further on the *COMET-CLI*.

The first reason is that it wasn't designed with our usage scenario in mind. The *COMET-CLI* was created only as an alternative to the *COMET* Front-End, so it can be used together with *COMET* on systems without a Graphical User Interface (GUI). The *COMET-CLI* was never really written as a smaller, stand-alone version of *COMET* which can be deployed separately from *COMET* and that only needs a connection to *COMET* to download the *Jobs* and to import the *Job Results*.

This however, doesn't fully answer the question why we've chosen to create a new tool instead of extending the old one. Which brings us to reason number two: The *COMET-CLI* is written in Python. As we mentioned in chapter 3, the Back-End and with that the entire domain and functionality of *COMET* is written in Java with the help of the Spring Boot Framework. This means that the entire code of the *COMET-CLI* is written in a totally different way and is basically incompatible with the Back-End logic of *COMET*. And since we wanted to design a new tool that is small, lightweight, can work separately from *COMET* and can mimic most of the functionality, we had to create a new tool.

This leads us to the technology stack that we've used for *COMET-P*. Since we needed a *CLI* and we wanted to share as much things with *COMET* as possible, we opted again for the Spring Boot Framework in combination with the Spring Shell [Sprb] extension. This resulted in an interactive shell application that provides a lot of functionality out of the box like the incredibly simple way to create custom commands, auto-completion, error handling and others. In addition to that, we have all the benefits of the Spring Boot Framework, like the ability to create RESTful Controllers and a built-in REST Client. These two features alone are really important for *COMET-P* since they make the communication between our tool and *COMET* much easier. And since both *COMET* and *COMET-P* are written in Java and use the Spring Boot Framework, we were able to reuse a lot of the code from *COMET* for our new tool.

Now, we want to dive-in into the implementation details and explain what parts of *COMET* we were able to reuse, what we had to change and what we had to additionally develop in order to make our new tool work.

For our *COMET-P* tool we were able to reuse all of the elements that we need from *COMET's* testing APIs, including the *InSpec* testing plug-in. This also includes parts of the domain model, specifically the sub-domain containing the entities for the *Job*, *Job Result*, *Job Result Item* and *Environment*. The only thing here that we had to modify were the Java Persistence API (JPA) annotations. *COMET* makes use of the *JPA* for easier persistence of the data, but since our new tool doesn't use a database at all, we had to remove the annotations so we can compile the code. This unfortunately means that we cannot use exactly the same code and the same entities for both projects, but since the entities shouldn't change that often and do not change the functionality of the rest of the systems, we think that this is not a major problem.

As one probably noticed, we just said that we used "all of the elements that we need" from *COMET's* testing APIs and not all of the elements. That's due to the fact that *COMET* also offers *Placeholder Resolvers* or *Code Transformers*, which every test plug-in has to implement and which are basically used to substitute placeholders in the test code with actual values. However, the Placeholders are currently being resolved just before the test files are generated, so we don't need that feature in our *COMET-P* tool, because we are downloading the already generated test files. That's exactly why we opted to not include this functionality in our tool.

The other things that we were able to reuse is the already mentioned Task Executorts: the abstract *ComplianceRun* and the extension *ComplianceExecution*. With them, we didn't have to make any change whatsoever. They were able to work in our tool without any modification.

As far as the features described by us in section 5.1.1 goes, we were able to implement everything and here we present every command/function that *COMET-P* currently has. Each parameter will be noted exactly in the way *Spring Shell* displays the parameters, when the *help* command is used. This means that mandatory parameters will be noted in single square brackets, e.g. *[-j]* and optional parameters in double square brackets, e.g. *[[-i]]*.

- authenticate. This command, as the name suggests, is used in order for *COMET-P* to have access to *COMET* and in particular, to its REST APIs. Usually, if *COMET-P* is started after *COMET* and can reach it, the authentication process happens automatically and the user doesn't need to worry about it. Also, the access token is refreshed every hour, so it doesn't expire. However, for the cases in which *COMET-P* was started before *COMET* or *COMET* was restarted at some point and the access tokens are no longer valid, we provide a command for authentication, so the user doesn't have to restart *COMET-P* or wait for the next refresh of the access token.

- list [[-d]] or [[-r]] or [[-f]]. By default this command attempts a connection to *COMET* and tries to get the list with the exported *Jobs* and then returns them to the user. The user can also choose to list only the downloaded *Jobs* by using the flag *-d* or the *Jobs* that are currently running or have already finished by specifying the flags *-r* or *-f* respectively. Because of the way *Spring Shell* handles parameters and because the *list* command should list only 1 type of *Jobs*, it is possible to use only 1 optional parameter at a time and the priority of the parameters is exactly as listed here, i.e. by using the *-d* flag, one will always get the downloaded *Jobs*, no matter if any other flag is used before that. For example *list -r -d* will always return the list with the downloaded *Jobs*.

- download [-j]. This is a very straightforward command. It is basically used to download the test files from *COMET*. The mandatory *-j* parameter stands here, and also in the *run* and *res* commands for *Job ID*. Thanks to the way *Spring Shell* handles parameters, one can skip the flag when using the command and directly use the id. For example, if one wants to download a *Job* with ID 130, one can write *download 130* instead of *download -j 130*.

- run [-j] [[-i]] [[-l]] [[-k]]. As one can see, our *run* command has the most number of optional parameters, excluding of course the *-j* parameter which is mandatory. The *-i* parameter specifies if the *Job Results* should be imported back to *COMET* directly after the *Job* finishes. The *-l* parameter specifies that the *Job* has to run on the systems on which *COMET-P* is currently running. This is the only modification to the *Job* and *Environment* configuration that we allow. Since *Jobs* have to be basically immutable, because *COMET* expects the results exactly for the *Job* it has exported, our tool shouldn't change any of the test files. Of course, here we don't account for modifications that a user can make to the test files. We allow the change of the *Environment* with the local one, because if for example the user defines an *Environment* in *COMET* that is not reachable via remote connection and then exports the *Job*, downloads it on the target SUT and runs it with the help of *COMET-P*, with the default configuration, *InSpec* will attempt remote connection via WinRM or SSH. That's why we implemented the option to run the tests on the same system. And finally, the *-k* parameter. It is there so the user can specify a path to an SSH key file. This SSH key file is of course used so *InSpec* can connect to the target SUT.

