The present work was submitted to
the Research Group
Software Construction

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

Master Thesis

Adapting Regression Test Optimization for Continuous Delivery

presented by

Titiruck Nuntapramote

Aachen, October 24, 2018

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens

Christian Plewnia, M.Sc.

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 Master's thesis entitled

Adapting Regression Test Optimization for Continuous Delivery

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 Masterarbeit mit dem Titel

Adapting Regression Test Optimization for Continuous Delivery

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, October 24, 2018 (Titiruck Nuntapramote)

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, October 24, 2018 (Titiruck Nuntapramote)

Acknowledgment

First of all, I would like to thank Prof. Dr. rer. nat. Horst Lichter and the research group SWC for providing me with the opportunity to write my master thesis and Prof. Dr. rer. nat. Bernard Rumpe for reviewing this thesis.

Special thanks to Andreas Steffens and Christian Plewnia for guiding me through this thesis in the last six months, Marwa Maghnie for proofreading my terrible writing, and lastly my friends and family for the continuous and unconditional support.

In retrospect, the last few months have been quite challenging and this thesis would not be possible without any of you.

Titiruck Nuntapramote

Abstract

Continuous delivery is nowadays a popular software engineering approach that is adopted by many organizations. Due to the nature of continuous delivery where software is produced in short cycles in an automated manner, the role of regression testing becomes increasingly crucial in order to ensure that the modifications to any existing software functionalities do not introduce new bugs. As most softwares grow larger and more complex over time, the size of the test suite consequently becomes larger as well. Hence, regression testing turns into a rather expensive maintenance activity and demands the execution to be more efficient. This leads to the need for an automated regression test optimization. Regression test optimization uses techniques, such as test suite prioritization and selection, together with the optimize data required to utilize the techniques to acquire which tests and in what order need to be executed in a particular iteration. As a result, a test suite that is usually smaller in size and optimally ordered is executed, effectively saving the limited resources such as time.

To adapt the regression test optimization for continuous delivery, it must be automated as part of the delivery process. Thus, this thesis provides an integration of the regression test optimization into the delivery system such that it ensures an easy implementation of new optimization techniques and at the same time, can be flexibly used in the delivery process. This thesis also solves the challenge regarding the architectural migration to refactor the monolithic regression test optimization system within the microservice architecture of the delivery system.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

Contents

Maintenance of a software is inarguably an important part of the software life cycle. A study shows that maintenance accounts to more of the overall cost of software production than the development of software itself [Pre10]. Regression testing is a major maintenance activity. It is performed whenever modifications, whether to fix defects or enhance existing functionalities, are made to the existing software in order to validate that they do not introduce new bugs or create undesirable side effects. During the course of software development, regression testing usually begins when a modification of the code occurs [LW89]. In the most naive, straightforward approach, regression testing can be executed as a whole i.e. all the tests are rerun. However, in reality, a software project is limited by resources such as time and this consequently renders it impractical to re-execute every test in some case [Pre10].

One of the recent software development trends is Continuous Delivery (CD). It adopts a shorter software life cycle approach and as a result introduces further restrictions and constraints to the execution of regression testing [YH07]. CD aims to automate the software development process i.e. build, test, deploy, and release the product in short and frequent iterations [HF10]. As a testament to its frequency, Martin Fowler refers to it as "*the world of Continuous Delivery, where releasing becomes routine*" [HF10]. Due to these constraints coupled with the fact that most softwares become more complex overtime, the demand arises for an even more efficient automation of regression testing.

Because of the impracticality of re-executing the entire test suite mentioned previously, in an ideal scenario, we want to only rerun tests related to the change in the most optimal order to save resources. This brings us to the topic of Regression Test Optimization (RTO). RTO is an ongoing area of research and a number of different approaches have been studied. However, most studies published in this area focus either on a single strategy or similar family of strategies with different input parameters and do not attempt to provide a common tool or platform that can process different strategies requiring different types of optimization data. Moreover, when RTO is used, it is rarely integrated into the delivery system. This leads us to the question of how can RTO be incorporated into a CD process such that different optimization strategies can be used.

Considering that RTO is a large field of study with various optimization strategies, the aim of this thesis is to integrate RTO and automate it as part of a delivery system

with enough flexibility to handle different strategies and along with it, various types of optimization data. For this we consider, RTO not as a singular activity or entity within the delivery system but rather as many small tasks that build up the RTO process itself. The challenge is such that the architecture and the domain model must be designed to accommodate this issue in a way that RTO can seamlessly be automated, which is at the heart of CD philosophy.

1.1 Thesis Contributions

The goal of this thesis is to adapt RTO for a delivery system which paves the way for an automated RTO process. The contributions of this thesis are outlined as follows:

1. Different RTO approaches are examined. A brief introduction of RTO is provided and the approaches are summarized and categorized.

2. The concept for an adaptation of RTO for a delivery system is presented, migrating from a monolith to microservices. The objectives and requirements are specified. The domain model of RTO is presented along with the implementation of the functional adaptation.

3. The thesis also provides the concept for an extension of the delivery system architecture where data and artifacts used in different RTO strategies can be collected and reused during the course of the entire delivery model to support the RTO process. The extended architecture is outlined and the implementation is provided. A test project with a complete delivery pipeline including RTO is used for evaluation.

4. The thesis concludes with recommendations of possible improvements and future work on the topic.

1.2 Structure

Firstly, chapter 2 gives an introduction about regression testing and RTO as it is the central concept of the thesis and provides a summary on RTO strategies and their categorizations. It also introduces the existing works, Lazzer and Jarvis, that are the basis of the works done in this thesis. Chapter 3 identifies the challanges, the scope of the thesis, and the requirements. Then chapter 4 investigates the existing systems and concepts related to the scope of this thesis. Next the domain model and architecture are outlined in chapter 5. Chapter 6 presents the iterative developments of the corresponding functional adaptation of RTO. In chapter 7, the implementation is evaluated against using a test project. Chapter 8 then discusses possible improvements and future works and concludes the thesis.

2 Background

Contents

This chapter introduces the central concept that drives the scope of this thesis, namely Regression Test Optimization(RTO). It provides a comprehensive background, necessary to understand RTO. Section 2.1 gives an overview of regression testing: its definition, objectives, and general workflow. Next section 2.2 elaborates on the idea behind RTO and its challenges. This section also provides a summary from available literature reviews and surveys on classification and categorization of different RTO techniques including Regression Test Prioritization (RTP) and Regression Test Selection (RTS).

Another important concept for this thesis is Continuous Delivery (CD). Section 2.4 provides a background and an overview of CD.

At the end of the chapter, we introduce the previous works which contribute greatly to the work of this thesis: Lazzer, a functional protype of an RTO framework and Jarvis, a self-organizing continuous delivery system.

2.1 Regression Testing

Software development is a continuous process and usually does not end at the first release of the product. This makes testing a very crucial component of the software development process since it is needed to validate the correct behavior of the software. As a software

changes during the course of its development, be it from an error correction, a refactoring, an improvement to an existing functionality or an implementation of a new feature, we must make sure that we do not introduce new bugs, or in a more technical term *Regressions*. This type of testing is referred to as *Regression testing*.

Regression testing is a process of re-testing a software to ensure that it does not behave undesirably after it has been modified. It is usually an activity done before every software release. The goal of regression testing is to determine whether the modification has regressed the program [Mye+11] and if so, to discover regressions and fix them as early as possible because the cost of damage is much higher if they are found once the software is released and in use.

Regression testing makes up a majority of testing efforts and resources in commercial software development [AO08]. Because small changes in one part of a system can cause problems in a distant, non-obvious part, it is important that the regression test suites are carefully designed to cover most parts of the code as well as the interactions between each part. As a result, a large system is inclined to have large regression test suites and as it grows over its development course, its test suits culminate in complexity. In an ideal scenario, regression testing should be run after every change made to the software. However, with large test suites, regression testing cannot be run as often as desired due to the runtime and the cost of its operation. Hence many organizations prefer to have pre-determined periodic runs of regression testing [AO08]. For example, the tests can be scheduled to be executed nightly with all the accumulated modifications made during the day.

Because of the aforementioned repetitive nature of regression testing, an automation of regression testing is desired and especially needed. In fact, a direct quote of from the book *Introduction to Software Testing* by Ammann and Offutt emphasized that "*regression tests must be automated. Indeed, it could be said that unautomated regression testing is equivalent to no regression testing*" [AO08]. According to an article by Onoma et al., the demand for the extensive use of regression testing has led many companies in the industry to develop their own in-house tools to automate the process [Ono+98]. Because the cost for regression testing is high, effective techniques are needed to reduce the number of tests and improve the order of the required test cases to be rerun [Ber07]. There are several techniques to optimize the test suite with the most prominent branches being: Regression Test Minimization (RTM) for minimizing the test suite by permanently reducing the size of the test suite, Regression Test Prioritization (RTP) for prioritizing the test suite and Regression Test Selection (RTS) for selecting a subset of tests from the test suite. Overall, this process is referred to as *Regression Test Optimization* and is discussed in the next section.

2.2 Regression Test Optimization (RTO)

Regression testing can be a very resource consuming activity. In the most straightforward scenario, we can re-execute all tests in every regression test run. Although this is possible, it is often not commercially feasible due to time and budget constraints. In fact, one of

the biggest challenges of regression testing is that over time the regression test suites grow in size and complexity. Old, obsolete tests are rarely discarded [Won+97]. This results in a high cost of resources to maintain and operate the process as well as a long execution time. There are a few approaches that can be used to overcome this problem. The simplest way is to reduce the number of test cases for each run by manually disregarding unnecessary tests. However, the task is tedious and prone to the nature of human error. Moreover, even to an experienced tester, it may not be obvious which tests should be removed on a particular run.

Another approach is to utilize RTO. A survey by Yoo et al. further summarized that there are three major branches of RTO approaches: test minimization, test selection and test prioritization [YH07]. To understand these three classes of techniques, it is important to first understand the classification of test cases. The original version of the program is referred to as P and the modified version as P'. Leung and White suggested grouping test cases into the three following, mutually exclusive classes [LW89]:

- **Reusable (R_t)** - This class includes the test cases that test unmodified parts of the program i.e. the common part of P and P'. Hence, with regards to RTO, they do not need to be rerun because they produce the same results.
- **Retestable (T_t)** - This class includes the test cases that test the modified parts from the program P to P'. These test cases should be rerun as per the goal of RTO to reveal possible regressions.
- **Obsolete (O_t)** - This class includes the test cases that are rendered obsolete i.e. they can no longer be used for testing the program P'. This can be caused by a number of reasons including:
  - The specification has changed and consequently, the input or output is no longer correct.
  - The program has been modified in a way that the tests are no longer valid.
  - The tests are *structural* but no longer contribute to the structural coverage of the program. Structural tests refer to the tests that are used to test the structure of the code, meaning they are more interested in how the program operates rather than the functionalities and results.

Regression Test Minimization (RTM) solves these challenges presented by regression testing by identifying and eliminating the obsolete test cases from the test set of program P'. Regression Test Selection (RTS) works by selecting a subset of test cases that are affected by the modifications of the program, or as defined earlier as retestable test cases. These techniques can be implemented both manually and automatically. In fact, an empirical study of manual and automated RTS by Gligoric et al. showed that developers performed RTS manually in $59.19 \pm 35.16\%$ (mean $\pm$ standard deviation) of the tests they studied [Gli+14]. On the other hand, Regression Test Prioritization (RTP) does not aim to reduce the number of test cases but instead focuses on the order of the tests being executed. In order to establish a clear distinction between these classes, the next subsections provide a brief definition and summary of each of them.

2.2.1 Regression Test Minimization (RTM)

RTM is sometimes referred to as *Test Suite Minimization* or *Test Suite Reduction*. In general, RTM aims to reduce the size of the test suites by eliminating obsolete test cases O_t from the test suites. Unlike other classes of techniques, this elimination can be permanent [YH07]. A formal definition of test suite minimization problem by Rothermel et al. is given as follows [Rot+02]:

Definition 2.2.1 (Test Suite Minimization Problem)
Given: *The test suite T, a set of test requirements R as $r_1, ..., r_n$ and subsets of T, $T_1, ...T_n$, each associated with requirement r_i such that any test cases t_j in T_i can be used to satisfy r_i).*

Problem: *Find a subset T' from T such that all requirements from R are satisfied.*

By this definition, we establish that a subset T' from T satisfies the same requirement set R, meaning that executing a test suite T' on a program P will achieve the same coverage as running T. In order to reach the optimal solution, the subset T' should be minimal. This minimal set problem is NP-complete[YH07].

RTM techniques, unlike any other classes of techniques, are not modification-aware, meaning that they only deal with a single version of the software. Thus, they can be run as a standalone operation to the targeted software. Furthermore, because of their permanent reduction of a test suite, they should not be used as part of a delivery pipeline due to the direct modification of the test suite. For these reasons, we disregard RTM techniques in the rest of the thesis.

2.2.2 Regression Test Selection (RTS)

Another class of RTO techniques that also works by reducing the size of the existing test suite is RTS. However the process of reduction is different from that of RTM. RTS techniques select a subset of tests from the original test suites that are affected by the modifications made to the original program. Hence this makes most of RTS techniques modification- or change-aware. Formally, Agrawal et al. defines a test suite selection problem as follows [Agr+93]:

Definition 2.2.2 (Test Suite Selection Problem)
Let: *$t(P, T)$ be the testing function that outputs the result of the test run of a program P with a test suite T.*

Given: *The test suite T, the original program P, the modified program version P'.*

Problem: *Determine a minimal subset T' from T such that $t(P', T') = t(P, T') \Rightarrow t(P', T) = t(P, T)$.*

By the definition, we can clearly see that the efficiency of a subset T' is limited by the efficiency of an original test suite T i.e. T' can be at most as efficient as T. And from

the earlier classification of test cases by Leung and White in section 2.2, we can conclude that this subset T' comprises the retestable test cases of T. This subset ideally should contain all the *faults-revealing* test cases of a modifiled program P' [YH07].

Because the purpose of RTS techniques is to save resources during the regression testing process, it is important to validate that the deployed techniques are indeed cheaper than the approach to rerun all tests. Therefore, the runtime of executing the optimization technique on the test suite in combination with the runtime of executing the optimized test suite is limited by the run time of executing the approach to rerun all tests. An RTS technique is only considered efficient if and only if its combined runtime is less than that of executing all tests.

Although RTS has been one of the popular topics in the field related to regression testing, finding a good selection is a difficult problem. Despite substantial results, several studies indicated that systematic or automated RTS is deployed only by a small fraction of software industries [Bis+11]. A large portion still uses some forms of manual analysis. This problem is extremely difficult in a complex software as many test cases can be overlooked due to a lack of knowledge regarding the source code itself to precisely select only the retestable test cases [Bis+11]. The most prominent approach of RTS techniques is the coverage based selection techniques which are explained in subsection 2.3.1. On the other hand, no other generic RTS technique that can be applied to different classes of application has been put forward due to the complexity of the problem as well as diverse specifications and requirements of each software system [ERS10]. Hence, depending on the software, different RTS techniques should be deployed.

2.2.3 Regression Test Prioritization (RTP)

RTP is another class of RTO techniques which concerns with the execution ordering of the test cases. RTP seeks to find an ideal ordering of test cases based on some desirable properties or criterion [YH07] such as the fault-detection rate, the execution time of the last test run. It is important to note that unlike RTS, RTP techniques do not depend on either versions, original or modified, of the software and assume that all test cases will be executed in the defined ordering [YH07]. Formally, Rothermel et al. defines test suite prioritization problem as follows [Rot+01]:

Definition 2.2.3 (Test Suite Prioritization Problem)
Given: *The test suite T, a set of permutations PT of T and a function $f : PT \to \mathbb{R}$ that maps PT to real numbers.*

Problem: *Find $T' \in PT$ such that for all $T'' \in PT$ and $T'' \neq T'$. $f(T') \geq f(T'')$.*

In the above definition, PT represents all possible orderings of P and a function f, when applied to any ordering will output a value such that it can be compared to find an ordering with the highest value (with the highest value being the most superior). Table 2.1 shows an example of a test suite and its fault detection information. Based on the fault detection rate, it is obvious from this example the most ideal ordering

Test	Fault revealed									
	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

Table 2.1: Example test suite with its fault detection information from [Rot+99]

should execute C first. Nevertheless, there are other prioritization techniques. Rothermel et al. compared nine different techniques in their work and we discuss them briefly in section 2.3. More detailed explanations can be found in [Rot+99] and [Rot+01].

In general, the most significant goal of RTP is to detect faults as early as possible. By detecting faults earlier in the testing process, fixing and retesting can be completed faster. However, there are many aspects to RTP, leading to many possible goals. The list below summarizes the goals suggested by Rothermel et al.:

- To increase the rate of fault detection of a test suite.
- To detect high-risk faults earlier.
- To increase the coverage of code in the *system under test (SUT)* at a faster rate i.e. code coverage criterion met earlier.
- To increase the reliability of the SUT at a faster rate.
- To increase the probability of revealing faults related to specific code modifications earlier.

2.2.4 Hybrid Approach

The last class of RTO techniques, Hybrid approach, is not one of the main branches of techniques mentioned earlier in section 2.2. The Hybrid approach combines the principle of both RTS and RTP. Duggal and Suri reported that there are a number of research proposing the hybrid approach as a solution to the RTO problem [DS08]. These include works proposed in but not limited to [ASK04], [SKS06], and [Won+97].

2.3 Categorization of RTO Techniques

There are a number of studies focusing on different RTO techniques. A few provide an overview and categorization of RTO techniques. The study by Yoo and Harman gives a comprehensive overview and definition of the main RTO classes: RTM, RTS, and RTP [YH07]. Biswas et al. provided a survey on different RTS techniques based on their

methodology [Bis+11]. The systematic literature review by Singh et al. presented an extensive study of many different RTO techniques and categorized them based on their input data. The following subsections provide an overview of this categorization based mainly on [Sin+12] but also include details from the other publications. The main focus of this section is to differentiate current state-of-the-art techniques by their data usage as it will be one of the focal points that drives the later concept of the thesis.

2.3.1 Coverage Based Approach

Coverage based approach, as its name suggested, uses code coverage as a metric for prioritization criterion. The idea behind this approach is that the more coverage the test achieves, the higher the probability of revealing faults.

Coverage based approach can utilize different levels of granularity i.e. statement, branch, and function. A study by Rothermel et al. reported nine different prioritization techniques including *statement*- and *branch*-level (decision) coverage prioritization techniques [Rot+99]. Rothermel et al. further divided these techniques into *total* and *additional* coverage. Total coverage is intuitive as the test cases are prioritized based on the coverage measurement of a defined granularity, be it branch or statement. On the other hand additional coverage builds on the idea that more may be gained if coverage information is adjusted after iteratively selects a test case with the greatest branch or statement coverage. Elbaum and Malishevsky, Alexey G Rothermel also later proposed *function*-coverage techniques [EM00].

It is obvious that in order to use coverage based approach, the source code needs to be made available. Nonetheless, Srivastava, Amitabh Thiagarajan proposed the binary code based prioritization technique. The binary form eliminates the overhead of the recompilation step for coverage information collection [Sin+12].

There are further techniques proposed by various researchers which fall under coverage based approach. Further details can be found in [Sin+12] and [YH07].

2.3.2 Modification Based Approach

The second category of RTO techniques is modification based approach, also known as change based approach. As the main concern of regression testing is to revalidate the modifications to the existing software, monitoring these changes is an obvious approach. Many RTS and RTP techniques are based on this assumption.

There are various modification based approach techniques reported in the literature. Wong et al. [Won+97] proposed a hybrid approach, a modification based prioritization technique, first mentioned in section 2.2.4. Further studies include a model based technique using Extended Finite State Machine (ESFM) [KTH05], heuristic based technique [KKT07], and model based using traceability links between models [Sil+10]. Hammant also outlined another technique called *Test Impact Analysis* (TIA), which is extensively researched by Microsoft. This technique is a hybrid of the coverage based and the modification based approach. The key idea is to analyze the call-graph (in essense

utilizing code coverage) of the current source code to determine which tests need to be re-executed after the modifications [Ham17].

2.3.3 Fault Based Approach

While coverage based approach techniques consider whether a statement, a branch, or a function (depending on the level of granularity) is covered by a test case, they mostly ignore the information from the test execution itself. With each run of the tests being executed, they either pass or fail. This information is valuable as it reveals which test cases are more likely to expose faults than others as well as what type of faults they are more adept to [Rot+99].