- res [-j] [[-m]] [[-i]]. The *res* command provides the user with the option to see the test results for a *Job* that has finished. The *-m* parameter provides the user with a little more information and in addition to the *Job Results*, it also displays the *Executor Command* that was used in order to receive these results. The *-i* parameter is used here (as in the *run* command) to import the *Job Results* back to *COMET*. We provide this option for the cases in which the user forgot to use the flag in the *run* command and the test results are only available locally.

In addition to these commands, we've also implemented a simple *Job Monitoring Service*. It basically tracks the *Jobs* and their state, i.e. are they only downloaded, are they running and have they finished. This service also helps us prevent to run a *Job* multiple times. In *COMET* every *Job* can and has to run only once and with the *Job Monitoring Service* we made that possible in *COMET-P* too.

As far as the configuration of *COMET-P* itself goes, i.e. the URL of *COMET*, what username and password to use to authenticate with *COMET*, on which port to run and so on, we managed to export all of these settings to an *application.ini* file, which can be shipped together with the executable *JAR* file. This enables easy configuration, which can be modified with a simple text editor.

This sums up pretty much the entire functionality of *COMET-P*. The only thing that we didn't manage to complete due to time constraints, is the ability for *COMET-P* to be controlled and monitored via REST Requests. However, we think that we provided good suggestions on how this feature should be implemented and we leave is as a possible and desired future work.

Finally, we present the Architecture of *COMET-P* in figure 6.2.

## 6.2 Compliance Testing of Complex and Distributed Systems

### 6.2.1 Testing of multiple nodes at once

As we've listed in section 5.2.1, we had to make only 3 conceptional changes to *COMET* in order to support the test of multiple nodes in 1 *Job*. In addition to that however, some further things had to be changed in the *COMET* Front-End and also in the Back-End

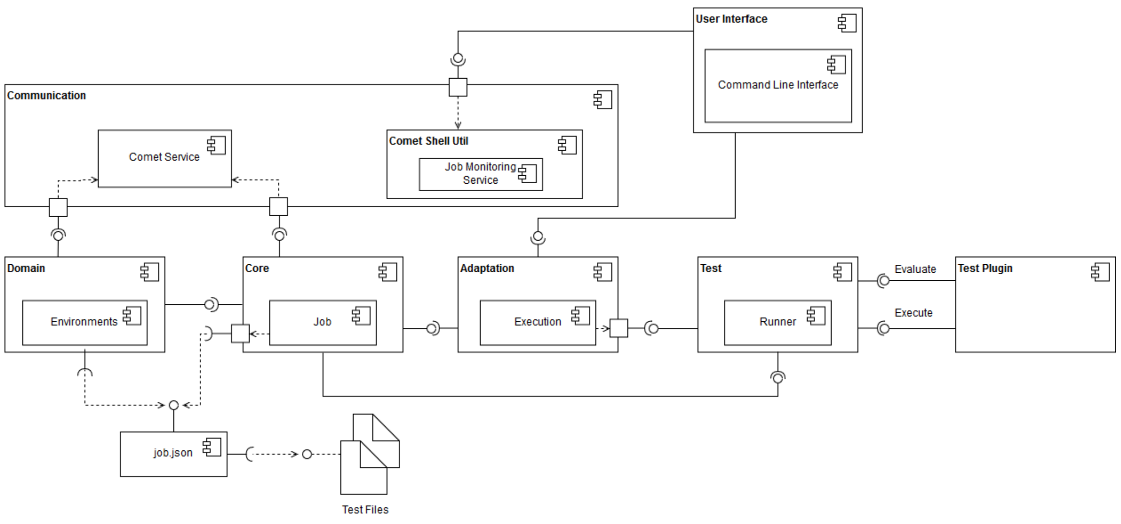


Figure 6.2: The Software Architecture of COMET-P

REST APIs, so the new feature can work. We are going to start with the modifications of the Back-End Spring project.

The first thing was, of course, to make sure that a *Customer Project* can have more than 1 *Environments*. As we've mentioned in section 5.2.1, this was already implemented, as shown in listing 6.6

```java
public class CustomerProject implements Serializable,
    ArtifactEntity, PlaceholderValueDefiningEntity {

  // ...

 @ManyToMany
 @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
 @JoinTable(name = "environment_projects",
                joinColumns =
                   @JoinColumn(name="projects_id",
                               referencedColumnName="id"),
                inverseJoinColumns =
                   @JoinColumn(name="environments_id",
                               referencedColumnName="id"))
  private Set<Environment> environments = new HashSet<>();

  // ...
}
```

Source Code 6.6: CustomerProject.java

The second thing was to change the relationship between *Job* and *Job Result* from One-To-One to One-To-Many. We omit the details about this procedure, as it was accomplished with a simple change in the *JPA* Annotation.

The third and final conceptional change was to simply create reports for each *Job Result* instead of creating only 1 report. We achieved this by simply iterating over all *Job Results* and creating a report for each one. One thing worth pointing about this change is that we didn't have to modify the database schema, because to this stage, reports were generated on every request and weren't saved to the database.

After we were done with the modifications to our domain model and database schema, we had to adapt the two REST Endpoints that return the results for the *Job* and the report. Both of them now return a JSON list with *Job Result* objects and *Report* objects respectively, instead of a single object for each one. This was quite straight forward to implement, but caused some incompatibility with the Front-End, which brings us to the changes that we had to make there.

There were 2 main things that we had to modify in the Front-End. The first change is under the *Compliance Run* page and specifically under the *Finished Jobs* tab. Previously, the user had the option to click on each finished *Job* and see a short summary for the *Job*, including an information if the *Job Results* are valid and if the testing succeeded. Since now we can have multiple results for a single *Job*, we had to remove this summary and only display the small icon that shows if the testing was successful. We removed the additional information, because in the current form of this page, there is no adequate way to display more than 1 summary. An entire page redesign was required, which would have created a lot of overhead and at the end wouldn't really bring a lot of value, because not much additional information will be provided to the user.

The second change is in the reporting page. There we added the option to display the reports for the different *Environments* on which the *Job* ran. Of course, this is only for the cases in which the *Job* ran on more than 1 machine. As we've already explained in section 5.2.1, a combined report for all nodes doesn't make sense, so we didn't implement one.

### 6.2.2 Distribution of Software Landscapes and Software Components to different nodes

The starting point for the implementation of the solution for this problem was, of course, to create our two new entities: *Mapping* and *Mapping Entry*. They currently look like this:

```java
public class Mapping implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToMany(cascade = CascadeType.ALL, mappedBy="mapping", fetch
     = FetchType.EAGER)
  private List<MappingEntry> mappingEntries = new ArrayList<>();

  // Getters, Setters, equals, hashCode and toString methods are
     omitted here

}
```

Source Code 6.7: Mapping.java

```java
public class MappingEntry implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @JsonIgnore
  @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
  @JoinColumn(name = "mapping_id")
  private Mapping mapping;

  @OneToOne(cascade = {CascadeType.ALL}, fetch = FetchType.EAGER)
  @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
  private Environment environment;

  @ManyToMany(fetch = FetchType.EAGER)
  @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
  @JoinTable(name = "mapping_entry_software_landscapes",
    joinColumns = @JoinColumn(name="mapping_entry_id",
       referencedColumnName="id"),
    inverseJoinColumns = @JoinColumn(name="software_landscape_id",
       referencedColumnName="id"))
  private Set<SoftwareLandscape> softwareLandscapes = new
     HashSet<>();

  @ManyToMany(fetch = FetchType.EAGER)
  @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
  @JoinTable(name = "mapping_entry_software_components",
    joinColumns = @JoinColumn(name="mapping_entry_id",
       referencedColumnName="id"),
    inverseJoinColumns = @JoinColumn(name="software_component_id",
       referencedColumnName="id"))
  private Set<SoftwareComponent> softwareComponents = new
     HashSet<>();

   // Getters, Setters, equals, hashCode and toString methods are
      omitted here
}
```

Source Code 6.8: MappingEntry.java

As one can see, our *Mapping Entry* entity has a Many-To-One relationship with the *Mapping* entity. In addition to that, it has a One-To-One relationship with the *Environment* entity, because of our constraint to have a *Mapping Entry* for each *Environment* for which we want to have a “reduced” *Customer Project*. An important thing that we want to mention here and that might be the reason for some questions, is why we allow for a *Mapping Entry* to have only *Software Landscapes* and *Software Components* and not *Compliance Rules*, *Compliance Rule Sets* and *Compliance Profiles*. The reason for that is simple: We assume that all of the compliance rules, rule sets and profiles that are included by the user in the *Customer Project*, should be valid for all *Environments*. That means, that in addition to tests for the selected via the mapping *Software Landscapes* and *Software Components*, the tests for the compliance rules, rule sets and profiles will also be executed. For now we are not planning to change this and give the user the option to also filter the compliance rules, rule sets and profiles, but this feature can be easily implemented in the future with only a few modifications.

After modeling our two new entities, we had to add the One-To-One relationship to the *Customer Project* entity. Finally, we had to figure out how to implement the constraint for a *Mapping Entry* to contain only the *Software Landscapes* and *Softare Components* that were defined in the *Customer Project* and to not contain any others. That isn’t an issue, if one uses the *COMET* Front-End to define the *Mapping Entry*, but since we have a RESTful Back-End, we had to implement this constraint in the Back-End too. We had basically two options to do that: a simple and a complicated one. The compliacted one was to add additional relationships and constraints to our entities, but this seemed like too much work for something that we could achieve much easier. The simple option was to simply iterate over all the *Software Landscapes* and *Software Components* defined in the *Mapping Entries* and to see if they are also defined in the *Customer Project*. This action has to be perfomed, of course, on every definiton or edit of a *Customer Project*, i.e. after every REST request to create or edit a *Customer Project*. One can see how we’ve implemented this in the following code snippet:

```
public boolean
    mappingContainsCorrectLandscapesAndComponents(CustomerProject
    project) {

   if(project.getMapping() == null ||
       project.getMapping().getMappingEntries().isEmpty()) {
     return true;
   }

   for(MappingEntry me :
       project.getMapping().getMappingEntries()) {

     if((project.getLandscapes()
     .containsAll(me.getSoftwareLandscapes()) == false) ||

     (project.getComponents()
     .containsAll(me.getSoftwareComponents()) == false)) {
        return false;
      }
    }

    return true;
  }
}
```

Source Code 6.9: Constraint for a Mapping Entry to only contain Software Landscapes and Software Components defined in the Customer Project

These are all of the changes that he had to make in our model in order for it to support this new feature. The next thing that we had to consider is how to make use of this new data. After some considerations, we decided that in order to use the information from the mapping, we had to create derived *Customer Projects* from our main one with the help of the data from the mapping. The reason for that might not be very obvious, but comes down to one thing: *InSpec*, or more specifically the *InSpec Profiles*. As we've already explained, after we start a *Job*, i.e. run the *Customer Project*, *COMET* generates the test files that are needed by *InSpec* to perform the testing and structures them in an *InSpec*-defined way called an *InSpec Profile*. What all of this means is that if we want to have multiple variations of our *Customer Project* with each one being derived with the help of the mapping, we also need to generate multiple *InSpec Profiles*. And here multiple is actually equal to the number of variations of the *Customer Project* or more precisely 1 *InSpec Profile* for every *MappingEntry* plus one *InSpec Profile* for the *Environments* for which we don't have a *MappingEntry*, if there are any, of course. We find that to be a bit of a limitation of the *InSpec* framework, because one cannot just "tell" *InSpec* which parts of the profile to consider while the *exec* command is executed and which not. Nevertheless, we had to find a workaround for this problem. The solution was two create two new test Task Executors: *ComplianceExecutionComplex* and *ComplianceExportComplex*, both of which extend the abstract task runner *ComplianceRun*. As with *COMET-P*, we needed a new test execution flow. In this work, we are going to explain and present only how the *ComplianceExecutionComplex* test runner works, because the one for the exporting is very similar. The interesting part with the exported *Jobs* is actually when the test results are imported back and that's why are going to focus on that instead.