Following the earlier assumption, Rothermel et al. proposed prioritization techniques called *Total fault-exposing-potential (FEP)* and *Additional-FEP* and introduced in their experiment a so-called *weighted average of the percentage of faults detected (APFD)*. Using APFD with an example from Table 2.1, the ordering C–E–B–A–D shows superior fault exposing potential than E–D–C–B–A which is in turn still better than A-B-C-D-E. [EM00] later introduced function-based FEP techniques, fault-index based techniques as well as techniques that are a combination of both.

2.3.4 History Based Approach

History based approach works based on historical test execution data. As historical data can be useful in increasing effectiveness, hence reducing cost for optimization process. Kim and Porter introduced the first history based RTO technique. It integrates with the notion that regression testing is memory full by using the execution information from the last test run [KP02]. An example of history based approach can be prioritizing test cases based on the execution time from the last test run.

2.3.5 Requirement Based Approach

Requirements can provide useful information for optimization. Srikanth et al. first introduced a system level technique metric *Prioritization of Requirements for Testing* or *PORT* [SWO05] for requirement based approach. This proposed metric is based on four factors presented in Table 2.2. The objective is to detect severe faults earlier hence improving customer perception of software quality. Faults with higher severity are mapped with higher prioritization value. Another two metrics for requirement based approach were also presented by [KM08] and [KM09]. The study by Srikanth, Williams, and Osborne showed that by focusing on higher value functionality, the testing efficiency as well as field fault occurrences were improved.

2.3.6 Others

Aside from the aforementioned categories, there are still other techniques utilizing other different approaches. One of them is genetic based approach e.g. time-aware genetic based prioritization by Walcott et al. [Wal+06] and genetic algorithms prioritization with

Study	Metrics
[SWO05]	Customer assigned priority of requirements, Developer-perceived implementation complexity, Requirement volatility, Fault proneness of the requirements
[KM08]	Customer priority, Changes in requirements, Implementation complexity, Usability, Application flow, Fault impact
[KM09]	Customer assigned priority of requirements, Developer perceived code implementation complexity, Changes in requirements, Fault impact, Completeness, Traceability

Table 2.2: Comparison of different metrics proposed for requirement based approach

a variety of mutation, crossover, selection and transformation operator by Conrad, Roos, and Kapfhammer [CRK10]. Another is composite approach applying more than one type of approaches e.g. component based with modification based approach by Wong et al. [Won+97]. Others that are not mentioned in detail include data flow based, cost based, graph based, model based, distribution based, probabilistic approach, etc.

2.4 Continuous Delivery

The topic of Continuous Delivery (CD) was first introduced by Humble and Farley in their book *Continuous Delivery: Reliable Software Releases through Build, test, and Deployment Automation*. The approach was developed to tackle the problems in software delivery. It asks the question: how can one ensure that the software can be reliably released at any time? To answer this, a few important principles in CD are suggested [HF10]. They are outlined and briefly described in this section.

Deployment Automation Automation is a central theme in CD. While not everything can be automated, the idea is to automate as much as possible. In essence, the deployment process needs to be automated. This is referred to as *deployment pipeline*, an automated software life cycle process from build, deploy, test to release. Deployment pipeline is at the heart of the CD principle and every organization has different implementations for it. Figure 2.1 depicts an example of a deployment pipeline with the commit stage including compile, unit test, analysis, and build installers.

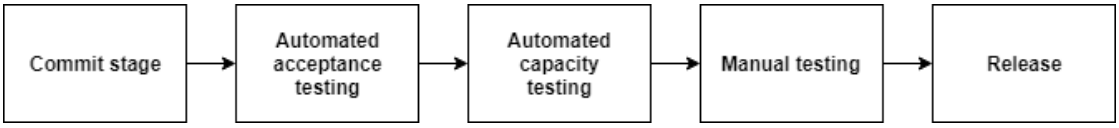


Figure 2.1: Example of deployment pipeline by [HF10]

Configuration Management Everything that is part of the software life cycle from build to release should be version-controlled. This may include the source code itself, requirement documents, test cases, database scripts, external libraries, configuration scripts, and more. This means that we can reliably identify at any given time what is created when, why, and by whom.

Bring the Pain Forward It is important to identify the pain point(s) in the software development process. This can be various things from testing to new technology or team member integration. If the final testing before release always slows down the process, consider having automated testing after every commit or build. If the developers always take too much time because the scope of functionality and requirements are not clear, consider having the requirement documentations done before and not during or after the tasks.

Built-In Quality This concept is closely related to the previous one. The earlier defects are detected, the cheaper it is to fix them. Defects caught after software release are always expensive. Hence, having a good testing strategy is important and should never be left to the end of the deployment process.

Done Means Released Definition of done has to be agreed by the organization. A *done* of one person has to mean the same for another and is ready to be released into production. This is related closely to the next principle where everyone has a shared responsibility toward the project.

Everyone is Responsible for the Delivery Process A software is built by a team and not a single person. Whether a software or a project succeeds or fails depends on the entire team. Everybody should be involved in the delivery process and a greater collaboration within the team should be encouraged.

Continuous Improvement All softwares evolve overtime. Hence it is important that the process itself evolves as well. With each release of the software, the team can reflect over the good points and the pain points and improve accordingly. Therefore, everyone should be involved in this process.

2.5 Previous Works

2.5.1 Lazzer

Plewnia developed a concept for an RTO framework and presented in his thesis a functional prototype named *Lazzer* in 2015 [Ple15]. The key objectives of the framework are to allow an easy implementation of new RTO techniques, to enable easy integration into existing projects, and to support evaluative studies of different RTO techniques. Its design follows predominantly the principle of the separation of concerns. The four main components are client adapter, data store, strategy, and test framework. They are built up around the framework's core as shown in Figure 2.2.

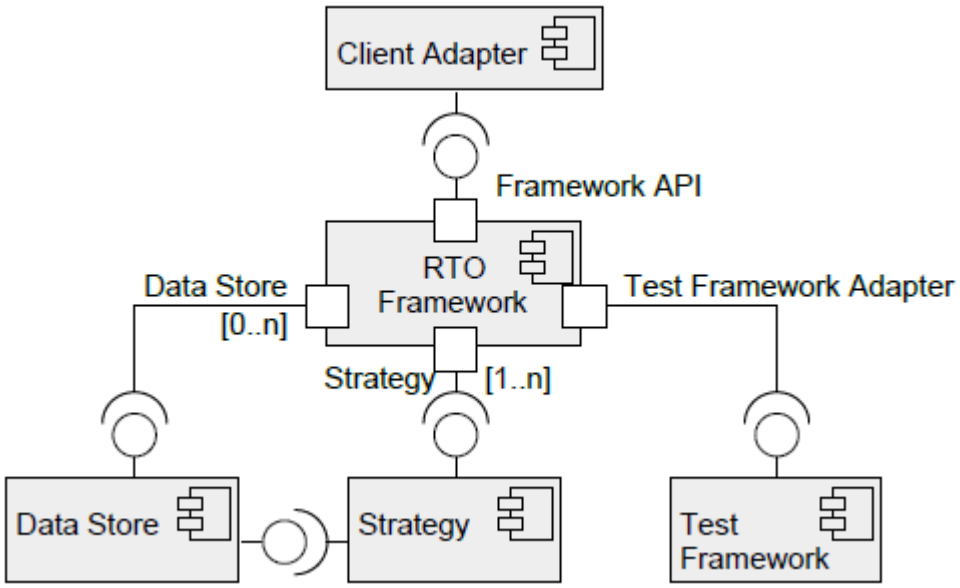


Figure 2.2: Coarse-grained architecture of Lazzer [Ple15]

Strategy is inarguably the main component of any optimization system. In this case, strategy realizes the concept of RTO techniques and encapsulates the optimization logic e.g. a selection technique that selects only previously failed tests.

Data Store is a component used to realize the concept of data collection. Because most optimization strategies cannot be run without some sort of input data, it is binded with strategies e.g. a selection technique based on failed tests requires test history information. A strategy may utilize multiple data stores or none at all and similarly a data store may also be used by multiple strategies. The access between strategies and data stores are delegated by a *Dependency Loader*.

Test Framework is used to implement the test suite for regression testing. It is connected to the central RTO framework via the interface *Test Framework Adapter*. This allows the users of the framework an integration with different test frameworks e.g. JUnit4, JUnit5, etc.

Client Adapter consumes the API provided by the RTO framework. This fulfills the objective where the integration to the framework from the existing projects is needed. In [Ple15], the command line clients and integration with maven are provided as examples.

RTO Pipeline Other than the aforementioned main components, another important concept of this framework is the RTO pipeline. The RTO pipeline realizes the RTO process which is essentially a sequence of tasks. It includes six consecutive stages (see Figure 2.3): test discovery, pre-optimization data collection, optimization, test run, post-test-run data collection, and reporting.

In the test discovery stage, the Lazzer searches for all tests in the test suite via the use of a test framework adapter. Analogously, the test run stage runs the tests in a similar manner. In the pre-optimization data collection stage, data stores used by specified strategies perform data collection tasks before the optimization can start. The optimization stage, as its name suggested, is where the optimization is performed using specified strategies one after another, output from one being an input to another. The final output of this stage is an optimized test suite that is run in the test run stage. After the test run stage, another data collection task is performed to collect the available test results via data stores, identical to the pre-optimization data collection stage. Finally, in the reporting stage, a report of test results is created for evaluation purposes. Different reporting mechanisms can be implemented depending on their intended usage.

In this thesis, the theoretical concept of this RTO framework and its functional prototype Lazzer are used as a basis for the RTO adaptation into the architecture of the delivery system. Details and metrics of the concepts and artifacts which are kept within the RTO framework and which are delegated to the delivery systems framework can be found in chapter 5.

2.5.2 Jarvis

Another main architecture that contributes to the major portion of this thesis is *Jarvis*. In 2018 Jarvis, an architecture for self-organizing continuous delivery system, was presented by Döring in his thesis [Dör17]. Jarvis was designed to tackle two major challenges in the domain of delivery systems, project evolution and modeling usability, with the following requirements: integration of new technology, modularity, activity abstraction, self-organizing, custom PDLs, model validation, best practices, and traceability. Its architecture aims at being flexible and maintainable from the delivery process modeling and the technical viewpoint. Jarvis uses the microservice architecture and Figure 2.4 depicts the overview of layers and services within the aforementioned architecture. The

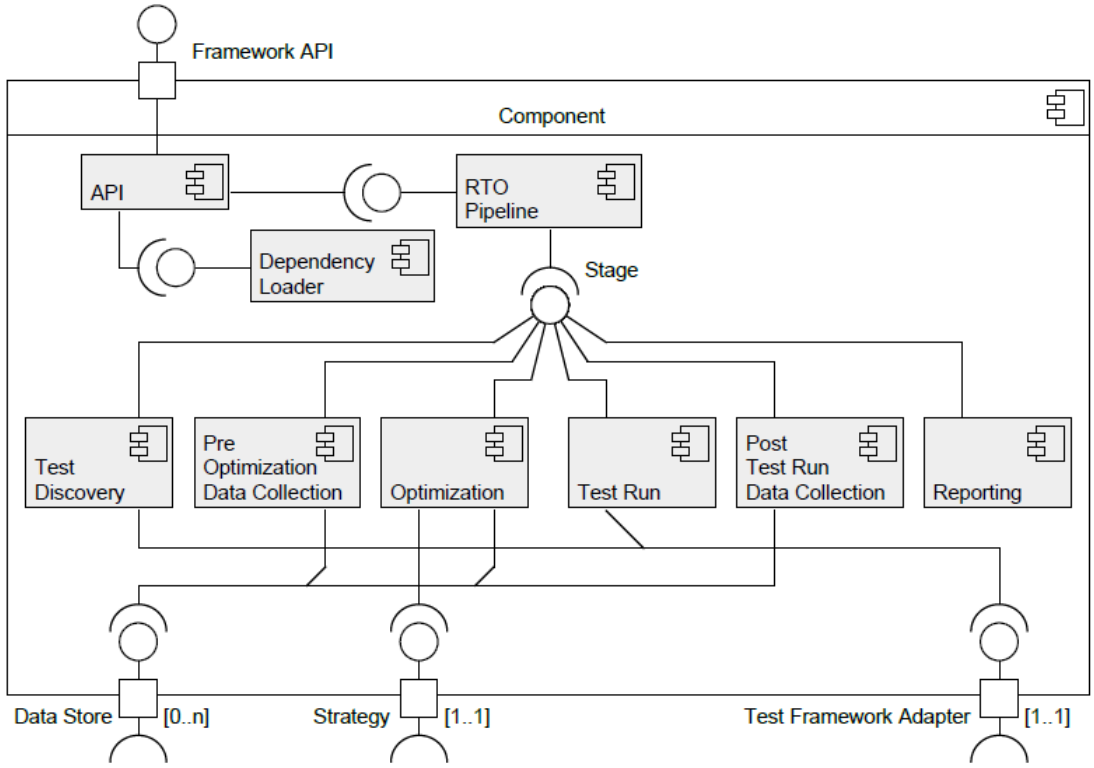


Figure 2.3: Complete architecture of Lazzer [Ple15]

core domain of the architecture that is of interest for the scope of this thesis lies within activity, delivery process management, and infrastructure layer.

Activity Layer contains *Activity Services* that realizes activities as microservices. Each activity service is self-contained and typically encapsulates a functionality of a tool, integrating new technology into the framework. Each service publishes its activity specifications during startup to offer its service as depicted by an interface between the activity and delivery process management layers. An activity service may also use an external tool to realize its implementation as depicted in a similar manner by an interface between activity and external provider layer. An activity service may be a transformation, an assessment or hybrid service depending on the type of commands it provides. This is where most work of this thesis takes place. This thesis also touches upon an extension to the activity concept. Details are discussed in chapter 5.

Delivery Process Management Layer houses services for delivery model management and execution. *Model service* manages the delivery models including import, validation, and translation of external to internal models. *Process planner* plans and optimizes a specified delivery model. *Orchestrator* controls the execution of required activity services.

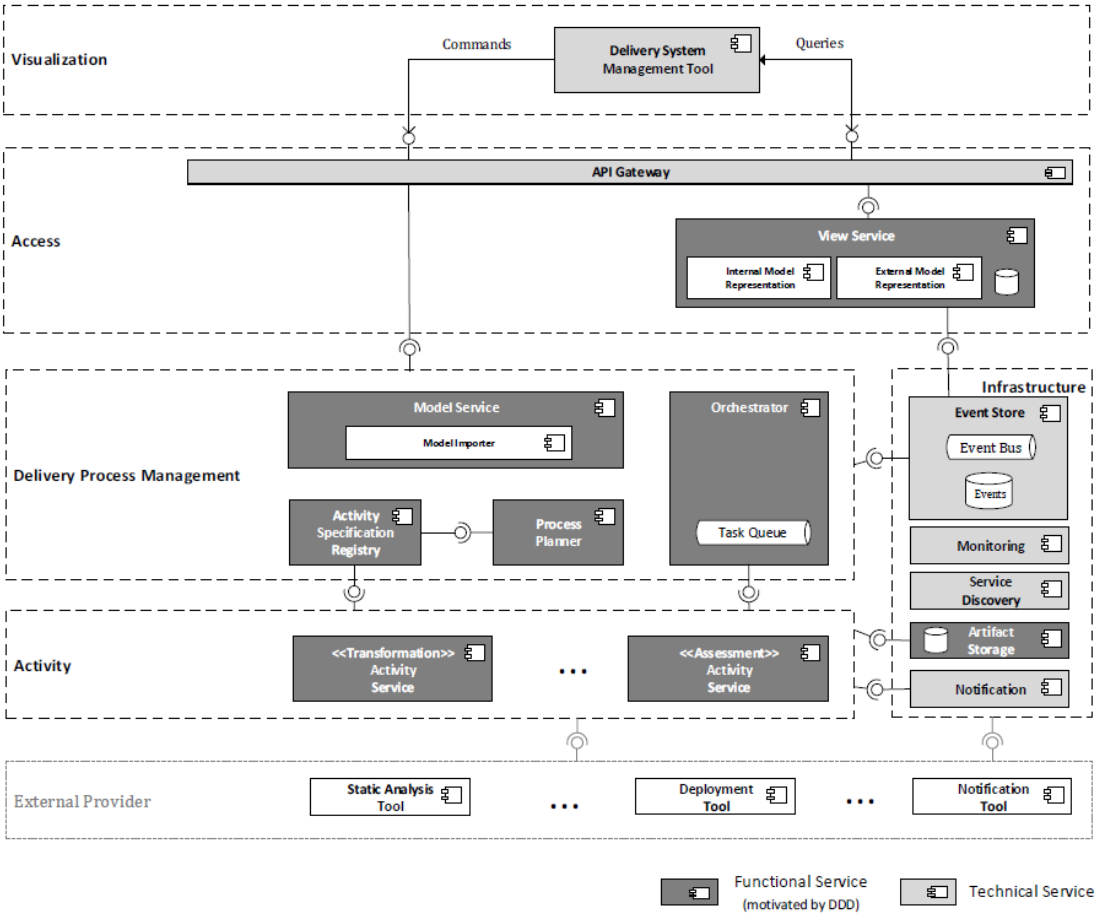


Figure 2.4: Architecture of Jarvis delivery system [Dör17]

Activity specification registry manages all available activities and their specifications. The delivery process management layer, at its heart, realizes the delivery system of Jarvis.

Infrastructure Layer contains the internal services that provide general purpose functionalities such as communication, transport, and storage used by other microservices in other layers. Except for *artifact store* which provides file persistence required by delivery models, other services provided in this layer are rather technically motivated e.g. *event store* for event sourcing, *service discovery* for the communication to services (locations unknown). Some concepts from our work for the framework extension also take place here.

To explain the complete architecture of Jarvis is out of scope for this thesis. More details regarding the architecture itself can be found in [Dör17].

3 Problem Statement

Contents

In the previous chapter, we gave a background overview of the central concepts and of the two existing systems used as a basis in this thesis. In this chapter, we present the problems tackled by this thesis. First we describe the challenges faced, followed by the scope and the requirements.

3.1 Challenge

As described in the previous chapter, the work in this thesis is based on two systems, the RTO system and the delivery system, analogously with one being the host and another being the guest system. Therefore, the integration of two different systems is not straightforward. Because the RTO system has a monolithic architecture and the delivery system is implemented as microservices, the architectures of both systems do not match. In order to integrate the RTO system into the delivery system, it must adopt the microservice architecture as well. The RTO system has to be distilled, broken down, and recomposed as microservices. Tackling this particular challenge is the main objective of this thesis. The adaptation should allow the RTO to run seamlessly as part of the delivery pipeline, while including all the relevant functionalities of the RTO process. As the end result, the adapted RTO should be as complete as the standalone, monolithic RTO system. Nonetheless, the adaptation has to also maintain the integrity of the delivery system itself by fulfilling all its previously defined architectural requirements.

3.2 Scope

The work of this thesis focuses on the architectural problem presented by the adaptation of the RTO system rather than the implementation of the RTO itself. We do not seek to improve the existing functionalities of the existing RTO system. Within the scope of this thesis, we aim to answer the following questions:

1. How to design the architecture of RTO adaptation such that it fits in the architecture of the delivery system?

2. Does the architecture of the delivery system itself need to be changed or extended? If so, how?

3. Does the microservice architecture of the RTO adaptation have any advantages or disadvantages over the monolithic architecture?

3.3 Requirements

Based on the scope outlined in the previous section, this section derives the requirements necessary for the successful adaptation of RTO. The requirements are divided into two main perspectives. We have, on one side, the users of the delivery system that want to integrate RTO into their delivery model and the developers of the delivery system that want to further extend the RTO functionality. On the other hand, we also look at the requirements from an architectural perspective. These requirements are derived from [Dör17] as the existing requirements of the system have to be maintained despite the RTO implementation.

3.3.1 User's Requirements

We can divide the users of RTO into two groups. The first group is the end users. Because the RTO system is now part of a larger system, the users are, in fact, the users of the delivery system itself. We first describe the requirements that the adaptation must provide for this user group in Table 3.1. Because we do not attempt to improve the functionalities, we have to maintain all the previously available functionalities to be as complete as the existing RTO system. Thus, the requirements are taken and adjusted from [Ple15].

FR1.1 describes the main use case of the RTO adaptation from the user perspective which is to integrate RTO and use it as part of the delivery pipeline process for their projects. Since the delivery pipeline is modeled by means of the delivery model, all the services related to RTO process must, therefore, be made available. It further allows users to specify the RTO techniques to be used by adding the optimization services (cf. FR1.2) in the model. To be effective as an RTO system, only the tests in the optimized test suite are run in the specified order (cf. FR1.3) and provide a test report as a result (cf. FR1.4). In addition to functional requirements, there is also a non-functional requirement. The integration of the RTO pipeline in the delivery model for users' project must not require any modification to the existing test suite (cf. NR1.1).

The second user group of our RTO adaptation is the developers. In Table 3.2, we provide the requirements associated with this user group. The use cases for the developers are associated with two main parts. The first part is the optimization itself. As we identified earlier that the RTO adaptation should allow the users to apply different RTO techniques (cf. FR1.2), the RTO adaptation must in turn allow the realization of these

Req.	Definition
FR1.1	The RTO adaptation must provide RTO functionality as services to be used in the delivery model i.e. as activities in the delivery model. The required input parameters must be configured and passed according to the guideline of the delivery system.
FR1.2	The RTO adaptation must allow the users to apply their chosen RTO technique(s) in the form of a service in the delivery model with the order indicated by the users or none at all, in a similar manner as FR1.1. The optimization should be run in the specified order.
FR1.3	The RTO adaptation must run only the tests in the optimized test suite in the specified order.
FR1.4	After the test run, the RTO adaptation must provide a report of the RTO test run including but not limited to: • the result of the test run whether it passes or fails, • the test run execution time, • the details on the failed tests.
NR1.1	The RTO adaptation must be ready for use without any modification to the existing test suites.

FR - Functional requirements

NR - Non-functional requirements

Table 3.1: List of requirements from the perspective of the users

Req.	Definition
FR2.1	The RTO adaptation must allow the developers to realize their own RTO techniques as a service that provides the optimized test suite as a result.
FR2.2	The RTO adaptation must provide the developers with a service to retrieve and store optimization data to be used with the implemented RTO techniques.
FR2.3	The RTO adaptation must allow the developers to implement their own data collection service to collect the optimization data. The data collection service is usually needed in two scenarios: • before the optimization is performed, • after the tests are executed.
FR2.4	The RTO adaptation must allow the developers to integrate a new data storage for storing the optimization data.
FR2.5	The RTO adaptation must allow the developers to implement an extension for a new test framework integration via an API.

FR - Functional requirements

Table 3.2: List of requirements from the perspective of the developers

RTO techniques. Because different projects may require different RTO techniques, the ability to extend and realize customized optimization services are the key of generalized RTO integration (cf. FR2.1). In order to realize new optimization techniques, the availability of the optimization data is crucial. FR2.2 identifies this necessity. Hence, the RTO adaptation must provide a service to retrieve and store these data such that it can be used to implement the techniques. It must also allow the integration of a new data source or storage for the optimization data (cf. FR2.4). In order to collect optimization data, it should also allow the developers to implement their own data collection service (cf. FR2.3).

The other part of the use cases deals with the test framework itself. Since different projects may use different test frameworks and implementing the adapters for all the test frameworks is not possible and against the objective of this thesis, the RTO adaptation must allow the developers to realize their own test framework integration by the use of an API (cf. FR2.5).

3.3.2 Architectural Requirements

Another perspective on the requirements is architectural requirements. Since we work with Jarvis as the delivery system, the architectural requirements are taken from the identified requirements of [Dör17]. As the architecture has to be extended to accommodate the

RTO adaptation, the original requirements must be maintained to preserve the integrity of the system. In Table 3.3, we enumerate these requirements. The requirements are presented in the reference to the context of RTO adaptation and associated extensions to the architecture. For the sake of having a complete list, we also include unaffected requirements, meaning that the adaptation of RTO has neither a positive or negative effect on them.

Req.	Definition
AR3.1	**Integration of Heterogeneous Technology** The RTO adaptation must allow an easy integration of different techniques and the tools that are needed to realize them. This requirement coincides with FR2.1 in the scope of RTO.
AR3.2	**Modularity** Following the modularity of the architecture, RTO adaptation must be realized in a modular approach to provide maintainability and flexibility. To this end, each element in the RTO process must realize its module instead of a single RTO module realizing the whole process.
AR3.3	**Activity Abstraction** Similar to AR3.2, the RTO pipeline must be abstracted as activities, with each encapsulating a single functionality of the RTO process to be used in the delivery model.
AR3.4	**Self-Organizing** To maintain the self-organizing context in the delivery process, RTO adaptation should acquire necessary information on its own and does not require additional information to be provided by the users from the modeling perspective. Any extension to the activity domain must adhere to this principle.
AR3.5	**Custom PDLs** Not affected
AR3.6	**Model Validation** Any extension to the activity domain to support the RTO adaptation should preserve the model validation property using the two phase validation process: validating external delivery model syntactically and internal delivery model semantically.
AR3.7	**Best Practices** The RTO adaptation should still ensure continuous delivery best practices in the delivery process.
AR3.8	**Traceability** Not affected

AR - Architectural requirements

Table 3.3: List of architectural requirements from [Dör17]

4 Related Works

Contents

The previous chapter identified the challenge, scope, and the requirements of the RTO adaptation. This chapter investigates existing works and tools for automated RTO and related concepts.

4.1 Tools

This section provides a comprehensive list of tools that realize automated RTO. The list includes only currently available, both commercial or open-source, tools and does not claim to be complete. There exists many more regression test automation tools in the market but they are excluded due to either the lack of test optimization or it is ambiguous to determine from their documentation whether the test optimization is included as part of their implementation. We also exclude other tools that were mentioned in some literature about RTO that were not made public or are no longer available.

It is also important for the work of this thesis to analyze the tool against the user's requirements. We omitted the analysis for the requirement from Table 3.1 because it is the requirements that can be evaluated only after the integration of the tools as part of the delivery system. Thus, we analyzed the tools only against the requirements from Table 3.2 to determine whether they are achievable.

4.1.1 OpenClover

OpenClover [Opeb] is an open-source code coverage tool. It was first know as Clover and was developed by Cenqua and later by Atlassian®. In April 2017, Atlassian decided to publish the source code of Clover, making it free and open-source. The key features

of OpenClover are code coverage measurement and test optimization. It also provides a functionality to support tool integration to existing CD platforms as well as other systems and intergrated development environments (IDEs). The implementation details of OpenClover are from their documentation [Opea].

Code Coverage is the main selling point of OpenClover, measuring the percentage of code covered by the automated tests. For its code coverage measurement, it uses the *source code instrumentation* approach in which the source codes are instrumented and compiled with a normal compilation tool to produce the measurement. There are different levels of code coverage offered by OpenClover: method, statement, and branch coverage. The first two are self-explanatory. The branch coverage (also known as decision coverage) measures coverage of possible branches in flow control structures.

Likewise, OpenClover also provides two different kinds of coverage: global and per-test coverage. Global coverage is self-explanatory. Per-test coverage records code coverage per test basis. This functionality is also available on distributed mode with tests running on separate Java virtual machines (JVMs). Per-test coverage is used as a basis for running OpenClover's test optimization.

Test Optimization offers both selection and prioritization RTO techniques. As it is a code coverage tool, its RTS technique can be categorized as using coverage based and modification based approach (see section 2.3.1 and section 2.3.2). When the changes are made to the source code, OpenClover re-instruments the codes to create a new code coverage measurement and compares the measurement with the previous one to select only modified tests. On top of the RTS technique, it also uses the RTP technique to order the tests to encourage early failures. According to its documentation, the tests are ordered by "*any failed tests from the previous build, all tests covering modified code, then in ascending order by test invocation time*". Hence, we conclude that it uses history-based approach (section 2.3.4) and possibly other approach (see section 2.3.6) to order the tests by their invocation time. It is worth noting that the users have no direct influence on the executed test optimization other than the selected level of code coverage. New techniques cannot be introduced by the users.

Tool Integration OpenClover provides an integration into existing CD platforms, namely Jenkins, Hudson, Bamboo and Anthill Pro. The integration is provided via by means of plugins but only for the code coverage functionality not the test optimization. OpenClover also provides many integration plugins to other existing systems and IDEs[1].

Requirement Analysis The main requirement for the RTO adaptation is to allow the developers to realize their own RTO techniques. However, OpenClover is limited to coverage based and modification based techniques. Thus, it does not fulfill FR2.1. There

[1] A list of all OpenClover's supported platforms:
`https://www.openclover.org/doc/manual/latest/general--supported-platforms.html`

is also no possibility to implement customized data storages because only Clover's own techniques are used, making FR2.1, FR2.2, and FR2.3 irrelevant. Nonetheless, it provides support for multiple test frameworks such as JUnit3, JUnit4, TestNG, etc., but it does not provide the possibility to implement customized integration to other frameworks as stated for FR2.5. OpenClover should be used to implement the RTO technique as an activity within our RTO adaptation instead of being used as the RTO framework itself.

4.1.2 Tricentis Tosca

Tricentis Tosca [Tri] is a commercial continuous testing platform that provides test automation and optimization functionalities that focus on facilitating Agile and DevOps. The key feature is in providing testing automation for different type of testing e.g. API testing, cross-browser testing, mobile testing, risk-based testing, etc. with minimum adoption and maintenance effort. It also provides test optimization functionality as well as application life cycle management (ALM)/DevOps integration. Since Tricentis Tosca is a commercial software, the information provided about its implementation is vague and not very detailed.

Test Automation is the key feature of Tricentis Tosca. As mentioned earlier, it provides various types of testing automation. This automation uses a model-based technique.

Test Optimization Tricentis Tosca uses risk coverage optimization. Based on the risk assessment of test target's functionalities, it minimizes the number of test cases and further uses its linear expansion methodology to refine the list. By its description, we assume that its RTS uses either a requirement based (section 2.3.5) or other approach. Its documentation does not indicate the use of any RTP technique.

Tool Integration Because its aim is to facilitate the software development life cycle using Agile and DevOps approach, it provides an out-of-the box integration into various ALM/DevOps/CD tools[2] e.g. Atlassian Bamboo, Jenkins, etc.

Requirement Analysis We cannot evaluate this because we do not have access to the internal architecture of the software. However, from its documentation, we assume that it does not provide possibility to extend the test optimization with customized RTO techniques.

4.1.3 Testsigma

Testsigma [Tes] is a commercial automated regression testing tool and has a headline targeted at agile and continuous delivery. The main domains of Testsigma are web, mobile, and web services regression testing. It has a complete user interface (UI) for regression test automation.

[2] A list of all Tricentis Tosca's supported systems:
`https://www.tricentis.com/devops-alm-tool-integrations`

Test Automation is integrated into the existing continuous delivery pipeline. Testsigma can schedule and run automated regression tests automatically with each build or at other predefined intervals.

Test Optimization Testsigma provides an ability to filter and prioritize tests based on test case priority, critical modules/requirements, and test case types (unit, integration, functional, and system). We can consider their RTP techniques as both requirement based and as other approach. Users have an ability to define their own filters and rules in the allowed scope. Testsigma also uses the RTS technique to automatically rerun only the tests that are affected by the source code modification. This approach is modification based. Testsigma also uses test parallelization to run the test concurrently to reduce the test execution time.

Tool Integration Testsigma also provides an option to integrate into CD tools such as Jenkins, CircleCI. In the case of Jenkins, it can be integrated via plugin to automate the test execution for each successful build in Jenkins by adding a task to the Jenkins pipeline.

Requirement Analysis Similar to Tricentis Tosca, we cannot evaluate this against our requirements.

4.1.4 Infinitest

Infinitest [Inf] is an open-source continuous testing runner tool with a plugin integration for IDEs. Infinitest encourages the concept of test-driven development as it runs the corresponding tests every time the source code is changed and recompiled.

Test Optimization Infinitest uses the RTS technique to rerun the tests that are affected by the changes in the source code each time it is compiled. By definition, the technique is categorized as modification based approach. The tests can also be included/excluded by a manual filter.

Tool Integration As Inifinitest is used during development itself, it does not provide an option for integrating into any delivery systems. However, it provides plugins for two IDEs, namely Eclipse and Intellij. The test is indicated with a color-coded status bar as failed, passed, syntax-errored, or waiting for changes.

Requirement Analysis As Infinitest uses only its own RTS technique, its architecture does not provide the possibility to flexibly implement new RTO techniques which is the most important requirement for our RTO system (see FR2.1). Consequently, it has no use of data storages for collecting and storing optimization data because it uses only its own technique. Thus, FR2.2, FR2.3, and FR2.3 become irrelevant. However it provides

integration options with various test frameworks by allowing developers to implement new test integration via an API, fulfilling FR2.5.

4.1.5 Ekstazi

Ekstazi [Eks] is described as a lightweight and scalable java library for regression testing. It is built for RTS research studies and has been described in various publications[3]. According to its website, the research studies used over 30 projects[4] to evaluate Ekstazi in 2015.

Test Optimization in Ekstazi uses modification based RTS technique to select the tests to rerun. The first time it is run, all tests are executed. In subsequent times, only the tests affected by the code changes are rerun.

Tool Integration Ekstazi library is currently distributed as a binary. However, it also provides an integration as a Maven plugin, Ant task and Java agent. There is no built-in integration into any existing CD tools.

Requirement Analysis Similar to OpenClover and Infinitest, Ekstazi only provides possibility to use its own RTS technique and does not provide easy integration for new techniques for FR2.1. Information regarding other requirements is not available.

4.1.6 Test Load Balancer

Test Load Balancer (TLB) [Tlbb] is an open-source test load balancing tool. TLB is written in Java and can be used on all platforms. According to its website, TLB was built to support both JVM and non-JVM based languages and runtimes. Therefore it is independent of the build and test framework as well as programming language of the tests.

Test Optimization The key feature of TLB is to optimize tests by means of load balancing as it automatically partitions tests such that they can be executed in parallel either on the same or different physical/virtual machines. TLB consists of two primary components [Tlba]:

- A server: a repository of test run data (test execution times, test results, etc.)
- A balancer: partitions and re-orders tests, given a server URL. A balancer further consists of two major functional components, a splitter and an orderer. An orderer is where the RTP technique is used.

[3] A list of publications for Ekstazi: `http://ekstazi.org/research.html`
[4] A list of projects used for Ekstazi's evaluation: See footnote 3

TLB has a built-in orderer that utilizes `FailedTestsFirst` strategy, meaing that the tests that failed in previous runs are executed first. This is a history based approach. Nonetheless, TLB provides users a functionality to define their own strategy and feed it to the orderer.

Tool Integration Though most build servers are capable of parallel execution, the decision still needs to be made on what tests need to be run on what machine. TLB provides this parallelization of tests. As TLB is a library, there is no integration into CD tools.

Requirement Analysis TLB uses a different approach to optimize regression testing which is to parallelize the tests. This cannot be categorized to either RTP or RTS technique. Nonetheless, it does provide the possibility to order the test by easily implementing a customized orderer, which is essentially an RTP technique. This partially fulfills FR2.1. In TLB, the server also acts as a data storage but only for historical test data. Thus, it does not fulfill FR2.2, FR2.3, and FR2.4 because only one possible data storage can be used and there is no possibility for extension. Nonetheless, it provides a component called test runner which acts as an adapter API for the underlying test framework. Hence, we consider FR2.5 fulfilled.

4.1.7 Summary

The earlier section outlined the tools for automated RTO. Table 4.1 summarizes the findings. From the features' point of view, we can divide the tools into two categories: a complete system and a library. The first three belong to the first and can be used as a complete testing system as part of the delivery pipeline in tools such as Jenkins. The rest that belong to the later category can be used during the development process and integrated to other tools via manual effort. Only one tool, TLB, offers an ability to implement customized RTP strategies. Nonetheless OpenClover, Tricentis Tosca, and Testsigma provide a functionality to adjust and configure the optimization. Interestingly, we did not find any existing delivery system with a built-in RTO as part of its delivery pipeline. The RTO functionality can only be realized using integrated tools such as the aforementioned ones.

According to the requirement analysis, we found out that all of the tools are not suitable for the RTO adaptation because there is always at least one requirement that is not achievable by all the tools. This is mostly due to the fact that they all implement their own RTO techniques and rely on a single type or selected few, with no or little possibility for flexibly implementing new RTO techniques. Consequently, this makes the use of data storages irrelevant in this context. Also, it is impossible to use the non open-source tools for the adaptation regardless of their requirement fulfillment because of the source code availability problem. However, for OpenClover, it can be used to implement its own activity for test coverage measurement for new RTO techniques and TLB for adding the test parallelization aspect to the RTO process.

Tools	Open-source	Technique	Approach	CD integration
OpenClover	yes	RTS,RTP	CB,MB,HB,OB	yes
Tricentis Tosca	no	RTS	RB,OB	yes
Testsigma	no	RTS,RTP*	MB,RB,OB	yes
Infinitest	yes	RTS	MB	manual
Ekstazi	no	RTS	MB	manual
TLB	yes	RTP*	HB**	manual

CB - Coverage based, MB - Modification based, HB - History based

RB - Requirement based, OB - Other approach

* Also uses test parallelization

** Customized implementations allowed

Table 4.1: A summary of the list of tools discussed in this section

4.2 Related Concept

This section presents a short discussion on the related concept to this thesis. The concept is not used directly in the actual development of design and implementation of the thesis.

4.2.1 Tools and Materials Approach

The Tools and Materials (T&M) approach was first introduced by Züllighoven in his book *Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools and Materials Approach*. The T&M approach is described as an *application-oriented* approach in software development with the focus on users, their tasks, and the business process of those involved. It uses concepts of the model and design metaphor to shape the software such that the structure of the application domain and the realized system should be similar. Below we provide a summarized overview of these concepts taken from [Zül05].

Application Orientation As described earlier, software development using application orientation focuses on the users, tasks, and business process. The characteristics of the application-oriented software are: (1) the functionality is oriented to the tasks to be solved within the application domain, (2) the system is easy to use by its users, and (3) the processes defined can be easily adapted to the actual requirements. It is crucial that the developers understand clearly the tasks required to be accomplished in the application domain. This might mean that they have to gain knowledge in the specialized domain.

Guiding Metaphor is a term that helps realize the unifying view in the software development process such that both the users and the developers can understand and design the software. It helps guide the development of functions and behaviors of the system. Züllighoven used a *workplace* as a guiding metaphor in his book.

Design Metaphor uses the real-world objects to describe a concept in the application system. It helps create a common understanding for users and developers as the ideas are represented in understandable terms. With the example of *workplace*, Züllighoven defined the designed metaphors as: *tool*, *material*, *automaton*, and *work environment* to match the same context as the guiding metaphor. A tool supports work processes and actions. A material refers to a work object that given actions, is turned into the work result. Materials are manipulated by tools. An automaton runs a task and produces a defined result. This is fully specified a priori. Finally, a work environment defines the place where work is performed. In the context of RTO (see section 5.2), we can look at the optimization and test execution task as tools and the tests themselves as materials where a test report is produced at the end of the process.

The aforementioned concepts provide all parties involved a common perspective and terms of the application domain. However, we still need more technical ground for the developers. This brings us to the conceptual patterns that the actual implementation is dependent on. They realize important analysis and design of the system. One conceptual pattern that is similar to our conceptual work for the RTO adaptation is the *Probe Pattern*.

The Probe Pattern

is one of the conceptual patterns presented in the book by Züllighoven. Figure 4.1 shows the structural view of the probe pattern. The probe pattern is an extension of the *Technical Automaton Pattern* which in turn extends the *Automaton Pattern* (also shown in Figure 4.1). Because the implementation of RTO requires optimization data to be stored and retrieved (depending on the optimization techniques used), this requirement can be viewed using the probe pattern.

Automaton is used to tackle the problem of recurring activities or routine by automating them. We can implement part of the activities as an automaton. In the case of RTO, we routinely store data before the optimization and after the test run, this process can be viewed as an automaton.

Technical Automaton extends the concept of an automaton further by mapping domain-specific solutions or devices to application system models in the software. The characteristics of the real systems are encapsulated within a technical automaton. A technical automaton informs other components in the software of state changes and events.

Probe extends the concept further because the clients of a technical automaton may not require to know the entire but only partial state of the technical automaton. A probe is defined by Züllighoven as a component that can determine a measurable value of a technical automaton and can be set to read and update this value. Referring back to RTO data collection, we can view the component that stores optimization data as a probe

as it is interested only in partial information produced by test run or other components such as code coverage data.