But before we start with the *ComplianceExecutionComplex* test runner, we had to find a way to use our data from the mapping. The solution that we opted for is to create a new *Mapping Utility* that uses the data from the mapping in order to create "derived" *Customer Projects* from the original one and then uses this *Customer Projects* to create *Jobs* for these *Customer Projects*. Currently our *Mapping Utility* looks like this:

```
public class MappingUtility {

  public List<CustomerProject>
      deriveCustomerProjects(CustomerProject project,
      Set<Environment> mappedEnvironments) {

```

```
    List<CustomerProject> derivedProjects = new ArrayList<>();
    ParentAwareVisitor traverser = new ParentAwareVisitor();

    for(MappingEntry me : project.getMapping().getMappingEntries()) {

      CustomerProject derivedProject = new CustomerProject();
      derivedProject.setId(project.getId());
      Set<Environment> tempEnvSet = new HashSet<>();
      tempEnvSet.add(me.getEnvironment());
      derivedProject.setEnvironments(tempEnvSet);
      me.getSoftwareLandscapes()
      .forEach(derivedProject::addLandscapes);
      me.getSoftwareComponents()
      .forEach(derivedProject::addComponents);

      // Set the Compliance Profiles, Rule Sets and Rules to be the same for every derived project
      derivedProject.setProfiles(project.getProfiles());
      derivedProject.setRuleSets(project.getRuleSets());
      derivedProject.setRules(project.getRules());

      derivedProject.setPlaceholders(project.getPlaceholders());
      derivedProject.setPlaceholders(project.getPlaceholders());
      derivedProject.setTitle(project.getTitle() + " derived");
      derivedProject.setDescription(project.getDescription());
      derivedProject.setArtifactId(project.getArtifactId());
      derivedProject.setSignificance(project.getSignificance());
      derivedProject.setVersion(project.getVersion());
      derivedProject.setMapping(null);

      derivedProject.accept(traverser);

      mappedEnvironments.add(me.getEnvironment());

      derivedProjects.add(derivedProject);

    }

    return derivedProjects;
  }

  private Job createSubJob(Job job, CustomerProject artifact, long derivationNumber) {

    Job derivedJob = new Job();
    derivedJob.setArtifactType(job.getArtifactType());
    derivedJob.setArtifact(artifact);
    derivedJob.setState(Job.JobState.QUEUED);
    derivedJob.setType(job.getType());
    derivedJob.setTitle(job.getTitle() + " derivation " +
        derivationNumber);

    return derivedJob;
  }

  public List<Job> createSubJobs(Job job, List<CustomerProject>
      derivedCustomerProjects) {

    List<Job> subJobs = new ArrayList<>();

    long derivedJobsCount = 1;

    for(CustomerProject cp : derivedCustomerProjects) {

      // Since we are not going to persist these Jobs, we only
          need an ID to differentiate them
      Job current = this.createSubJob(job, cp, derivedJobsCount);
      current.setId(derivedJobsCount);
      subJobs.add(current);
      derivedJobsCount++;
    }

    return subJobs;
  }
}
```

Source Code 6.10: MappingUtility.java

As one can see, we have three new methods. The first one, *deriveCustomerProjects* takes our original *Customer Projects* and creates as many derivations of it as the number of mapping entries. Each derivation is almost identical to the original *Customer Project* with the exception of the *Software Landscapes* and *Software Components* that it has. These are, of course, specified in the mapping for each derived project. Probably an interesting point here is the second parameter that this method takes, the *mappedEnvironments* set. We use this set to keep track of the *Environments* for which the user defined a mapping. After we finish the derivation process, we remove the *Environments* for which a mapping exists from the set of *Environments* defined for the original *Customer Project* and if there are any *Environments* left for which a mapping doesn't exist, we also add the original *Customer Projects* to the list of *Customer Projects* for which a *Job* has to be created. We are performing this action in the *ComplianceExecutionComplex* test runner, as we shall later demonstrate.

The second method in our utility, *createSubJob* is used to create a so called "sub" *Jobs* with the help of the main *Job* and a derived *Customer Project*. As with a regular *ComplianceExecution*, we treat the entire testing process as one *Job*. This means that even for complex executions, we have one master *Job* in which the entire testing is performed. However, since 1 *Job* means 1 execution of the connectivity test, the *InSpec check* command and the *InSpec exec* command, we had to create “sub” *Jobs* if we wanted to keep the same execution principle. So, naturally, this is what we did and that’s why we have this method.

The third method is actually the one that is used by our test runner and basically utilizes the *createSubJob* method. It delivers a list of *Jobs* which is created from the main *Job* with the help of the derived *Customer Projects*.

Now we can move on to our new *ComplianceExecutionComplex* test runner. We won’t present it’s entire source code here because of space reasons and we encourage the reader to take a look at it himself/herself. We will only present some parts of the *run* method that we find to be important.

After the *Customer Projects* and the *Jobs* are derived, the execution process is started as normal for each *sub Job* with the help of the *InSpec TestRunner*. Another interesting point here is that we use a *HashMap* to keep track which *Job Result* belong to which *Environment*. At the end of the testing process, these *Job Results* are sorted according to the order of the *Environments* in the original *Customer Project*. Part of this process is presented in the following listing:

```java
@Override
public void run() {

    // The master Job is created as usual and and the process is omitted here

    // Derivation of Customer Projects and Jobs is also skipped, as it is already explained

    Map<Long, JobResult> environmentResultSequence = new HashMap<>();
    Map<Long, Report> environmentReportSequence = new HashMap<>();

    for(Job currentJob : subJobs) {

        // Performing the testing via the InspecTestRunner
        for(Environment currentEnv : currentProject.getEnvironments()) {

                JobResult currentResult = runner.getResults().get(currentJobResult);
                Report currentReport = this.reportUtility
                .createReportForJobResult(this.job.getId(), currentResult, currentProject);

                environmentResultSequence.put(currentEnv.getId(), currentResult);
                environmentReportSequence.put(currentEnv.getId(), currentReport);
```

```

                    currentJobResult++;
                }
        }
}
```

Source Code 6.11: ComplianceExecutionComplex.java

As one can notice, in addition to the *HashMap* for the *Job Results*, we also have a *HashMap* for the *Reports*. One could also see that these reports are generated directly after the testing is completed as opposed to upon a request from the Front-End. The reason for this is that we ran into some obstacles with the old way of report generation. Previously, the reports weren't persisted in the database and they were generated on every request. This wasn't very efficient, but wasn't also an issue. The problem we had with this method for report generation is that the *Customer Project* was need in combination with the *Job Results* in order for the report to be generated. And since we are not persisting our derived *Customer Projects*, we had no choice but to generate the reports directly after the execution of the tests, when we still had the derived *Customer Projects*. Then, we had to persist this report in the database.

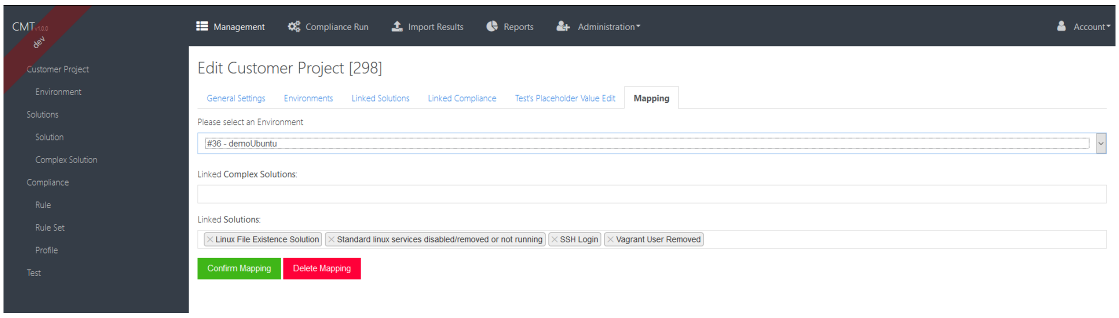
We could have accomplished that in couple of ways, but we decided that the easiest way was to extend our *Job* entity and add a *reports* attribute/column to it. There we could persist the reports directly after they are generated. And since our entire *COMET* Back-End communicates via RESTful APIs with the Front-End, we chose to persist these reports directly as JSON string, so we didn't need to perform any conversions after that. This entire new process of report generation was applied also to all other scenarios of execution, i.e. to the regular *Compliance Execution*, where we don't have a mapping, and also to the *ComplianceExport* and *ComplianceExportComplex*. With the *ComplianceExport* and *ComplianceExportComplex* we, of course, generate the reports when the *Job Results* are imported back by the user via the Front-End or with the help of *COMET-P*. An important thing worth pointing out is that in the case of the *ComplianceExecutionComplex*, we had to derive again the *Customer Projects* and *sub Jobs*, because the information for them is needed for the report generation.

With that, we pretty much explained all the details about how our mapping works in practice and how we've implemented the functionality in our Back-End. The only thing that left unmentioned is how we differentiate between our test runners, *ComplianceExecution* and *ComplianceExecutionComplex*, and how we pick which one to use. This is actually rather simple and we make the choice only based on the fact if the *Customer Project* has a mapping defined by the user.

Next, we want to move to our Front-End and demonstrate how we've integrated the new features in it. This part was far less interesting, because we only had to add the menu for the mapping in the page for editing of *Customer Projects*. We managed to create a very simple menu, the majority of which we actually reused. As one can see in figure 6.3, the mapping menu looks almost identical to the menu for selecting the "to be included" in the *Customer Project*, *Software Landscapes* and *Software Components*. The reason for that is, of course, that this is pretty much what the mapping does.

As for the reports page, we didn't need to add anything more to it, after we've already edited it when we added the support for testing multiple *Environments* at a time. The page currently looks like as visualized in figure 6.4 and figure 6.5.

The final thing that we had to do is to adapt our *COMET-P* tool, so it can also execute *Jobs* that have a mapping. And since our tool basically mimics the functionality of *COMET*, we only had to include our *ComplianceExecutionComplex* test runner in *COMET-P*, adapt a few things and we were ready.



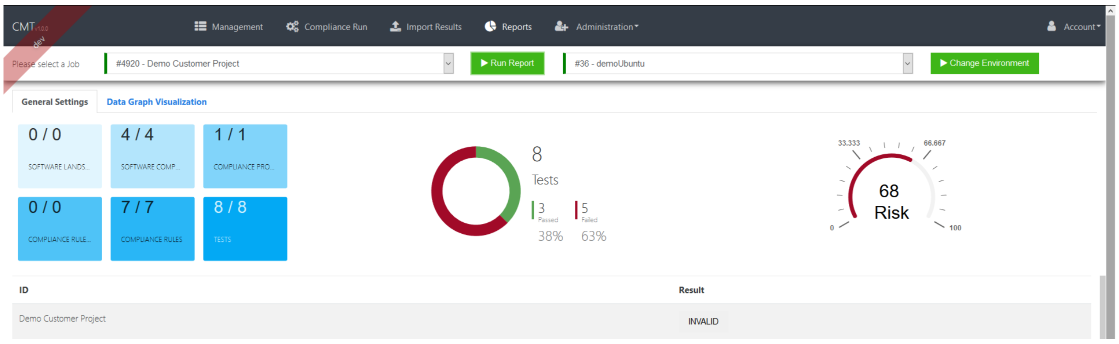
Edit Customer Project [298]

Please select an Environment: #36 - demoUbuntu

Linked Complex Solutions:

Linked Solutions: Linux File Existence Solution | Standard linux services disabled/removed or not running | SSH Login | Vagrant User Removed

Figure 6.3: The new Mapping tab under the Customer Project Edit menu, which allows the user to define a Mapping

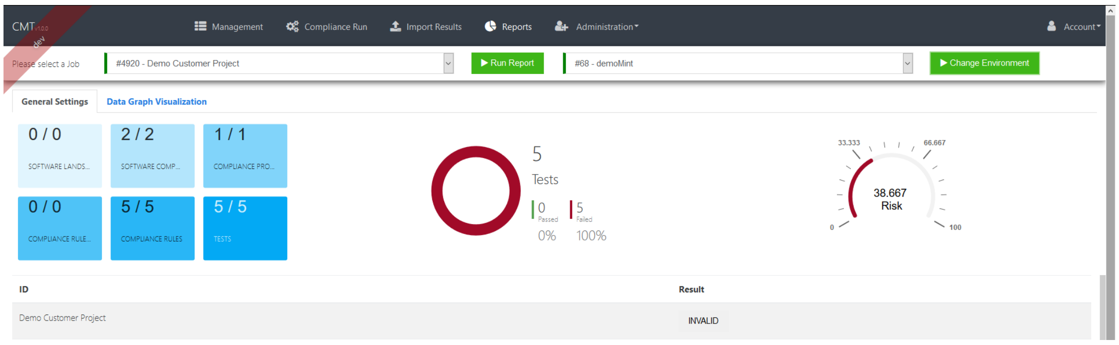


#4920 - Demo Customer Project

#36 - demoUbuntu

| ID | Result |
|---|---|
| Demo Customer Project | INVALID |

Figure 6.4: The modified Reports page that shows the reports for the different Environments



| ID | Result |
|---|---|
| Demo Customer Project | INVALID |

Figure 6.5: The modified Reports page that shows the reports for the different Environments

# 7 Evaluation

It's not a bug - it's an
undocumented feature.