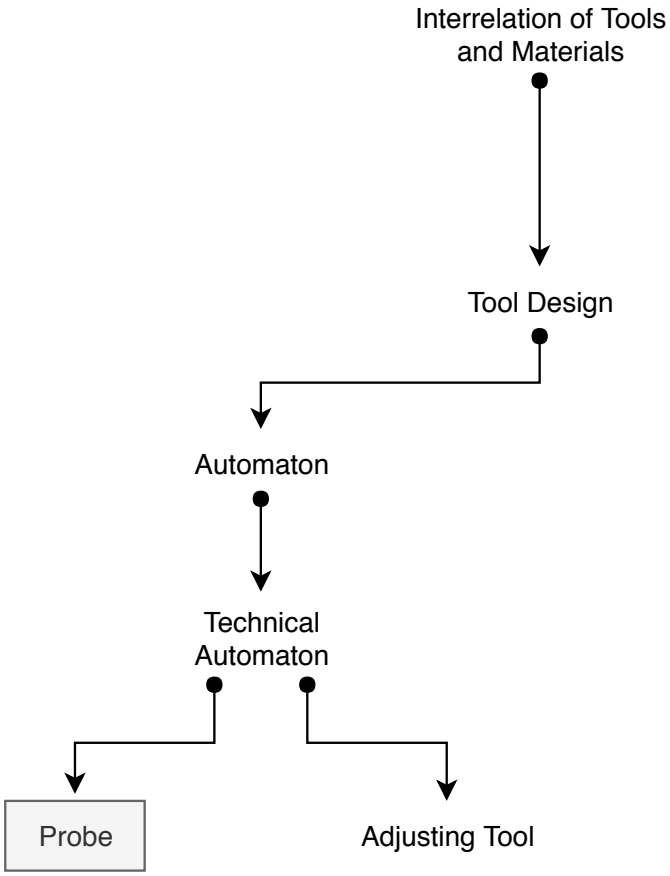


Figure 4.1: Probe pattern diagram from [Zül05]

We provide in this section the T&M approach and the probe pattern which look at the design and adaptation of RTO, specifically to the optimization data collection process of RTO, on another perspective.

5 Concept

Contents

This chapter tackles the problem and the design decision from the RTO adaptation. First, we introduce two major architectural concepts that will be mentioned throughout the rest of the thesis. Then, based on the principles of Domain Driven Design (DDD) [Eva04], the domain model of the RTO is presented and the core domain of the delivery system is extended. The last part of this chapter presents the conceptual architecture of the RTO adaptation, how the architecture changed from a standalone RTO framework into part of the delivery system.

5.1 Background

When dealing with building a software to support multiple functionalities and clients, often the questions arise of how the architecture should be designed. In order to establish a better understanding for the architecture of the RTO adaptation, we first provide a brief overview of the two architectures that are mentioned often throughout the course of this thesis. Because we also deal with migration of architecture, the refactoring process is also presented.

5.1.1 Monolithic Architecture

Monolithic, as its name suggested, means that all components of the software are unified as a single piece. Monolithic architecture is designed to be self-contained. The

components within the software, though can be separated into different modules, are interconnected and interdependent. The software is deployed as a single application, consisting of all the defined functionalities, be it database access, user interface, or message exchange. As a result, monolithic softwares are simple to develop, test, and deploy. While monolithic softwares have their advantages, challenges often arise as they grow over time. When changes are made to the existing software, the whole system is affected as the components within are tightly coupled. At some point in time, the software can grow too large and too complex that it becomes an obstacle to continuous deployment practice [RS16]. Nonetheless, Fowler suggested that it is quite a common pattern that many successful microservices started out as a monolith and was broken up over the course of its development [Fow15].

In the context of this thesis, the existing RTO prototype, Lazzer, is of a monolithic architecture where its usage can be accessed via the framework API. As depicted in Figure 2.3, we can clearly see that all the RTO functionalities are composed within the framework itself and the clients of the framework can access and run them only by using the provided API.

5.1.2 Microservices

As opposed to a monolithic architecture, microservices are composed of components and services. At its core, a microservice architecture is an approach to develop a software or application as a collection of smaller services, with all being independently deployed and run [LF14]. A service can actually be viewed as a small monolithic application with its own business logic and only interacts with other services via their exposed interface, resulting in the loose coupling between services. The microservice architecture takes a direct advantage of the problem domain decomposition where each service focuses on solving only a single part of the problem. For example, user interface specialists can focus on the view service, while database teams can focus on database access service. This means that the developers can focus on what is important and deploy more frequently because of the modularity of the architecture.

The delivery system, Jarvis, employs the microservice architecture (see Figure 2.4). This ultimately means that in order to integrate RTO into the system, we must also adopt the microservice architecture as well.

5.1.3 Refactoring

Refactoring is an unavoidable task when working with two different software architectures. It is referred to as the process of changing a software system such that it does not alter the external behavior but improves the internal structure of the code [Fow+99]. Refactoring always goes hand in hand with testing because only through testing that we can guarantee that the refactoring is executed safely [LR06]. Refactoring is also a continuous process, with no set ending. It is usually carried out in response to eliminate *bad smells*. The smells can be weaknesses in the code structure, design, or even architecture itself (violation of design principle). The benefits of refactoring are multi-fold and listed as follows [Fow+99]:

- To improve the design of software: When the code is continuously changed, it loses its structure. The loss of structure makes it difficult to understand the design in the code, consequently difficult to further preserve it. This can lead to duplication of code, resulting in violation of *once and only once* principle and so on.

- To make the software easier to understand: It is important to write readable and understandable code for the future development as well as for other developers in the team.

- To help in finding bugs: Good understanding of the code goes a long way in spotting bugs.

- To make the development faster: Refactoring helps the developer develop code faster. Because refactoring improves the code and design quality, developers can focus more on adding new functionalities instead of trying to understand the code and fixing bugs.

There are usually a few scenarios where refactoring takes place e.g. during bug fixes, code review, etc. The most common one is when a new functionality is added to the software. This is the case for us as we aim to integrate RTO into the delivery system. The big factor here is that the current design of the architectures does not allow a good integration of two systems easily because of the architectural mismatch. Thus, we need to refactor the RTO system.

According to Lippert and Roock, the classic approach to software design where a complete design is created up front before starting an implementation is rarely feasible, especially in combination with the modern approaches such as agile development process. He presented the idea of a stepwise process called *emergent design* where the software design is continuously adapted to the changing conditions and requirements [LR06]. This is the principle that we used for our refactoring process, demonstrated through our iterative development approach in section 6.2.

Refactoring a Monolith into Microservices

Although refactoring includes tasks like eliminating duplicate codes, rearranging inheritance structure, and so on, refactoring can also be practiced on a higher level. For instance, our work focuses on architectural refactoring from a monolithic system to microservices.

Similar to the concept of emergent design, refactoring a monolith into microservices should be done incrementally. New functionality is added gradually. Over time, the amount of functionality in the monolith becomes smaller, until it disappears or becomes another microservice [RS16]. Martin Fowler coins the term *Strangler Application* for this type of strategy. Richardson and Smith presented three different refactoring strategies, following this strategic approach [RS16]:

1. Stop making the monolith bigger: When the application becomes unmanageable, stop making it bigger. A new functionality should be implemented as a standalone microservice, instead of adding more code into the original, monolithic application.

2. Split frontend and backend: This strategy suggests refactoring the monolith by splitting the presentation layer from the business logic and data access layer.

3. Extract services: This strategy works by turning existing modules with the monolith into microservices. When dealing with a large monolithic application, it is important, however, to prioritize which modules to be converted by some criteria.

Because this thesis does not seek to add new functionalities to the RTO system, the first strategy is not necessary. Similarly, we cannot use the second strategy because we do not work with the presentation layer as it is already managed by the delivery system. Nonetheless, the third is the main strategy we use to convert our monolithic RTO into microservices. We first use domain driven design principle to define different domain contexts within RTO and with that in mind, we iteratively extract each context at a time. The rest of the chapter presents the decomposition of RTO problem domain and the concept for implementing RTO as microservices.

5.2 Domain Driven Design

In this section, we apply the principles of Domain Driven Design (DDD), which is the term first coined by Evans [Eva04], to develop the concept for the work. The domain model is the way of structuring the knowledge of a particular domain and pinpointing the most important elements [Eva04]. It is therefore not binded to a single diagram but rather an abstraction of concepts, relationships, and interactions between them. In DDD, there are two main perspectives, namely tactical and strategic design. The concepts such as entities, repositories, and aggregates embody the tactical design but are actually relevant beyond the scope of the DDD. On the other hand, strategic design provides a semantic look into the application domain and addresses the boundaries and contexts in a complex project. It predominantly underlies the importance of understanding the domain knowledge. To this end, we therefore use the principles of strategic design to investigate and identify the relevant domain of our work. In order to achieve this, we follow the three following principles [Eva04]:

Unification Context refers to a consistency within a domain model across contexts which must be explicitly defined.

Distillation is a crucial process to make prominent the contributions or selling points of the system because a lot of time may be spent on surrounding issues instead of the core problem domain of the system.

Large-Scale Structure helps get a clear, big picture of the system. While we use distillation to focus on the core domain, the relationships and interactions of the whole system should be clarify using some large-scale structure to avoid confusion.

We first dive into the problem domain. According to DDD, we must understand our application domain which is RTO process and how it works. When we look at RTO process, we can create a hierarchy of RTO by its functions to ease an understanding of the domain model. As shown in Figure 5.1, RTO can be viewed as consisting of an optimizer and a test executor. The optimizer, obvious in its name, performs all optimization-related tasks. On the other hand, the test executor includes test-related operations where interactions with the actual test suites are required. They include a test discovery to discover the test suites available for the optimization from a project and a test runner to actually run the optimized test suites.

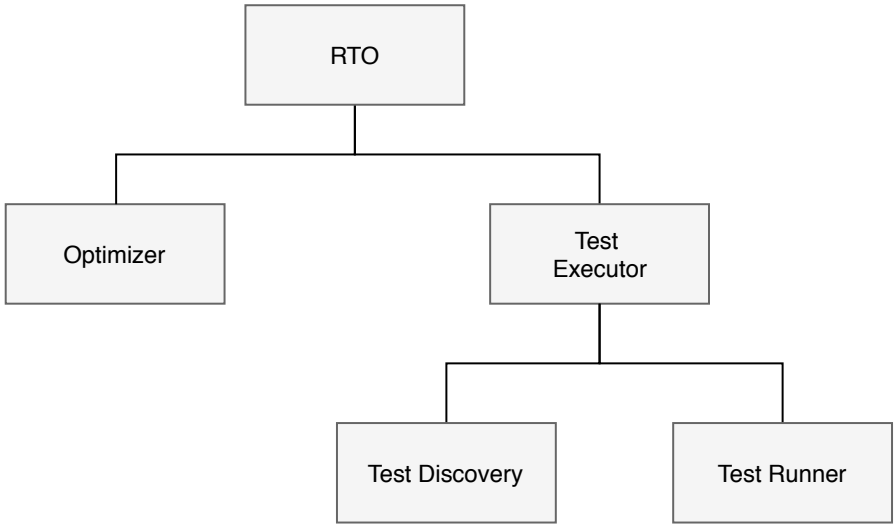


Figure 5.1: RTO process hierarchy overview

The operations in RTO work in a sequential manner i.e. it does not make any sense to run the test suite before optimizing it. To understand how the operations work together, we provide an overview of RTO pipeline in Figure 5.2. The pipeline is similar to the one from [Ple15] but is color coded by the domain contexts. It shows the RTO pipeline with the sequential operations performed. The RTO process starts at the test discovery where the tests are collected for optimization. Depending on the strategies used, pre-optimization data collection may be executed. In fact, this stage may be executed even before the test discovery stage itself as long as it is before the optimization stage e.g. code coverage data is collected during the compile. Using the collected data along with the tests from the first stage, the specified strategies are used for optimization. If more than one strategy is used, they should be run in order and as a result, we get a list of optimized tests. In the next stage, the optimized tests are then executed and the test

results are collected for the future optimization (again depending on the strategies used). Finally, the test results may be used for reporting, further quality control tasks, and so on.

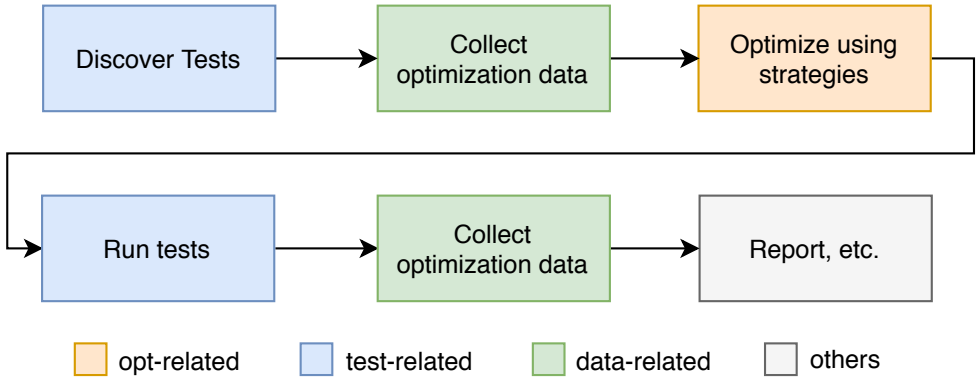


Figure 5.2: RTO sequential pipeline

Distilling into the pipeline, the operations can be divided into four separate areas: (1) optimization-related, (2) test-related, (3) data-related, and (4) others. In fact, the only category which is in RTO-specific domain is the optimization-related one. Test-related operations can be generalized to the testing domain i.e. without the optimization, it is simply a normal testing process. Data-related operations are general data collection tasks that are used in many application domains. The last category is of other domains such as reporting. We will omit this category completely from our domain model because it does not actually contribute to the functionality and the completeness of the RTO process itself, but rather an add-on to the RTO domain.

From the RTO domain knowledge, Figure 5.3 depicts an overview of the RTO core domain. The domain model was developed iteratively, starting from a single component that executes all RTO-related tasks into a more modular model. In this illustration, the domain model is again divided into three main areas similar to Figure 5.2, excluding others. Below we describe each area of the RTO core domain.

Optimization-Related Area

Optimization-related area is the main domain context of RTO. Without it, RTO is not realized and merely a normal regression testing.

Optimizer is an element running the optimization. Each optimizer uses a strategy, either selection or prioritization, and data retrieve from data storage to perform the optimization. It optimizes a test suite produced by the test discovery or an optimized test suite produced by another optimizer by selecting a subset of it or ordering it, depending on the implemented strategy.

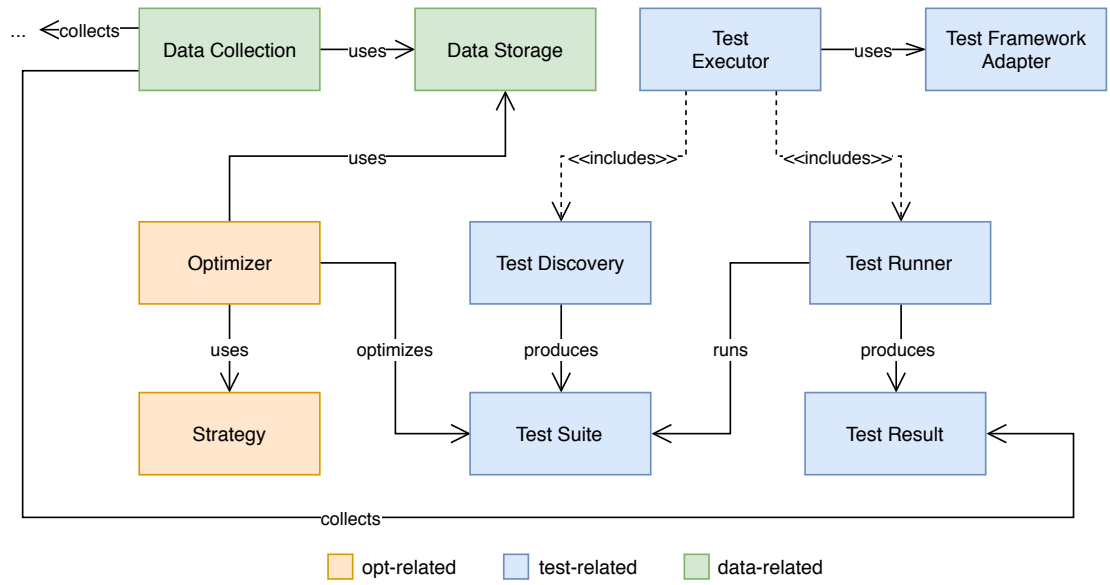


Figure 5.3: RTO core domain model diagram

Test-Related Area

Test-related area is not strictly RTO specific, but rather general in the testing domain. It encapsulates the elements needed in order to have a complete RTO process which is to obtain the tests before and run them after the optimization.

Test Discovery is included in the test executor as it requires an interaction with the actual test suite in the project. As a result, the test discovery produces a test suite which is collected with the help of a test framework adapter. The test discovery is needed to obtain the original test order.

Test Runner is also included in the test executor as well for the same reason as the test discovery. It takes an optimized test suite as an input and runs it via the test framework adapter to produce a test result. A test runner can be used regardless of whether a test suite has been optimized beforehand. A test discovery and test runner together make up a normal testing process without the optimization.

Data-Related Area

Data Collection is an element used purely for collecting data of interest from input data into a data store (persistence unit). In Figure 5.3, data collection collects the test result and other data that might be produced elsewhere in the delivery pipeline and is not part of RTO context e.g. code coverage data, etc.

5.2.1 RTO Services as Activities

Jarvis describes activities as the units of work that make up the delivery process. The activities are built as microservices which are made accessible via their *commands*. An activity can contain multiple commands. Originally, a command is classified into any of the three types: transformation, assessment, and quality gate. Consequently, an activity is classified by its commands' type. An activity that contains more than one type of command are considered a hybrid activity, mostly combining transformation and assessment commands.

In the scope of this thesis, we also introduce a new activity type called *probe* (not to be confused with the probe pattern mentioned in chapter 4) to support our concept for the RTO adaptation. Below we give a brief explanation on each type [Dör17], outline the concept for probe and then go on to describe the classification of the RTO activities. Figure 5.4 provides an overview with an inheritance diagram of activity classification.

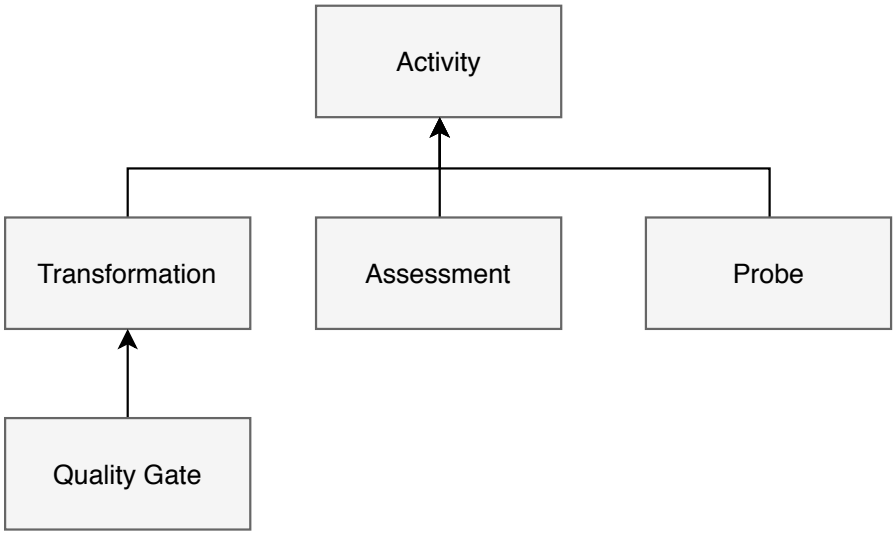


Figure 5.4: Inheritance structure of activity classification

Transformations are the main type of activity in the delivery process. Any delivery model must contain at least one transformation activity. Transformations take one or more artifacts, transform (e.g. mutate, translate, process, merge, etc.) them, and output a new artifact. A good example of transformations is compilation, which takes source code and outputs executable code.

Assessments perform some sort of measurements and output a measurement report. They take one artifact, assess it and output a report regarding the input artifact. A typical example of assessments is testing i.e. tests are performed and output measurable metrics such as success and failure rate as a result.

Quality Gates are a control activity that can abort the execution of the delivery pipeline when the input artifacts do not meet the quality criteria. Each quality gate takes an input transformation artifact together with a corresponding assessment report (output from assessments) and a policy, evaluate the artifact, and either reject or promote it depending on the evaluation result. Because the quality gates can ultimately output promoted artifacts, they are regarded as a special type of transformations.

Probes are not a part of the original types of activity outlined in [Dör17]. However, with the introduction of the RTO domain, there is a need to define a new type of activity. The RTO process demands it because the data collection component described in the previous section does not fit any of three previously described activity. We have deliberated over the inheritance structure of the activity classification whether assessments are simply a special type of probes because they both can output some sort of metric data. In the end, we concluded that probes should be a separate class of activity because their internal function is different from that of assessments and their outputs are not used for any further quality control in the quality gates. In essence, probes do not contain any internal logic, neither transformation or assessment, other than probing/extracting for partial state of data from the input artifact(s). Probes can take one or more artifacts and output meta artifact as depicted in Figure 5.5.