Author Unknown

Contents

To evaluate our contributions, we will take a look at two example scenarios. The first one will be for *COMET-P* and the second for *COMET* and the testing of distributed systems. With the help of these example scenarios, we will try to not only show how we've improved *COMET*, but we will also point out some limitations and how they can be removed.

Before we begin with the evaluation, we want to say something about our testing environment/target SUT. Unfortunately, we don't have access to a proper distributed system and building one isn't exactly a child's play. However, defining the same *Environment* in *COMET* multiple times with a different name effectively allows us to use it a couple of times for the same *Customer Project*. This is a good enough simulation of a distributed system since the idea here is just to have multiple nodes to test.

For our remotely accessible "distributed" system, we are going to use a node with a Linux Ubuntu distribution running inside a virtual machine and another node running Windows 10, again in a virtual machine. For our non-remotely accessible machine, we are going to use a Windows 10 computer, on which we are also going to run *COMET* and *COMET-P*. One might ask, why the virtual machine with Windows 10 is remotely accessible and the Windows 10 computer, on which we are running *COMET* is not? The reason for that is simple: As with SSH, WinRM has to be activated, so one can use it to access a remote Windows machine. By not activating it and by not choosing to overwrite the *Environment* with the local one, when starting the *Job* in *COMET*, we can effectively simulate a machine that is not remotely accessible and we can use *COMET-P* to test it for compliance. We will activate WinRM however for the Windows 10 system running on the virtual machine.

As far as the actual compliance tests goes, we will use the same tests that were used in the evaluation part in [Mos17]. The idea here is, again, not to perform actual compliance testing, but to only demonstrate how our contributions improved *COMET*.

## 7.1 COMET-P

The first test scenario is, as explained above, to run *COMET-P* on a machine that is not remotely accessible and to perform the compliance testing with its help. To achieve this, we are running *COMET* and *COMET-P* on our Windows 10 machine and we are not enabling WinRM.

The *Customer Project* that we defined for this *Job* was very minimalistic and contained 1 *Software Component* with 1 *Compliance Rule* and 1 *Compliance Test*, which basically tested if a file exists on the system. For this scenario, we didn't really need anything more complicated. The *Job* was exported by us as usual in the *COMET* Front-End with the method that we implemented. Then, the *Job* was downloaded in *COMET-P* and executed with the following commands:

```
// Download the Job with ID 5049
download 5049

// Run Job 5049, import back the test results on completion,
// overwrite the Environment with the local one
run 5049 -i -l
```

Source Code 7.1: Downloading and Running a Job with COMET-P

As one can see, we are overwriting the *Environment* with the one on which *COMET-P* is running, i.e. our local Windows 10 machine. We are also choosing to import back the *Job Results* to *COMET*, when the testing is complete. One can also do this manually via the *Results Import Page* at a later point, if *COMET* is not reachable by *COMET-P*. In the event that *COMET* is not reachable, one can manually transfer the *Job* files too, so the testing with *COMET-P* is alway possible and has no limitations due to the lack of connection. Such a testing wouldn't be possible without *COMET-P* on a machine that is not remotely accessible. Of course, one could deploy *COMET* on such machines, but *COMET* is a bigger application, which has more dependencies, requires a database and is overall slower. These are problems, which we managed to all solve with the help of *COMET-P*.

One small limitation that are new tool has, is if we have multiple *Environments* for the *Job* and we want to overwrite one of them. We currently don't allow that and it is debatable if we should implement such an option, because the idea when testing multiple machines at a time, is to test them remotely and not to run *COMET-P* on one of them. The overwriting of *Environments* is however not the biggest current issue.

In addition to this small limitation, we also have a problem, which we unfortunately didn't manage to solve with *COMET-P* due to the limited time frame in which we had to complete our thesis. This problem has to do with the testing of distributed systems that are not remotely reachable at all. We still don't have a solution for this scenario. Currently, we cannot deploy *COMET-P* on each node of the distributed system, test it independently from the others and then import back the results. The implementation of this feature is not very straight forward and might become a bit complicated. One has to decide how to split the test files or to tell *COMET-P* which one to use. Ideally, *COMET-P* has to know on which *Environment* it is currently running and this should be also reflected in *COMET*. In this scenario the *COMET-P* instances have to be identifiable, i.e. we will need some kind of ID for them. Overall, supporting such a testing will create a lot of communication overhead between *COMET* and *COMET-P*, but it has to be implemented at some point if the future, because it will further expand the testing “skills” of *COMET*. That’s why we are going to include the problem in the chapter about Future Work.

## 7.2 Compliance Testing of Complex and Distributed Systems

For the second part of our evaluation, we have the following scenario: We will run *COMET* on our local Windows 10 machine and then have 2 virtual machines, which will simulate our distributed system. The first virtual machine will be running a Linux Ubuntu distribution and the second one a copy of Windows 10. Both machines will have enabled remote connection. For the Linux one we will have SSH as the method for remote connection and for the Windows machine we will have WinRM enabled.

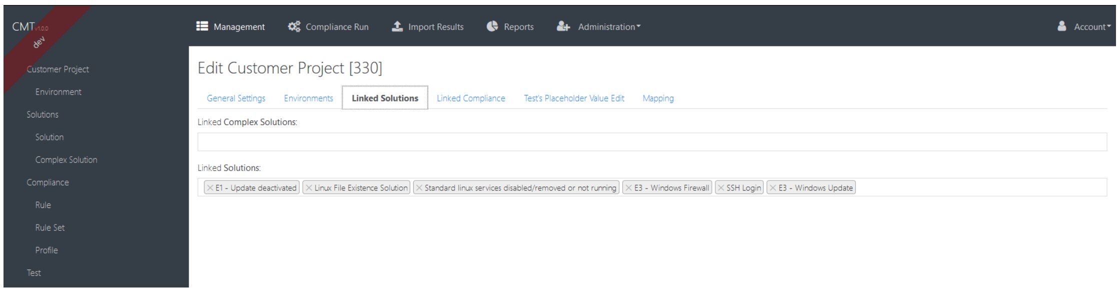
Here we will skip the case in which all of the *Environments* have to have the exact same configuration and have to be tested with the exact same compliance tests, i.e. the case in which we don’t need a mapping. This case is a simplified version of the testing that we are going to perform next and is nothing special, because we basically run the tests for all *Environments* in the *Customer Project*.