Nonetheless, it would not make sense to create a new class of activity, only to use it in the RTO domain. Thus, we define in the following, the three different levels of data that probes can be used to collect:

1. Regarding artifact(s) produced by an activity during a delivery pipeline run e.g. test history, code coverage, etc. The obvious use case for this type of data is our RTO data collection service.

2. Regarding the delivery process itself (the executed delivery model) e.g. execution time, etc. The typical use case for this type of data can be to collect KPIs data for report.

3. Regarding the delivery system e.g. memory consumption. This can be useful for system management.

In the scope of RTO, we are only interested in the first level of data.

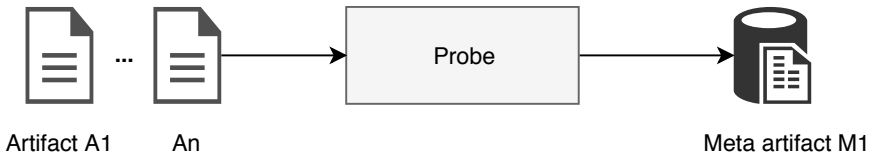


Figure 5.5: Probe activity

As an activity is a basic unit of work of the delivery system, RTO services must be realized as an activity as well. Table 5.1 provides an overview of the classification of RTO services as activities, together with their input and output. The test discovery and RTO optimization are classified as transformations. The test discovery takes a compiled code and transforms it into a test suite (basically a list of test classes). Similarly, the RTO optimization transforms an input test suite into an optimized test suite. The test run is classified as an assessment, similar to that of normal testing. It takes compiled code, together with an optimized test suite, performs the regression testing, and in the end outputs a test report. The RTO data collection, as described earlier, is classified as a probe. It simply takes an optimization data and probes for a partial state of data of interest.

Command	**Type**	**Input**	**Output**
Test Discovery	Transformation	Compilation	Test suite
Test Run	Assessment	Compilation, test suite	Test report
RTO Optimization	Transformation	Test suite	Test suite
RTO Data Collection	Probe	Optimization data	Probed data

Table 5.1: Classification of RTO services

5.2.2 Context Awareness

We clearly established in the previous section the concept of realizing RTO services as activities where each activity can be used to compose the delivery process by means of the delivery model. Nonetheless, we still need to address the issue of storing and retrieving the optimization data needed to run the RTO optimization itself. We know that there are different types of data as mentioned in section 2.3). Regardless of the types, in the context of the delivery system, depending on the data source used, the optimization data must be distinguishable by its delivery model identifier in order for the data retrieval to be able to collect the proper optimization data. A simple scenario for this is when we need historical test data information e.g. test execution time. The data needs to contain the information about by whom and where they are produced as depicted in Figure 5.6.

With that foundation in mind, the concept of being context (or history) aware is then needed to support the RTO adaptation. We refer to the context awareness as the ability for the activities within the delivery process to identify themselves on each delivery process execution by some sort of identifier. Previously, the concept of context awareness only existed within the orchestrator and modeler of Jarvis in the delivery process management layer (see subsection 2.5.2). However, the activities are left out of the loop and are not aware of how and whom calls their services. Hence, this information

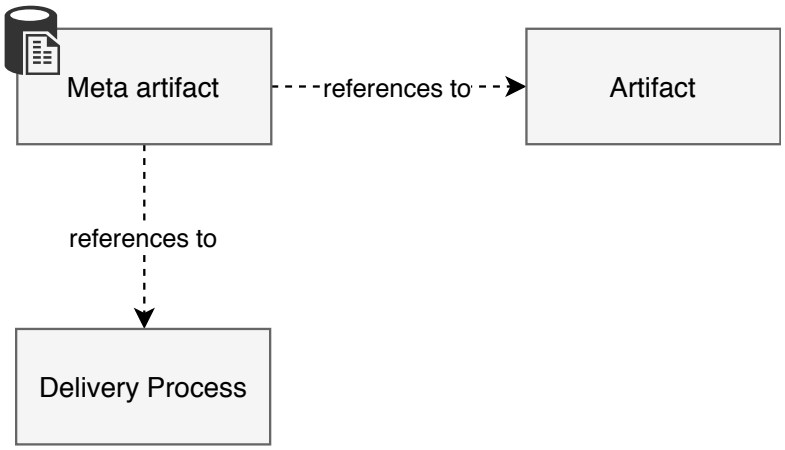


Figure 5.6: Meta artifact with context awareness concept

must be made available to the activities. Because a meta artifact is produced by a probe activity, this information can be provided via the probe activity. Thus, we can achieve the context awareness for the meta artifact by the principle of transitivity.

5.2.3 Delivery System Extension

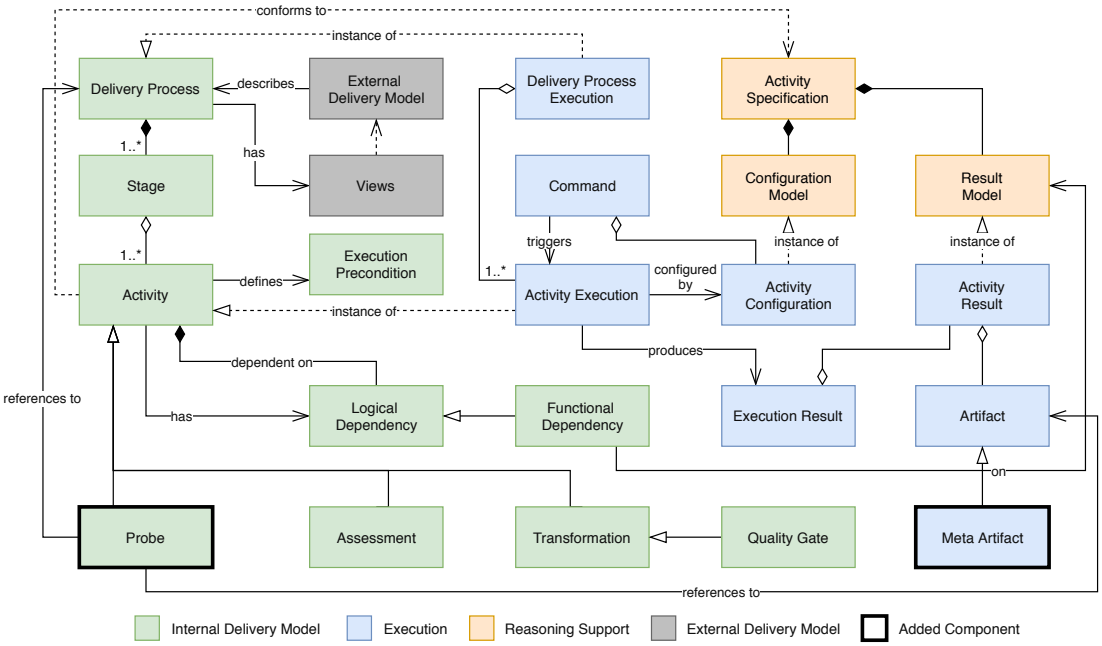


Figure 5.7: Updated delivery system core domain

With the new concept of probe activity and context awareness, the original delivery

system core domain must be extended. Figure 5.7 depicts the new delivery system core domain. The two added components, probe and meta artifact, are marked by the bold, black borders. Probe inherits from activity with a reference to both delivery process and the input artifact. It belongs to the internal delivery model area, similar to other types of activity. Meta artifact inherits from artifact. There is no direct reference to delivery process and the corresponding artifact because it indirectly references them via the probe activity that produces it.

With the extended delivery system core domain, we also updated the explanatory model to help ease the understanding of the domain with the added concepts, provided in Figure 5.8. With its focus on the building block of the delivery process, not all components from the core domain are depicted. The meta artifact is also excluded from this model because they are produced and stored internally within the probe activity and the outputs are not further used within the delivery process.

The first upper three levels belong to the original explanatory model. In short, at the core of this model, the transformation activities make up the main delivery process and are parameterized by the configuration from the upper level (from the delivery model). For the quality gates, the configuration usually contains a policy for the acceptance criteria The assessment activities evaluate the artifact, produced by transformations, based on some defined measurement. With the resulting report, the artifact is either promoted or rejected by the quality gate. With the additional concept for probe, the bottom level, *Probe Level*, is added to the model. The probe activities probe the artifact produced by a transformation or report produced by an assessment for some data and use that to produce meta artifact to store in some data source.

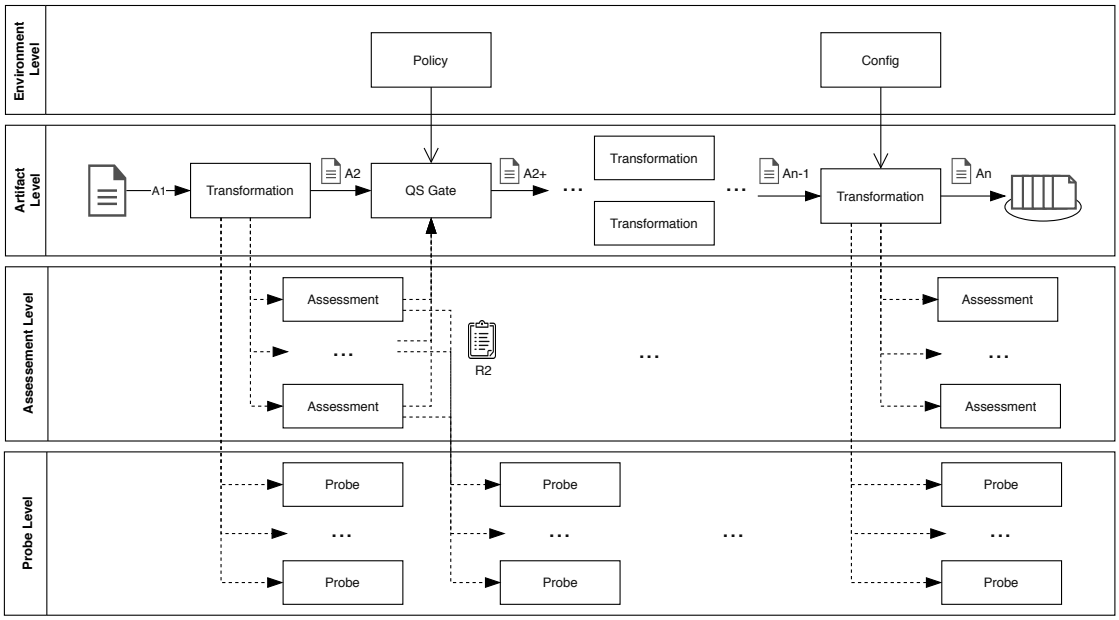


Figure 5.8: Updated explanatory model for core domain

5.2.4 Integration into Jarvis

It is important that the concept developed for the RTO adaptation can be integrated into Jarvis. Looking back at the architecture of Jarvis from Figure 2.4, we can clearly see that the RTO adaptation must be realized within the activity layer. Because we developed the domain model for RTO adaptation around the idea of microservices, the integration into Jarvis is quite straightforward. As we want to model the RTO process in the delivery model to be run as part of the delivery pipeline, they must be implemented in Jarvis as activities. We presented previously the concept for RTO services as activities and the extension to to the delivery system with a new type of activity and the context awareness. With these concepts, we do not need any further changes to integrate RTO into Jarvis.

5.3 RTO Adaptation Architecture

In this section, the complete architecture of RTO adaptation is presented. First, an overview of the adaptation is provided within the delivery system environment before going into detail about each component. We give a background information about activity, its classification, and an extension to the existing activity concept. We also discuss the RTO pipeline and its data flow, and finally, the architecture of data storage in this section.

5.3.1 Architecture Overview

The architecture of theRTO adaptation is established around a microservice pattern, which is the basic pattern of the delivery system, as depicted in Figure 5.9. Following the domain model defined in the previous section, the main operations of RTO established are extracted from a standalone, monolithic system into many smaller microservices. In essense, the RTO pipeline is moved from being run as a whole within an RTO system called by an API (see Figure 2.2) into its own individual service being run as part of a delivery pipeline instead. The blue components represent the added microservices necessary for the realization of RTO within the system. The overall architecture is very similar to the RTO domain diagram.

The *RTO test service* encapsulates the functionalities that require an interaction with actual tests from the test framework e.g. JUnit4 using a *test framework adapter* interface. This realizes the requirement FR2.5. The service includes *test discovery* and *test run*. The *test discovery* produces the original test suite and the *test run* executes the optimized test suite and produces the test report (see FR1.4) since all test frameworks differ in their actual implementation. The actual tasks of discovering test suites and running them are delegated to the adapter. These functionalities are the essence of any testing system. They can be used independently of any optimization to realize a simple testing pipeline. It is also important to note that though the microservice design pattern advocates directly the principle of the separation of concerns, the two functionalities are still realized in

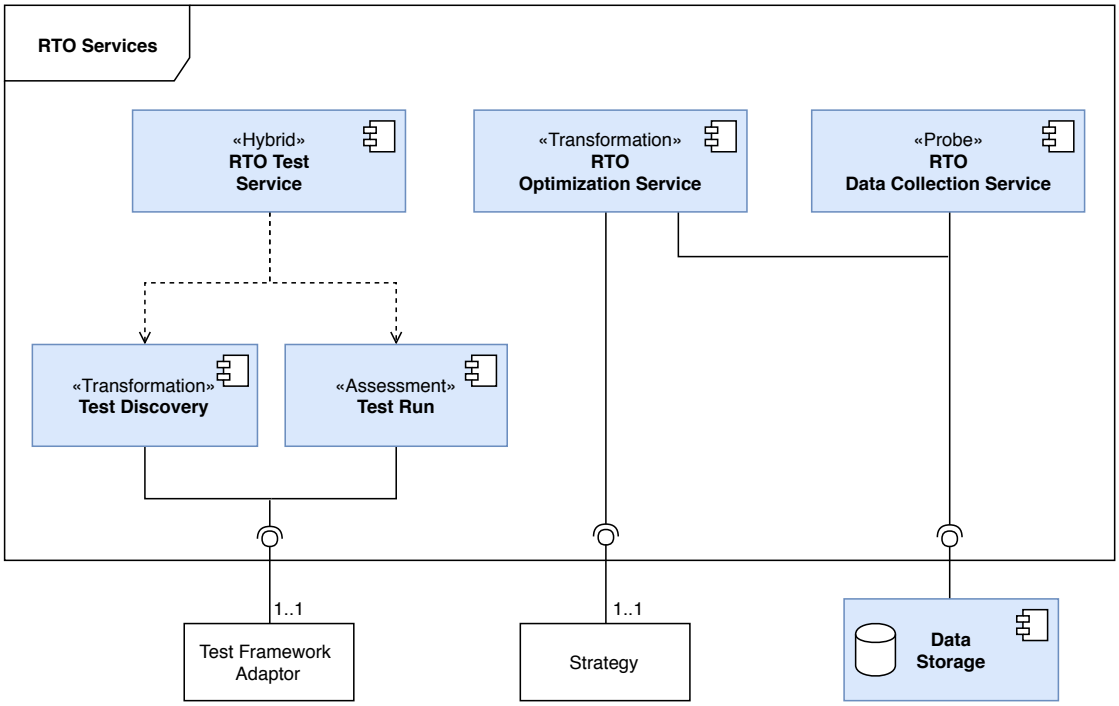


Figure 5.9: Architecture of RTO adaptation

the same service. The reason is that we want to encapsulate the interaction to the test framework within a single service, rather than having two separate services for each task.

The *RTO optimization service* is used to realize the optimization logic by means of strategy. A strategy can be any type of RTO technique, be it selection or prioritization. It takes a list of test classes and produces a list of optimized test classes. By the separation of concerns principle, a single service should encapsulate only a single strategy. To realize a new strategy, a new optimization service should be implemented. This aims to tackle requirement FR2.1 where a new optimization strategy can be easily integrated to the existing system. This also enables them to be combined in the delivery process to form a sequential optimization techniques (see FR1.2).

Another important component is the *RTO data collection service*. While the data collection task is a generalized task that does not belong only to the RTO domain, it still needs to be included in order to realize the complete RTO process. The *RTO data collection service* collects optimization data and stores them via *data storage*. The service is usually used after the test run (e.g. test history) or before the optimization (e.g. test coverage data). The *data storage* provides the interface to store and retrieve data from a particular data source. A new data source can be realized and configured under this service. The *RTO optimization service* also uses this interface to retrieve data to run the optimization. This realizes both the requirements FR2.2 and FR2.3. By providing each task of the RTO process as a service, we ultimately tackle the requirement FR1.1 and FR1.2.

Figure 5.10 shows how the RTO adaptation fits within the Jarvis architecture. All services lie in the activity layer, except for the data storage service which lies in the infrastructure layer as it belongs to a generic domain. This is described in subsection 5.3.3. The rest of the services stay the same as the original Jarvis architecture in Figure 2.4.

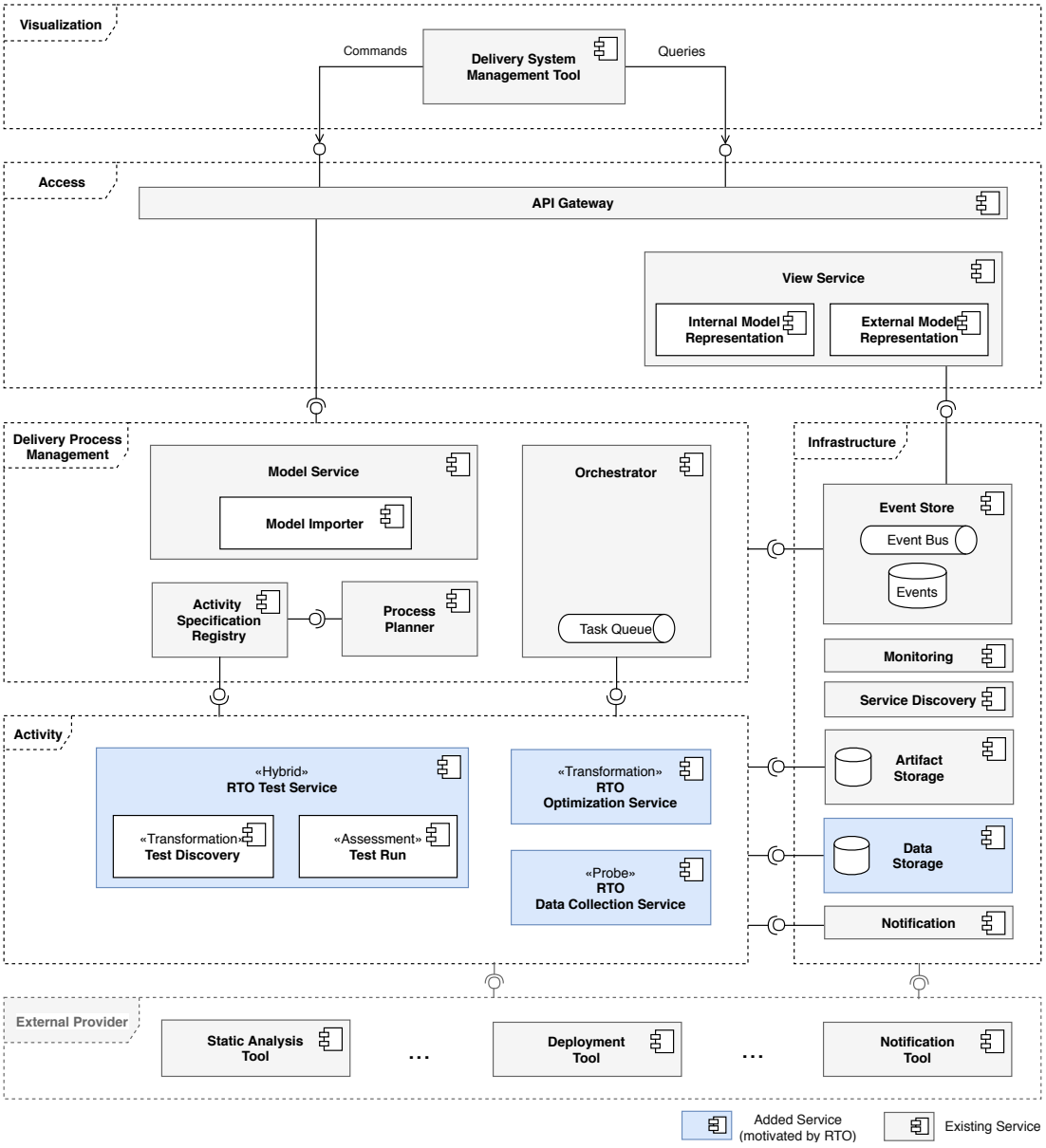


Figure 5.10: Complete architecture of Javis with RTO adaptation

5.3.2 RTO Pipeline

This section discusses the RTO pipeline within the delivery system. As each task in the RTO process takes the form of an activity, the RTO pipeline is essentially moved from within the RTO system to the delivery pipeline. All the RTO activities are independent of one another due to the microservice architecture in which all interactions between components are limited to the offered interfaces. Hence, the task of executing the activities sequentially is delegated to the framework (the orchestrator) of the delivery system. We do not go into the details of the pipeline modeler and orchestrator of the delivery system as it is out of scope of the thesis. Figure 5.11 visualizes the general work flow of the RTO pipeline without any data collection services. The artifacts are transfered from one service to another via *artifact storage*. An artifact storage, as its name suggests, is a storage for intermediate artifacts during the execution of the delivery pipeline. This component is omitted from Figure 5.9 because it belongs to the infrastructure layer of the system and is generally used for in all contexts, not limited to RTO.

As data collection depends greatly on the type of RTO techniques used for the optimization, it might not even be used at all in some cases e.g. *alphabetical prioritization*, etc. Nonetheless, data collection is mainly needed in two scenarios, either pre-optimization or post-execution. Figure 5.12 shows an example of each scenario. Depending on where in the delivery pipeline the optimization data is produced, data collection may happen earlier in the delivery pipeline, even before the start of the RTO process e.g. code coverage during compilation. On the other hand, the post-execution data collection is usually strictly limited to collecting the current test run related information e.g. test result, test execution time, etc. These data are then retrieved for optimization during the the next run of the delivery pipeline in the RTO optimization service, forming the concept for test historical data.

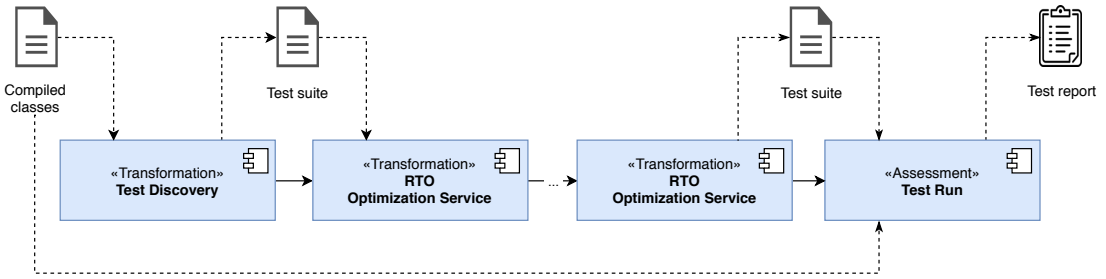


Figure 5.11: General RTO activity pipeline, excluding RTO data collection services

5.3.3 Data Storage

In subsection 2.5.1, we described a concept of a data store within the context of an RTO system [Ple15]. In short, a data store provides data necessary to perform an optimization on a test suite. Previously, in the monolithic prototype, the dependency loader is used to match a strategy with its corresponding data stores. Different strategies can share a data

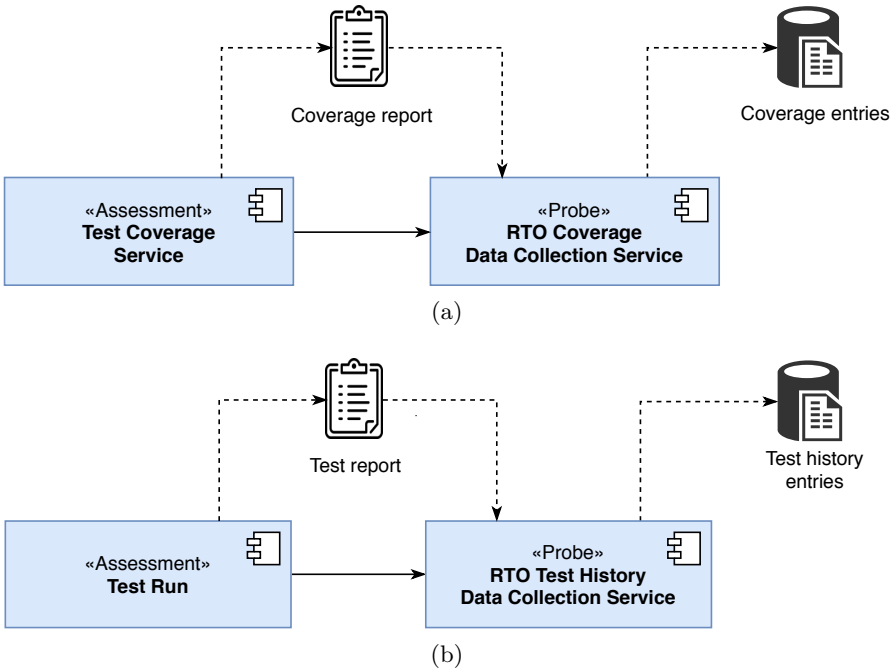


Figure 5.12: Examples of the RTO data collection service workflow. (a) shows the pre-optimization scenario and (b) shows the post-execution scenario.

store and multiple data stores can be used within a strategy. However, as we transform from a monolithic system to microservices, this concept contradicts the loose coupling principle which is one of the trademark characteristics of the microservice architecture. In order to solve this challenge, we take a look at a similar component in the infrastructure layer of the CD system. Artifact storage is a service within the infrastructure layer of the system. We first mentioned the artifact storage in the previous section as an intermediate storage for artifacts transfer between the activities. With its interface, an activity can upload and download targeted artifacts needed to be passed on or used within the activity itself.

Data Storage as a Service

Similar to the artifact storage, we can conceptualize another component offering storage services. Because the usages are different to that of the artifact storage, it cannot be reused. Therefore, we introduce a new component called *data storage*. Contrary to the artifact storage, the data storage is used as a permanent data storage, not for temporary data transfers. Hence it contains context-aware data, meaning they are aware of where and when they are produced. Though its initial purpose is to tackle problem within the RTO domain, data persistence is a very generic domain and can therefore be reused for other purposes as well. As depicted in Figure 5.10, the data storage also lies within the infrastructure layer of the system.

Integration of a New Data Source

Another important issue to tackle is that depending on the RTO techniques used, different types of optimization data are needed as discussed in section 2.3. It is important that the system allows an integration of a new data source (see FR2.4). This problem brings us back to the similar architectural concept introduced earlier in the chapter: monolithic and microservices. As data storage is provided as a service, there are two main approaches of extending them:

1. Expose a new data source using the same data storage service (monolithic)
2. Add a new data storage for a new data source (microservices)

The architecture of the framework allows us to do both with relatively minimal effort. A new data source e.g. relational database, time-series database, etc. can be added to the data storage we have conceptualized and be exposed using the same interface. This ultimately follows the concept of *Enterprise Information Integration (EII)* where all data is exposed via a single interface with a uniform representation. By choosing this option, we keep all accesses to the optimization data in one place. However, there are concerns over integration feasibility if many data sources are used and also the interface design to accommodate this integration. Another viable option is to add a new data storage service for each data source added. This follows the separation of concerns principle where each service only provides access to only its corresponding data source. Figure 5.13 illustrates these two options. Obviously, there is no right or wrong answer. This decision depends on the nature of the project, the heterogeneity of the optimization data used, and the preference of the developer.

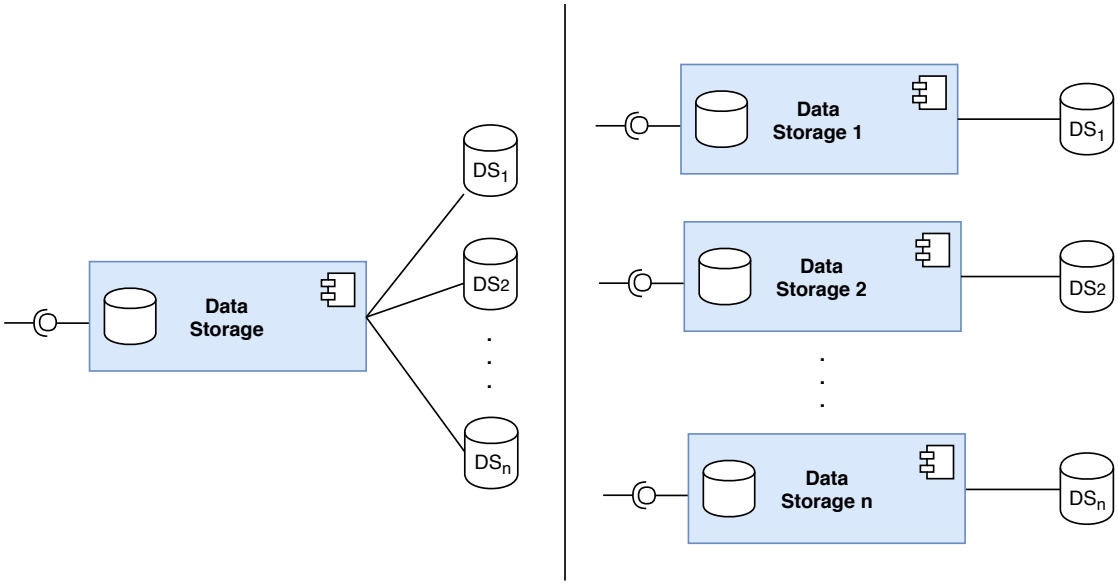


Figure 5.13: Two different approaches of extending the data storage. The left is monolithic and the right is microservices.

6 Realization

Contents

This chapter presents the implementation of the RTO adaptation in the Jarvis delivery system as described in chapter 5. First, the technologies used are outlined. Then, the detail of the implementation is presented following an iterative approach, together with the evolution of the architecture. Lastly, this chapter concludes with an exemplary delivery model which includes the complete RTO pipeline.

6.1 Technologies

The implementation utilizes several technologies. The choice of technologies is derived from the two existing systems used, Jarvis and Lazzer. The initial implementation of the RTO adaptation is taken directly from Lazzer, though it is refactored and decomposed during its course of development.

The implementation is written in the *Java* programming language with the Java test framework *JUnit4*. *Maven* is used for building the projects. All services implemented use *Spring Boot*. The implementation of data storage uses a *MySQL* database as a data source with *Hibernate* for data persistence in the database.

6.2 Iterative Development

The implementation of RTO adaptation follows an iterative approach as we decomposed the Lazzer framework into the Jarvis delivery system. We decided to perform iterative development so that we can refactor the code incrementally by extracting only part of the Lazzer framework at a time. By the end of the implementation, we finished a total

of five iterations. After each iteration, the tests are performed to validate that all the services still produce desirable results.

Iteration 1

At the beginning, we integrated Lazzer as a monolithic service. There is only one *RTO service* activity that runs the entire RTO process using the Lazzer framework API. In Figure 6.1, Lazzer refers to the entire Lazzer framework (see Figure 2.3 for the internal structure) and the RTO service acts simply as a client to the API. All the configurations must be passed to the RTO service i.e. list of strategies, test directories, system under test (SUT) directories, data stores, and test framework adapter. The entire RTO-related implementation still belongs within Lazzer, meaning that the RTO service has to reference all the modules as Maven dependencies to run Lazzer. The data sources used in the data storages have to be made as a prerequisite for running the service. Moreover, any new strategies and data storages have to be added to Lazzer instead of the RTO service itself. Nevertheless, as a result of iteration 1, we had a running RTO service with a simple `run` command to execute RTO.

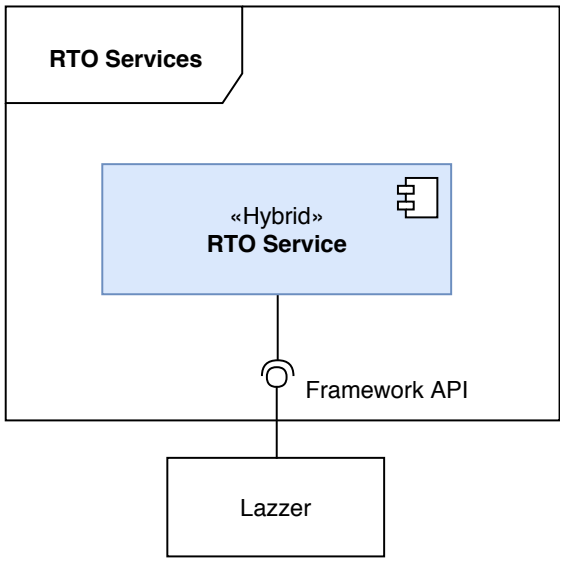


Figure 6.1: RTO architecture after iteration 1

Iteration 2

In iteration 2, we attempted to move the RTO pipeline from within the Lazzer framework into the RTO service activity. This is the first step of our refactoring toward microservices. As shown in Figure 6.2, we divided RTO service activity into two scopes: test-related and optimization-related service. The first deals with test-related tasks, including `test discovery` and `test run` commands, by delegating the actual execution of these tasks

to the *test framework adapter*. The second, *RTO optimization*, deals with optimization-related tasks using the `optimize` command. This service has a 1-to-n relationship with *Strategy*, meaning that we can run any or multiple strategies (given that we provide it with their corresponding *data storages* as parameters) via this command. The problem with this optimization service is that Lazzer previously used a *dependency loader* to load corresponding data storages for a strategy. By breaking down the framework itself, we can no longer use this along with the dependency injection as the service depends on a static dependency to be available during the compile time i.e. Maven dependencies for all known strategies and data storages must be added to the `pom.xml` file. By the end of this iteration, Lazzer is subdivided into test- and optimization-related modules, no longer acting as a single unit of work. By subdividing of the functionality within Lazzer, we can clearly separate the concerns between each command.

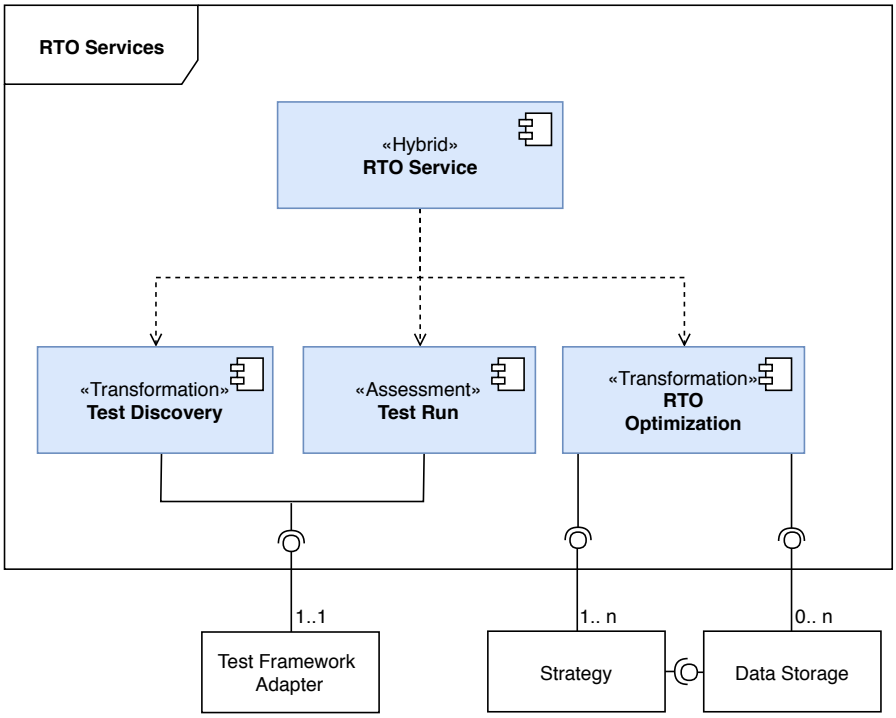


Figure 6.2: RTO architecture after iteration 2

Iteration 3

In iteration 3, we finally broke down the RTO service into separate microservices. One of the main reasons for doing so is the size of the RTO service which is too big which is in itself a bad architectural smell especially for microservice architecture. The *RTO test service* remains for all test-related services. Because of the static dependency problem from the previous iteration, we decided to make the RTO optimization into its own activity on a per-strategy basis, meaning that each optimization service only implements

one strategy. By having one strategy for each service, we know exactly what data storages it depends on. Consequently, we can remove the dependency loader altogether. This also gives us a compliance to the separation of concerns principle. As one service only contains one functionality or in the case of RTO test service, one single entry point for accessing test framework adapter. In spite of that, strategy and data storage remained a very tightly coupled concept which is not desirable because this means that in order to run the service, we always depend on the data source used by the data storage to be up and available.

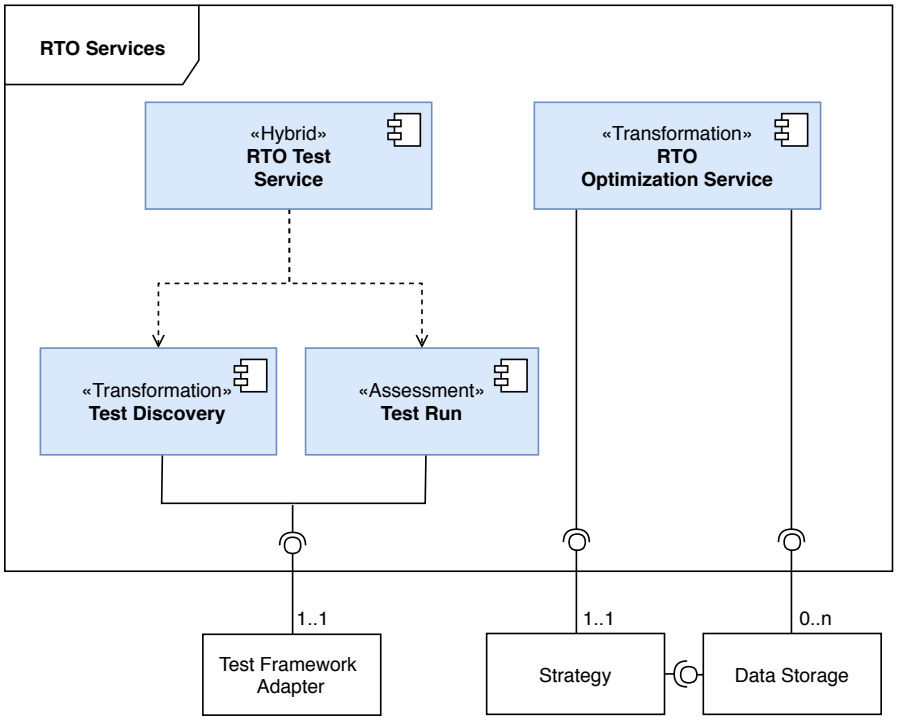


Figure 6.3: RTO architecture after iteration 3

Iteration 4

In iteration 4, another big change was introduced into the implementation. Though we solved the static dependency problem for the optimization service, there was still a problem in the scenario where optimization data is to be collected and stored into the data storage after the test run because the RTO test service does not have any notion of the data storage. Therefore, a concept for *RTO data collection service* was developed to tackle this problem. This led to the introduction of probe activity. The data collection service simply takes input artifacts and stores them in the designated data storage. This can be performed at any stage of the delivery process where the optimization data becomes available. As a result, we could move the pre-optimization and post-execution data collection from the RTO pipeline in Lazzer to the delivery system. Nonetheless, the

issue still remained that the data storage is then very tightly coupled to the optimization service and data collection service as we used a built-in database to first provide the proof of concept.