Speaking of the *Customer Project*, here we don’t need anything special again, as for the first scenario. We just need a *Customer Project* which has multiple *Software Landscapes* or/and multiple *Software Components*, so we can distribute these between the different *Environments*. In figure 7.1 one can see how we’ve defined our *Customer Project*. Then, in figure 7.2 and in figure 7.3 one can see how we’ve defined our *Mapping*. For this *Customer Project* we chose not to include any *Compliance Rules*, *Compliance Rule Sets* or *Compliance Profiles*, but if we had included some, the tests for them would run on all nodes.

We are not going to include a picture of how the results and the reports for this *Job* look, because they are pretty similar to the ones we showed in section 6.2.2. What we can definitely say however, even without showing the results for this concrete *Job*, is that with the help of the mapping that we created, we are now able to test distributed systems for compliance. This is major improvement, because now one can have a complex distributed system as he/she wants to, and test it for compliance with *COMET* with the help of only 1 *Customer Project* and only 1 *Job*.

The only real limitations, if one can call them that, is that all *Compliance Rule, Rule Sets* and *Profiles* will be executed on all nodes and that the Placeholders have to have the same values for all nodes. Our *Mapping* currently doesn’t support *Compliance Rules, Rule Sets* and *Profiles* and also *Placeholders*. One can argue however, that this is the correct behaviour, because the idea for these rules is for them to be valid for the entire system, i.e. the entire *Customer Project*. There are some cases though, in which it would be good for the *Mapping* to support *Compliance Rules, Rule Sets* and *Profiles.* For example, if we take a look at the *Linux Baseline Profile*, which we already mentioned and which is defined as a *Compliance Rule* and we want to add it to our *Customer Project*, we would get bad test results. The reason for that is, of course, because we also have a Windows *Environment* in our *Customer Project*, for which the *Linux Baseline* tests will fail. This means, that if we really want to include these tests to our *Customer Project*, we would have to define them either as a *Software Landscape*, or as a *Software Component* and then use this *Landscape/Component* in the mapping. This however, wouldn't be the correct way to define these baseline tests, so one could say that our *Mapping* can't be used in all cases. But because one could argue, if the *Mapping* really should support *Rules, Rule Sets* and *Profiles*, we leave this option out. Everything can be added to the *Mapping* with only a simple extension of the model and a few tweaks though, so we don't consider this to be a major issue.

The other thing that one can argue about is, if it should be possible to substitute the *Placeholders* for each *Mapping Entry*, i.e. for each *Environment*. This is again debatable and there are pros and cons for the support of this feature. On one hand, the *Placeholder Values* should be the same for the entire system, and on the other the option to define them for every *Environment* would give us more flexibility. Either way, we chose not to support this feature for now, but it can also be added very easily with only a few modifications.

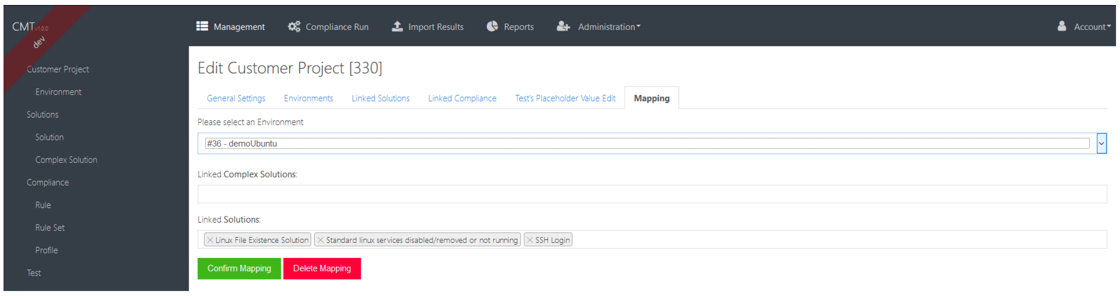


Edit Customer Project [330]

Linked Complex Solutions:

Linked Solutions: E1 - Update deactivated | Linux File Existence Solution | Standard linux services disabled/removed or not running | E3 - Windows Firewall | SSH Login | E3 - Windows Update

Figure 7.1: The Customer Project used to evaluate the testing of Distributed Systems



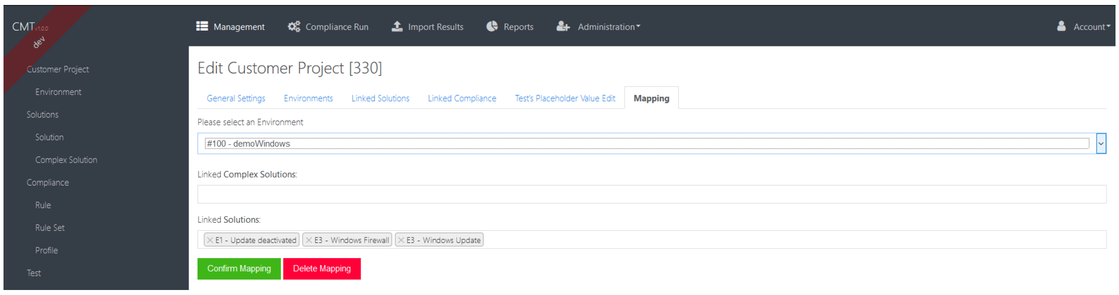
Edit Customer Project [330]

Please select an Environment: #36 - demoUbuntu

Linked Complex Solutions:

Linked Solutions: Linux File Existence Solution | Standard linux services disabled/removed or not running | SSH Login

Figure 7.2: The Mapping for our Linux Ubuntu Environment used to evaluate the testing of Distributed Systems



Edit Customer Project [330]

Please select an Environment: #100 - demoWindows

Linked Complex Solutions:

Linked Solutions: E1 - Update deactivated, E3 - Windows Firewall, E3 - Windows Update

Figure 7.3: The Mapping for our Windows 10 Environment used to evaluate the testing of Distributed Systems

# 8 Conclusion

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger Dijkstra

Contents

In this thesis we managed to make 2 big contributions to *COMET* and with that to also solve 2 of the bigger problems that it had. First, we developed our *COMET-P* tool, which can work independently from *COMET* and can test machines that are not remotely accessible. Our tool provides a solid base for further development and with a little extra work, one would be able to execute even more complex testing procedures. But the bigger contribution that we made to *COMET*, is the extension with the help of which *COMET* can now test multiple machines at a time and also, and more importantly, distributed systems.