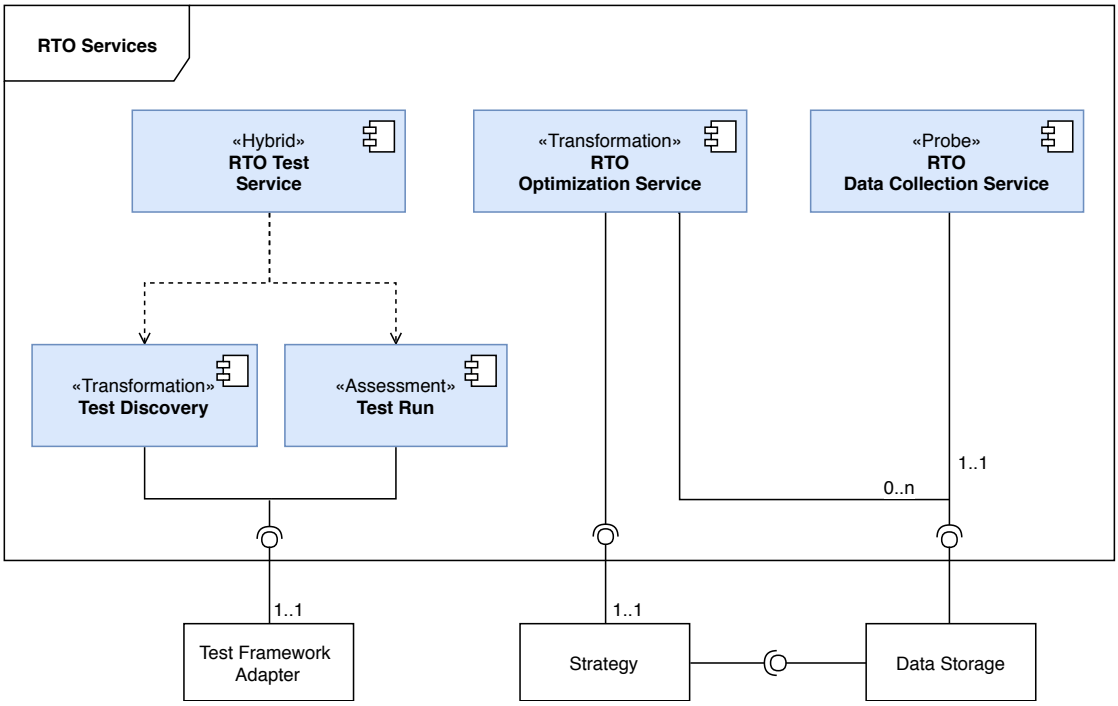


Figure 6.4: RTO architecture after iteration 4

Iteration 5

Iteration 5 is the last development iteration. This iteration results in the final architecture depicted in Figure 5.9 from the previous chapter. In this iteration, we attempted to decouple the data storage from other services completely as such behavior is not desirable within the microservice architecture. Moreover, it is very difficult to maintain the automated deployment when many services use to the same data storage. Hence, we decided to implement the data storage as its own data management service. The services that need access to the data storage to store or retrieve data can do so via the exposed API of the data storage service. We also previously discussed the concept of how to scale up or out to use multiple data sources in subsection 5.3.3.

As a result after five development iterations, we completely moved the RTO pipeline entirely from Lazzer to the delivery system. This allows us to flexibly add new optimization services (with new strategy) and data storages.

Table 6.1 summarizes where in the system the main Lazzer artifacts reside after the RTO adaptation. Lazzer in the figure now refers to the refactored Lazzer library at the end of the last iteration. The only artifact still belongs completely within Lazzer is the

Artifact	Lazzer	Jarvis	
		Class	Service
Test Framework Adapter	X		
Strategy		X	
Data Store (Storage)			X
RTO Pipeline			X

Table 6.1: Matrix for Lazzer artifacts after RTO adaptation

test framework adapter as we have no reason to extract it to Jarvis. Test framework adapter is used by RTO test service as seen in Figure 6.4 via its interface. On the other hand, the strategy is moved to Jarvis side. It is implemented within each RTO optimization service and is then used by the service to perform the optimization. Data storage, as mentioned earlier, is now its own service within Jarvis infrastructure layer. The RTO pipeline itself from test discovery, data collection, optimization, and test run is implemented as individual services and can be composed as a delivery process by using a delivery model.

6.3 RTO Activities

After looking into an overall development of RTO adaptation, we present in this section the detailed implementation of each RTO service. As mentioned in previously, the RTO services are implemented as activities within Jarvis using Spring Boot, but we do not go into detail about Spring Boot application. First, we provide an overview of RTO activities in general. To implement these activities, the dependency jars that need to be included can be seen in Source Code 6.1 (using Maven). The first three dependencies are necessary for Spring framework. The fourth one is the common library from Jarvis framework to access utility classes. The last one is needed to implement a service as an activity required by Jarvis framework. These are included for all RTO activities. Nonetheless, there are other different dependencies for RTO libraries that are included depending on the service that is implemented. All the activities implemented also follow the same class structure as seen in Figure 6.5. The class diagram shows a simplified version as each activity is slightly different in the internal inheritance structure, for example RTO test service has an abstract class that implements the `Command` interface and the commands simply inherit from this abstract class. In general, the service is provided as a `RTOCommand`. An activity can contain infinite numbers of commands. Each command is associated with `RTOCmdProperties` and `RTOCmdResult` in which the inputs and the outputs for the commands are defined. When the activity is deployed, the command is then provided with parameters and returns the result in Json format. The business logic of the activity is then implemented within a `RTOCommandExecutor` called from the command itself, which extends `SimpleCommandExecutor`. In our case, the executor is where all the RTO logic is placed.

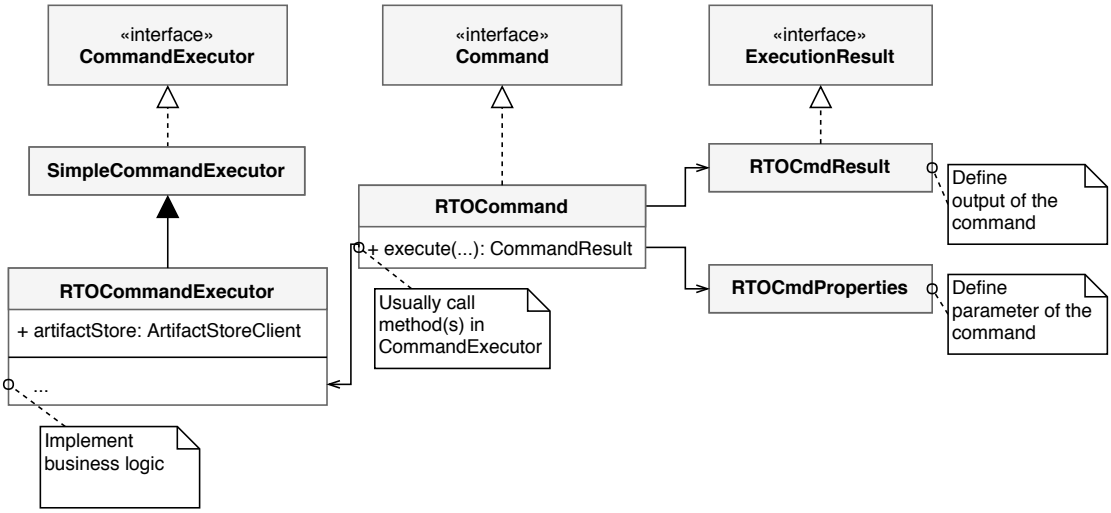


Figure 6.5: Simplified class diagram depicting the internal structure of RTO activity (Spring Boot framework not included)

```
    <dependencies>
    ...
    <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>

    <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>

    <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <dependency>
    <groupId>doering.thesis</groupId>
    <artifactId>common</artifactId>
    </dependency>

    <dependency>
    <groupId>doering.thesis</groupId>
    <artifactId>pipeline-service-starter</artifactId>
    </dependency>
    ...
```

```
28 |     </dependencies>
```

Source Code 6.1: Maven dependency for RTO activities

The RTO activities use RTO framework library which is adapted from Lazzer to implement their business logic. It is composed of six modules (see Figure 6.6). Out of the five modules, only three are added as dependencies in the RTO activities; namely `rto-entities`, `rto-test-execution`, and `junit4-adapter`. `rto-entities` contains the entity classes used as wrappers (avoiding serialization purpose) for the inputs and outputs e.g. `RTOTestClass`, `RTOTestMethod`, etc. They are used by all RTO services. `rto-test-execution` contains all logics relating to the RTO process. This is used only by RTO test service. Lastly, `rto-test-framework-adapters` contains all implementations for a specific test framework. In this thesis, we use `junit4-adapter` which is an implementation of JUnit4. It is also only used by RTO test service. The other modules *rto-core* and *rto-utils* are the internal implementation for the other modules.

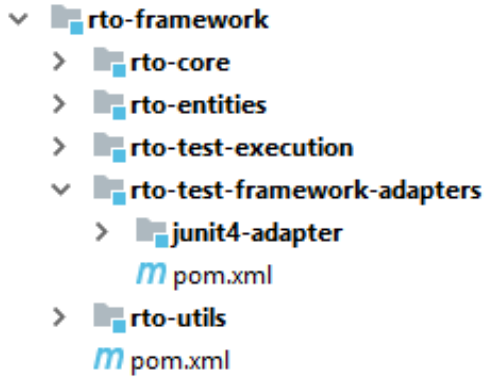


Figure 6.6: RTO framework (library) modules

6.3.1 RTO Test Services

The RTO test service is an activity that contains two commands: test discovery and test run. They operate independently of one another but share the same executor `RTOTestExecutor`. As mentioned in subsection 5.3.2, activities pass along their artifacts (inputs and ouputs) using artifact storage. Therefore, both commands first pre-process the input and get the corresponding artifacts from the artifact storage and post-process the output and store in the artifact storage afterward.

The RTO test service uses `rto-test-execution` and `junit4-adapter` in order to implement the functionalities for the commands. Below we give an explanation about each command.

Test Discovery

`RTOTestDiscoverCmd` implements a transformation command which transforms exe-

cutable source code into a test suite. The command itself calls a method `discoverTests` in `RTOTestExecutor` class which uses the RTO library module `rto-test-execution` to implement test discovery. A specific test framework adapter `junit4-adapter` is used and configured in `createSettingsBuilder`. In the code snippet in Source Code 6.2, we omit the pre- and post-processing of input and output data as well as the setting builder part because the details about them are negligible in our scope of work.

Parameters

- `classes` - an artifact reference to the target folder (containing compile classes)

Result

- `testClasses` - an artifact reference to the discovered test suite (a list of test classes)

```
1   //RTOTestDiscoverCmd.java
2   @Override
3   public RTOTestDiscoverResult
        executeInWorkspace(ExecutionMonitor meta, List<String>
        testClassPaths, List<String> sUTClassPaths) {
4       ...
5
6       Optional<TestSuite> testSuite =
            executor.discoverTests(testClassPaths, sUTClassPaths);
7
8       ...
9        return new RTOTestDiscoverResult(testRef);
10  }
11
12  //RTOTestExecutor.java
13  public Optional<TestSuite> discoverTests(List<String>
        testClassPaths, List<String> sUTPaths) {
14      Lazzer lazzer =
15          LazzerFactory.createLazzer(
16              createSettingsBuilder(testClassPaths,
                    sUTPaths).build());
17      Optional<TestSuite> testSuite = lazzer.discoverTests();
18      return testSuite;
19  }
```

Source Code 6.2: Test Discovery

Test Run

Test run (or sometimes called test execute) is an assessment command which runs the test and returns a test report and assessment metrics regarding the test result. In Source

Code 6.3, we omit the command part as it is the same as the one in Source Code 6.2. The command calls `executeTests` method in `RTOTestExecutor` class which uses to the same library to run the test and get a test report as a result. Other implementation details are very similar to the implementation from the test discovery above.

Parameters

- `classes` - an artifact reference to the target folder (containing compile classes)
- `testOrder` - an artifact reference to the optimized test suite (a list of test classes)

Result

- `testReport` - an artifact reference to the test report
- `passedRate` - a passed rate of the test

```
1    //RTOTestExecutor.java
2    public TestRunReport executeTests(List<String> testClassPaths,
         List<String> sUTPaths, List<RTOTestClass> testOrder) {
3        ...
4        TestRunReport testRunReport =
             lazzer.executeTests(Optional.of(testSuite));
5        return testRunReport;
6    }
```

Source Code 6.3: Test Run

6.3.2 RTO Optimization Service

RTO Optimization service is an activity that encapsulates the test optimization logic. In general, a service should only implement a single technique. An optimization service usually consists of a single assessment command to optimize the test suite and output the optimized test suite. This service only uses `rto-entities` for RTO datatypes, but not any other modules as all the techniques are implemented within the service. In this thesis, we implemented a few services utilizing history based techniques (HB). All of them need an access to the data storage that contains test history data. Hence, we implemented an intermediate class that inherits from `SimpleCommandExecutor` called `SimpleRTOTestHistoryCommandExecutor` that contains an implementation for storing and retrieving data from the data storage (see Source Code 6.4).

```
1    public class SimpleRTOTestHistoryCommandExecutor extends
         SimpleCommandExecutor {
2        private final RTODataStoreClient dataStore;
3
4        public int storeReport(String modelId, RTOTestRunReport
             report) {
```

```
5               return dataStore.store(modelId, report).getId();
6           }
7
8           public List<RTOTestResult>
                retrieveLastTestResultsFor(String modelId) {
9               ResponseEntity<List<RTOTestResult>> testResults =
                    dataStore.retrieveLastTestResults(modelId);
10              return testResults.getBody();
11          }
12
13          ...
14      }
```

Source Code 6.4: SimpleRTOTestHistoryCommandExecutor class

Below we give an example of how a typical RTO optimization service looks like.

Example: RTO Test Execution Time Prioritization Service

This service implements an RTP technique ordered by a test execution time. Source Code 6.5 depicts the main method that contains the optimization logic. It uses a class `TestExecutionTimePrioritization` which implements the test execution time prioritization strategy. Any strategy used in the RTO optimization service implements an interface `OptimizationStrategy`.

Parameters

- `testRef` - an artifact reference to the test suite to be optimized (a list of test classes)
- `modelId` - the running delivery model ID. This is needed to solve the context awareness issue (see subsection 6.5.2)

Result

- `testRef` - an artifact reference to the optimized test suite

```
1       //RTOOptimizeTestExecutionTimePriExecutor.java
2       public TestSuite
            optimize(RTOOptimizeTestExecutionTimePrioritizationCmd
            rtoCmd, List<RTOTestClass> testClasses, ExecutionMonitor
            monitor) {
3
4           //Create test suite from test classes
5           TestSuite testSuite =
6               RTOTestClass.createTestSuiteFromRTOTestClasses(testClasses);
7
8           //Retrieve test results from data store
```

```
9           List<RTOTestResult> fromTestHistory =
                retrieveLastTestResultsFor(
10                  rtoCmd.getProperties().getModelId());
11          if(fromTestHistory == null || fromTestHistory.isEmpty()) {
12              return testSuite;
13          }
14
15          else {
16              //Create strategy and Optimize
17              TestExecutionTimePrioritization strategy = new
                    TestExecutionTimePrioritization(fromTestHistory);
18              TestSuite optimizedTestSuite =
                    strategy.optimize(testSuite);
19              return optimizedTestSuite;
20          }
21      }
```

Source Code 6.5: Test Execution Time Prioritization

6.3.3 RTO Data Collection Service

RTO data collection service is a probe activity, containing a `probe` command that probes the inputs for only interesting data and stores them in the data storage. It follows the same class structure as in Figure 6.5. However, it does not actually contain any complicated logic for transformation or calculation. As mentioned previously, RTO typically needs to collect data in two scenarios: pre-optimization and post-execution. For this reason, we introduced a new probe activity concept, which is a new type of activity used for collecting different levels of data. The implementation details can be found in subsection 6.5.1. Below, we provide an example of a data collection service for the test history data.

Example: RTO Test History Data Collection Service

As the example above for the optimization needs test history data for its optimization, RTO test history data collection service is implemented as an activity for post-execution test history data collection to be used in the next delivery pipeline run. It contains a single probe command to probe for data. The executor extends the same base class `SimpleRTOTestHistoryCommandExecutor` as the example from RTO optimization services because it needs the same access to store data in the data storage for the test history data. As seen in Source Code 6.6, the service gets the test report, stores it in the data storage and returns the report id, but it is not used anywhere in the RTO process. Hence, we omit it from the list of results.

Parameters

- `testReport` - an artifact reference to the test report (from Test Run)

- `modelId` - the running delivery model ID.

```
1   //RTOTestHistoryProbeCmd.java
2   public RTOTestHistoryProbeResult execute(ExecutionMonitor
        monitor) {
3       RTOTestRunReport report = downloadAndGetReportFromRef();
4       int id = executor.collectAndStoreTestHistory(this, report);
5       return new RTOTestHistoryProbeResult(id);
6   }
7
8   //RTOTestHistoryProbeExecutor.java
9   public int collectAndStoreTestHistory(RTOTestHistoryProbeCmd
        collectCmd, RTOTestRunReport report) {
10      return
            storeReport(collectCmd.getProperties().getModelId(),
            report);
11  }
```

Source Code 6.6: Test History Data Collection

6.4 Data Storage

Data Storage is introduced to replace a concept for data store in the original Lazzer prototype. In this thesis work, we implemented a data storage to support our implementation of the aforementioned RTO activities. Since we used an example of RTO optimization service and data collection service that utilizes test history as their optimization data, the data storage that we implemented is specifically used for accessing the data source for test history. Because we know the exact data that we need to store and retrieve, we decided to use a traditional relational database for this. Since Jarvis already uses a MySQL database, it is therefore also used here. The implementation uses Hibernate with JPA-specfic implementation for accessing, persisting, and managing the data entity between Java and the database itself. The `RestController` of the data storage service is then mapped to a `FeignClient` to be used by RTO activities in a similar manner as the artifact storage. The data storage also lies in the infrastructure layer of the delivery system. The concept for extending it is provided in subsection 5.3.3. We do not go into the implementation details here because it is not of interest of this thesis. The code snippet is provided in Source Code 6.7.

```
1   @FeignClient(name = "data-store", configuration =
        RTODataStoreClient.RTODataSToreClientConfig.class)
2   public interface RTODataStoreClient {
3       @RequestMapping(method = RequestMethod.POST, value =
            "/testrunreport/{modelId}")
4       RTODataStoreClient.UploadResponse
            store(@PathVariable("modelId") String modelId,
            @RequestBody RTOTestRunReport report);
```

```
5
6           @RequestMapping(method = RequestMethod.GET, value =
                "/testrunreport/last/{modelId}")
7           ResponseEntity<RTOTestRunReport>
                retrieveLast(@PathVariable("modelId") String modelId);
8
9           @RequestMapping(method = RequestMethod.GET, value =
                "/testrunreport/last/{modelId}/results")
10          ResponseEntity<List<RTOTestResult>>
                retrieveLastTestResults(@PathVariable("modelId") String
                modelId);
11
12          @RequestMapping(method = RequestMethod.DELETE, value =
                "/testrunreport/{modelId}")
13          void delete(@PathVariable("modelId") String modelId);
14      }
```

Source Code 6.7: Data Storage Client

6.5 Framework Extension

After presenting the implementation details related to the RTO process in the previous sections, we provide in this section the realization for concepts that require changes and extension within the Jarvis framework itself.

6.5.1 Probes

The main concept that is developed to support the RTO process is the probes. Probes are a new type of command (consequently its containing activity) in Jarvis with the logic to only probe for interesting data. In order to add this concept to Jarvis, we define a new `CommandType` named `PROBE` and add logic for the model planner to include probes.

Probes can take inputs from both transformations and assessments. Therefore, they are planned after both of them, but in a similar manner as that of the assessments with the quality gates. They do not need to be explicitly included in stages of the delivery model. At the moment, probes do not return any result. If they do, the results are not further used in planning.

In order to create command of type probe, users can simply use `CommandType.PROBE` to define the command. Source Code 6.8 shows an example of how to define probes.

```
1       @PipelineCommand(name="probe", type= CommandType.PROBE)
2       public class RTOTestHistoryProbeCmd implements
            Command<RTOTestHistoryProbeResult> {
3           ...
4       }
```

Source Code 6.8: Probe example

The delivery model is also extended to support probe activity. It can be added to the delivery model under `Probes:`. The syntax is shown in Source Code 6.9.

6.5.2 Context Awareness

Because the RTO process needs to distinguish each delivery pipeline run from each other, we need a concept of context awareness for the activity (see subsection 5.2.2). Because this concept is not previously needed within Jarvis and not the main focus of our thesis, we decided to solve it with a simple solution by adding an alias for context information as part of the input parameters for the commands.

In order to use context information, the alias reference can be added to the delivery model as depicted in Source Code 6.9. There are two defined alias references:

- Model ID - `p://this/model/id`
- Model Name - `p://this/model/name`

6.6 RTO Pipeline: Exemplary Delivery Model

We have presented the realization of all the concepts in the previous sections. In this section, we provide an example of how the RTO pipeline can be modeled. A typical RTO pipeline usually follows a general pattern depicted in Figure 5.2. In the delivery model, the users can compose each step in the RTO process as a seperate activity, similar to Figure 5.11 and Figure 5.12 for data collection. It starts from test discovery, followed by a pre-optimization data collection (if needed). Nonetheless, a pre-optimization data collection activity can be modeled even earlier in the pipeline for example during compile time, depending on how and when the optimization data is created. Then, a list of opimization services are performed. The order in which they are added to the stages are important to the final optimization result since the optimization is performed in the defined order. Finally, the RTO pipeline ends at test run and post-execution data collection (if needed). Source Code 6.9 depicts a typical RTO pipeline that can be used in the delivery model.