We now want to summarize what we've accomplished in our work and present a few topics for future work.

## 8.1 Summary

In chapter 1 we presented briefly the problems on which we worked in this thesis and outlined how our work is structured.

Then, in chapter 2 we made an introduction to *COMET* and explained how it works. This was very important for the next chapters, because our work is entirely based on *COMET*.

In chapter 3 we presented some of the problems and limitations that *COMET* had, then described in more detail the 2 problems that we worked on in this thesis, and explained why we picked them.

After that, in chapter 4 we wrote about related work in the field of *Compliance Testing* and *Compliance Testing of Distributed Systems*.

Our real work however, started in chapter 5 where we presented the concept for the solutions of our two problems. First, in section 5.1, we explained how we are going to solve our first problem and make *COMET* portable. Then, in section 5.2 we presented how we plan to make the testing of distributed systems with *COMET* possible and with that solve our second problem.

After that, in chapter 6 we implemented our solutions and explained how we did it. We followed the same structure as for chapter 5 and presented the details for the solution of the first problem in section 6.1 and for the second in section 6.2.

Finally, we evaluated our solutions in chapter 7. In addition to showing how they solved the 2 problems, we pointed out some limitations that they have and how they can be removed in the future.

## 8.2 Future Work

As far as future work goes, there is still a lot to be desired from, improved and added to *COMET* and *COMET-P*. Here we present some open problems and topics. The list is not by any means complete and can be extended with some of the open problems listed in [Mos17].

### 8.2.1 Independent testing of the nodes in a distributed system

Currently, both *COMET* and *COMET-P* can only perform the testing, if all of the nodes are accessible at the time of starting the *Job*. It is not possible to test only a few of the nodes and then, at a later point, to test the rest of them and to add the results. That means, that for the scenario in a distributed system in which all of the nodes are not accessible via remote connection, one cannot create one *Customer Project* for the entire system and then execute parts of it independently on the different nodes. This is not a major issue, but it is a limitation, which has to be removed.

### 8.2.2 Monitoring and Control of COMET-Portable instances from COMET

Another nice to have feature, which will be particularly useful when the problem described above is solved, will be to extend *COMET-P* so, that its configurations and settings can be changed remotely via REST requests. Also, monitoring and controlling all of the *COMET-P* instances, which are connected to *COMET* will be very helpful. Remotely starting a test, scheduling a tests and so on, are just some of the things that can be achieved here.

### 8.2.3 Improvement of the Reporting and Report Visualization

Although *COMET* provides some report tooling, it is by far now ideal. The violation score is not perfect and also the *Data Graph Visualisation Tree* in the *Reports* page can be much better. The tree currently doesn't reflect which parts of the *Customer Project* are mapped to which *Environment/Node* and it is not visually pleasing. In addition to that, more detailed reports in form of PDF files are also a feature that can be added to the report tooling.

### 8.2.4 General Improvements and Optimizations for the COMET Front-End and Back-End

*COMET* was created more or less with the *Proof of Concept* idea in mind and we sort of continued that trend in this thesis. At some point however, the entire system has to be optimized and some things have to be fixed, so everything is production ready. A big performance improvement that can be made is the asynchronous execution of tests for *Jobs* with multiple nodes. Currently, the testing is performed sequentially for each node. As a result, the entire testing procedure takes a lot more time and that can turn into an issue for systems with a lot of nodes.

# Bibliography

[Ang] *Angular*. URL: https://angular.io/ (visited on 02/22/2018) (cited on page 3).

[Apa] *Apache JMeter*. URL: http://jmeter.apache.org/ (visited on 02/22/2018) (cited on page 7).

[Che] *Chef Automate*. URL: https://www.chef.io/automate/ (visited on 04/18/2018) (cited on page 10).

[Fas] *FasterXML/Jackson*. URL: https://github.com/FasterXML/jackson (visited on 03/21/2018) (cited on page 27).

[Fun] *Functional Testing*. URL: https://en.wikipedia.org/wiki/Functional_testing (visited on 02/07/2018) (cited on page i).

[Gat] *Gatling*. URL: https://gatling.io/ (visited on 02/22/2018) (cited on page 7).

[HW03] G. Hohpe and B. Wolf. *Enterprise Integration Patterns*. The Addison-Wesley Signature Series. 2003, pp. 1–574 (cited on page 9).

[Ins] *Inspec*. URL: https://www.inspec.io/ (visited on 02/22/2018) (cited on page 3).

[Jhi] *JHIPSTER*. URL: http://www.jhipster.tech/ (visited on 02/22/2018) (cited on page 3).

[Mic] *Microservices*. URL: https://en.wikipedia.org/wiki/Microservices (visited on 02/22/2018) (cited on page 8).

[Mos17] M. Moscher. "Continuous Compliance Testing". In: (2017), p. 147 (cited on pages i, 1–3, 7, 9, 10, 27, 43, 50).

[Mys] *MySQL*. URL: https://www.mysql.com/ (visited on 02/22/2018) (cited on page 3).

[Non] *Non-Functional Testing*. URL: https://en.wikipedia.org/wiki/Non-functional_testing (visited on 02/07/2018) (cited on page i).

[Ope] *OpenVAS*. URL: http://www.openvas.org/ (visited on 02/22/2018) (cited on page 7).

[Sen] *Sentinel*. URL: https://www.hashicorp.com/sentinel (visited on 04/18/2018) (cited on page 10).

[Spra] *Spring Boot*. URL: https://projects.spring.io/spring-boot/ (visited on 02/22/2018) (cited on page 3).

[Sprb] *Spring Shell*. URL: `https://projects.spring.io/spring-shell/` (visited on 03/23/2018) (cited on page 28).

[Upg] *UpGuard Core*. URL: `https://www.upguard.com/product/core` (visited on 04/18/2018) (cited on page 10).