At the moment, all parameters need to be provided manually in the delivery model. Smart planning is not implemented because of it being out of scope for this thesis. This topic is discussed further in subsection 7.4.2.

```
1  stages:
2    - name: buildTTGateway
3      transformations:
4        - checkout
5        - compile
6        - rto-dist
7        - rto-optimize-testtime
8        - rto-optimize-failedfirst
9
10 transformations:
```

```
11  - name: checkout
12    service: git-service
13    command: checkout
14    parameters:
15      repositoryUri:
            https://github.com/sungkangaw/rto-example-triangletester.git
16
17  - name: compile
18    service: maven-service
19    command: compile
20    dependsOn:
21      - alias: repo
22        ref: p://this/transformations/checkout/workspace
23    parameters:
24      workspace: "@repo"
25
26  - name: rto-dist
27    service: rto-test-service
28    command: discover
29    dependsOn:
30      - alias: compile
31        ref: p://this/transformations/compile/classes
32    parameters:
33      classes: "@compile"
34
35  - name: rto-optimize-testtime
36    service: rto-optimize-testexecutiontimeprioritization-service
37    command: optimize
38    dependsOn:
39      - alias: testclasses
40        ref: p://this/transformations/rto-dist/testClasses
41      - alias: model
42        ref: p://this/model/id
43    parameters:
44      testRef: "@testclasses"
45      modelId: "@model"
46
47  - name: rto-optimize-failedfirst
48    service: rto-optimize-failedfirstprioritization-service
49    command: optimize
50    dependsOn:
51      - alias: testclasses
52        ref: p://this/transformations/rto-optimize-testtime/testRef
53      - alias: model
54        ref: p://this/model/id
55    parameters:
56      testRef: "@testclasses"
57      modelId: "@model"
58
```

```
assessments:
  - name: rto-run
    service: rto-test-service
    command: execute
    dependsOn:
      - alias: compile
        ref: p://this/transformations/compile/classes
      - alias: testclasses
        ref:
           p://this/transformations/rto-optimize-failedfirst/testRef
    parameters:
      classes: "@compile"
      testOrder: "@testclasses"

probes:
  - name: rto-probe
    service: rto-test-history-probe-service
    command: probe
    dependsOn:
      - alias: testreport
        ref: p://this/assessments/rto-run/testReport
      - alias: model
        ref: p://this/model/id
    parameters:
      testReport: "@testreport"
      modelId: "@model"

qualityGates:
  - strategy: auto
    policies:
      - name: PassedTestRate
        interpretation: threshold-
        actualValue: passedRate
        setPoint: 0.9
        assessmentRef: p://this/assessments/rto-run
```

Source Code 6.9: Delivery model example

7 Evaluation

Contents

After establishing the concept in chapter 5 and the implementation of the aforementioned concept in chapter 6, we evaluate our work in this chapter. First, the RTO adaptation is evaluated against the requirements presented in section 3.3 and analyzed in terms of the software engineering quality. Then, we provide the comparison of the RTO adaptation between monolithic and microservice architecture, outlining the pros and cons of each. Lastly, the chapter ends with a further discussion.

7.1 Requirement Analysis

The following section discusses the implementation of RTO adaptation against the established requirements.

7.1.1 User's Requirements

To determine whether the realization of the RTO adaptation meets the user's requirements, this subsection provides an evaluation against user's requirements for the end users in Table 7.1 and for the developers in Table 7.2. Each requirement is determined whether it is fulfilled.

Req.	Evaluation
FR1.1	Fulfilled: yes The RTO services are provided in the form of activities and commands. They can be used to form a delivery model as shown in the example in section 6.6.
FR1.2	Fulfilled: yes Similar to the previous requirement, the users can choose to use any available RTO optimization services of choice or none at all in the given delivery model. The test suite is optimized in the order of the optimization services specified in the delivery model.
FR1.3	Fulfilled: yes Only the tests from the optimized test suite as an output of RTO techniques are run by the test run command. Likewise, all tests are run if RTO optimization services are not provided in the delivery model.
FR1.4	Fulfilled: yes The reference to the test report is provided as part of the result of the test run command. The actual report can be retrieved from the artifact storage, if needed, to be analyzed or further used i.e. via a command of an activity.
NR1.1	Fulfilled: yes The RTO pipeline modeled in the delivery model can be run successfully without any changes to the physical tests themselves. The underlying functionality of each RTO step has not been changed from that of Lazzer.

Table 7.1: Evaluation of the requirements from Table 3.1

7.1.2 Architectural Requirements

The following briefly discusses whether the changes and framework extension made to accommodate the RTO adaptation meets the original architectural requirements of Jarvis. Although this is not the main focus of this work, we must guarantee that the previously defined architectural requirements are still fulfilled regardless of any changes to the existing framework. The evaluation is provided in Table 7.3. Some of the requirements, namely AR3.5, AR3.6, and AR3.8, are omitted from the evaluation because they are out of context of the realization in this thesis.

Req.	Evaluation
FR2.1	Fulfilled: yes Developers can realize new RTO techniques by implementing a new optimization activity. The example is shown in subsection 6.3.2.
FR2.2	Fulfilled: yes This is divided into 2 parts. Developers can retrieve and store data using data storage client. • Developers can retrieve data using data storage client within their optimization activity. • Developers can implement probe activity similar to the one provided in subsection 6.3.3 to store optimization data using using data storage client as well.
FR2.3	Fulfilled: yes This is similar to the second point of the previous requirement. Developers can implement probe activity to collect and store data of choice.
FR2.4	Fulfilled: yes This is not provided directly by the RTO adaptation but delegated to the Jarvis framework itself. New data storage needs to be implemented as part of infrastructure layer of Jarvis. Section 5.3.3 provides an overview of how to include a new data storage depending on the type of data used: adding new data sources to the provided data storage or implementing multiple data storages. For the first option, developers can simply extend the current data storage.
FR2.5	Fulfilled: yes A new test framework integration can be realized in the RTO library by implementing the given API. This can then be used in the RTO test services. Currently only the implementation for JUnit4 is provided.

Table 7.2: Evaluation of the requirements from Table 3.2

7.2 Software Engineering Attribute Analysis

After evaluating the RTO adaptation in terms of its requirements, we look into it in regards to the software engineering attributes. Given the fact that our evaluation does not include a case study or an empirical study, we cannot make an objective evaluation of the RTO adaptation in actual practice. Nonetheless, we made a few observations on different aspects of the RTO adaptation. There are a few standards for measuring software quality and attributes. The most prominent one, ISO/IEC 25010 [ISO11],

Req.	Evaluation
AR3.1	**Integration of Heterogeneous Technology** In Jarvis, new technologies can be integrated by means of activity services. This practice stays the same for the RTO adaptation. New RTO techniques can be easily adapted/implemented as an activity service without any changes to the other part of the delivery system.
AR3.2	**Modularity** The Jarvis architecture is designed using microservice architecture. The same design principle is used to design the RTO adaptation. It is realized in a distributed manner with each RTO aspect as its own microservice, following the DDD concept. It provides flexibility and extensibility to further development of RTO e.g. adding new techniques, a new framework adapter, data collection services, etc.
AR3.3	**Activity Abstraction** The delivery process steps are encapsulated as activities with each activity representing a unit of work. The RTO activities can be referenced in the delivery model without knowing the internal implementation details. Moreover, as we introduce the probe activity, it can also be referenced in the same way in the delivery model. There is no change in regard to the information hiding principle, reusability, and composition.
AR3.4	**Self-Organizing** In the self-organizing context, the planner transforms the delivery model into the delivery process. This planning is not affected by RTO adaptation. However, as we have introduced probe activity, we make sure that the model planner can plan and complete the delivery model that includes one or more probe activities. Nonetheless, smart planning is not implemented for RTO. The issue is discussed in subsection 7.4.2.
AR3.6	**Model Validation** Because we extended the delivery process to support the probe activity and the context awareness concept, we made sure that the external model is still syntactically validated before converting to the internal delivery model. Furthermore, the process planner was also extended to validate the internal delivery model against these extra concepts.
AR3.7	**Best Practices** The self-organization process in Jarvis ensures that CD best practices are enforced. By adapting RTO as part of the delivery process, further CD best practices can be enforced. The RTO adaptation encourages best practices by introducing the benefits of RTO such as resource saving to the testing process. Also depending on the RTO techniques used, other principles can be enforced e.g. fail fast, etc. We consider this requirement to be met.

Table 7.3: Evaluation of the architectural requirements from Table 3.3

is a standard established by the *International Standard Organization (ISO)* and the *International Electrotechnical Commission (IEC)*. ISO/IEC 25010 defines on top of its predecessor ISO/IEC 9126 the product quality model, comprising of eight quality characteristics: functionality, performance efficiency, compatibility, usability, reliability, security maintainability, and portability. Other than these characteristics, an online resource by *Microsoft* [Mor07] also defines a few other software engineering attributes that are a desired characteristic for this work such as flexibility. We compile the list of relevant software engineering attributes and discuss the RTO adaptation according to each attribute in the subsequent subsections.

7.2.1 Functionality

Functionality, or functional suitability, is perhaps the most important software engineering attribute. This attribute comprises of functional completeness, correctness, and appropriateness. We tested the functionalities of the RTO adaptation by creating delivery models for the open-source projects. The models are similar to the one from Source Code 6.9, only replacing it with a different git repository. All the RTO techniques used are history-based techniques, utilizing test history data for the optimization. The first round of the delivery pipeline execution always runs all the tests because the optimization data is not yet available. The subsequent runs then use the optimization data. The planned delivery model and the delivery model execution are depicted in Figure 7.1.

Figure 7.1: Planned delivery model for Apache commons-csv

.

Although we know that all the requirements are met as we have discussed in the previous sections, there are nonetheless some limitations to the implementation.

Test Framework Usage

In regard to FR2.5, the current RTO test service activity which contains test discovery and test run commands currently only supports JUnit4. The reference to the dependency for the test framework adapter of JUnit4 is added statically in the `pom.xml` file for the Maven project. During the evaluation, we applied the RTO process in the delivery model for some open-source projects. However, we were not able to apply our work to some projects on the basis that they use other test frameworks. In order to support other test frameworks, the implementation of RTO test service should consider adding test framework as a parameter and dynamically added the dependency, if possible. A step further would be to add the test framework as part of the project information such that the service can automatically determine which test framework adapter to use.

Implementation of Data Storage

We presented the concept on how data storage can be extended in subsection 5.3.3 and the example data storage for test history data using a relational database in section 6.4. The idea is to delegate the work of data management and encapsulate it in the data storage service(s). Nonetheless, the concept is only proven for optimization data that is or can be stored using a relational database. In order to use other types of databases e.g. time series database, graph database, etc. to support other RTO techniques, the corresponding data storage services need to be implemented. This requires extending the delivery system framework with new services in its infrastructure layer. This would require the developers to possess knowledge regarding the internal workings of the framework itself which is not as easy as implementing a new activity for a new RTO optimization technique.

Context Awareness

As outlined in subsection 6.5.2, the implementation of context awareness is to provide context information as parameters in the delivery model. However, with this implementation, it is quite redundant when multiple activities are required to use the same parameters. In the example delivery model from Source Code 6.9, the context model ID is already used 3 times. Nonetheless, it is debatable how the concept of context awareness should be implemented whether it should be kept as parameters but perhaps automatically filled by the pipeline planner or the orchestrator without being outlined in the delivery model or the context information should be made available directly to the activity during the delivery process execution itself.

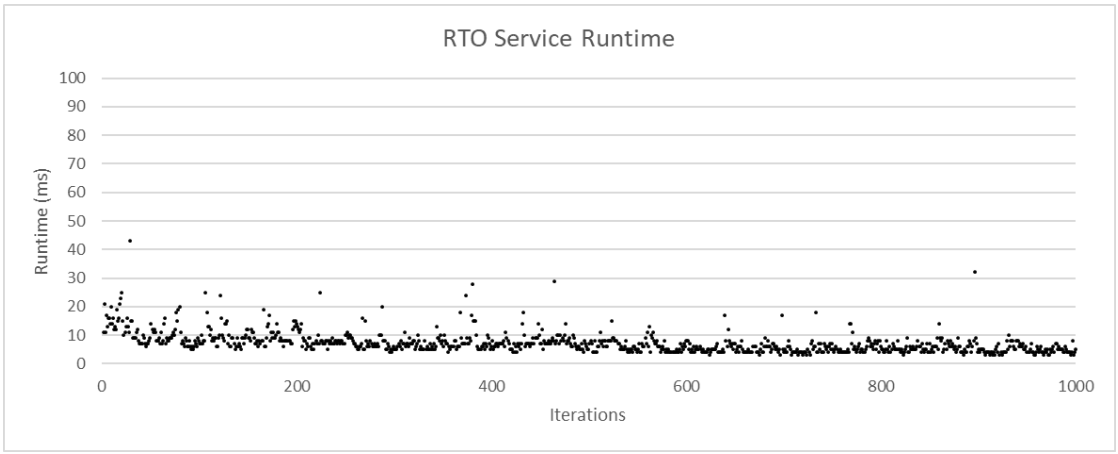
7.2.2 Performance Efficiency

Performance efficiency measures the performance relative to the resources used under defined conditions. Since the main purpose of adopting RTO is to save resources by automating the process, the RTO adaptation should not slow down the actual regression testing process. However, the performance of RTO depends largely on the RTO techniques and strategies used to run the optimization itself and the retrieval and storage of the optimization data. In addition, since we work with microservices, the network performance has to be factored in as well as the uploading and downloading of artifacts passing between the services. In order to measure the performance objectively, the evaluation needs to be independent from these factors.

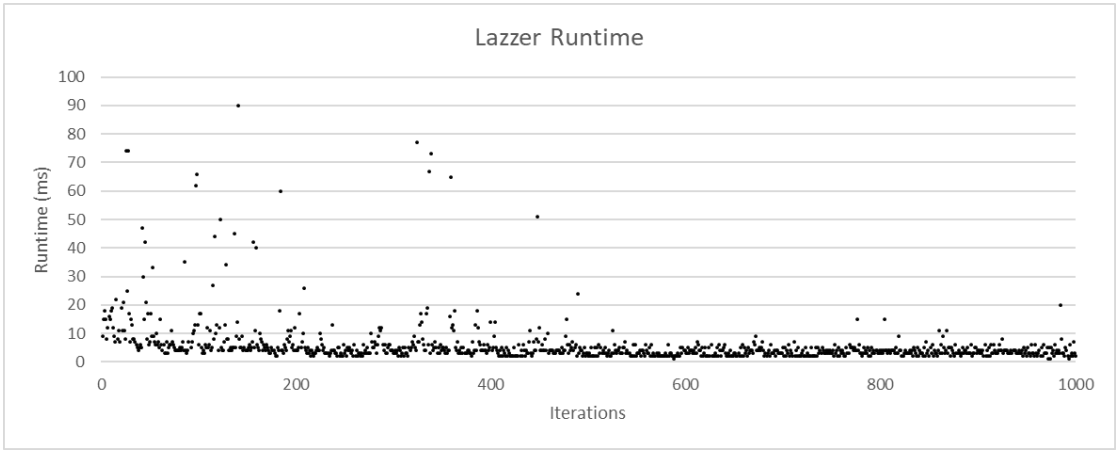
We followed the same evaluation approach as [Ple15] to measure the performance by using no strategies and data storages and replacing test framework adapter with the mock object. As a result, we wanted to obtain the bare minimum needed to run RTO. The evaluation is implemented as a JUnit test using the *Mockito* framework. The test is set up similarly to the one from [Ple15] with the runtime results written to a csv file. In fact, the result from the Lazzer performance test is used as a benchmark for our work. Furthermore, to remove the network factor, we access the `RTOTestExecutor` directly to run its `discoverTests` and `executeTests` methods. We measured the run times

for running both methods as well as the total runtime. In order to obtain comparable results, the performance test for Lazzer was rerun from the same machine as the RTO adaptation. The test was executed using Intellij IDEA 2018 1.2 (Community Edition) with JDK 1.8.0_151 on a Microsoft Windows 10 Pro machine with the following specs: Intel Core i7-5500U 2.40 GHz processer and 16 GB RAM.

The results from all 1000 iterations are shown in Figure 7.2. Furthermore, Table 7.4 shows the statistical data from the runtime results. Please note that we took out the first iteration when calculating statistical data where the objects are being initialized. As a result, we can see that the runtimes of both are comparable and in the same order of magnitude. From this, we can conclude that the change in architectures of the RTO adaptation has little to no effect on the performance of the basic RTO process itself.



(a)



(b)

Figure 7.2: Scatter plots over the runtime of 1000 executions of (a) RTO adaptation and (b) Lazzer performance test. The plots show similar results. Some outliers are out of range.

Runtime (ms)	Lazzer	RTO Adaptation
First iteration		
Test Discovery	-	793
Test Run	-	800
Total	707	1593
Average		
Test Discovery	-	1.24
Test Run	-	5.99
Total	6.28	7.23
Max/Min		
Test Discovery	-	9/<1
Test Run	-	41/2
Total	137/1	43/3
Median		
Test Discovery	-	1
Test Run	-	5
Total	4	6

Table 7.4: Result from 1000 iterations of the RTO adaptation performance test. Result from Lazzer is used as a benchmark. The statistical data are calculated without the first iteration and shown in milliseconds.

Furthermore, we also measured the performance of RTO adaptation as a whole process in the delivery model as well because this is how the RTO adaptation will be used by the end users. Due to a technical reason, the whole Jarvis system was run as docker containers on a separate virtual machine (Linux Ubuntu 16.04.2, 6 Intel Xeon CPU E5-2420 1.90GHz processors, 15 GiB RAM). Hence, the performance results cannot be compared to the previous results objectively. Additional factors such as communication overhead, network performance and parameter processing (to and from artifact storage) should be taken into account when reviewing this data. In order to achieve minimum model, we only used the test discovery and the test run service (including the quality gate by default) on an empty project. The planned model is shown in Figure 7.3 and the screenshot of the KPIs data after 100 successful pipeline executions in Figure 7.4.

Considering that for big projects, the actual runtime for executing RTO could be in the order of minutes, hours, or more. The runtime performance of the minimal execution of the RTO adaptation is considered negligible.

Figure 7.3: Minimal planned model used for running the test executions

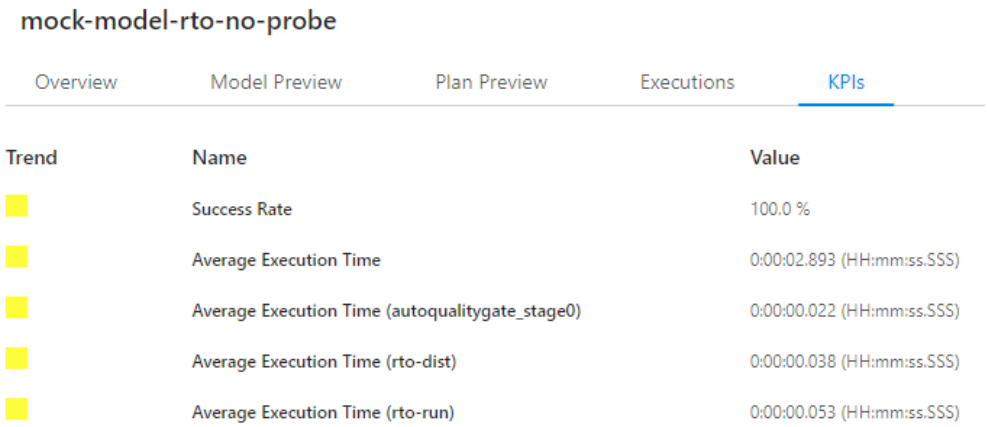


Figure 7.4: Screenshot of the KPIs data from 100 pipeline executions

7.2.3 Usability

Assessing the usability of the RTO adaptation is not an easy task. Normally, an empirical study or at least a case study should be conducted to gather this information. However, due to the limitation of time for this thesis, we are only able to provide our argument regarding the usability of this RTO adaptation.

In chapter 4, we provided various tools available for automated RTO. Although we cannot claim the list to be complete, we cannot find any open-source tools that provide the integration of RTO to be modeled directly in the delivery model. The usage of RTO requires either a manual implementation for tool integration or is limited to a single type of RTO techniques. In this regard, our work tackles both problems. Adapting RTO in the delivery process is now relatively easy by simply modeling the RTO process in the delivery model. Various RTO techniques can be used by implementing them as activities. The intermediate result of each step in the RTO process can be individually inspected. Moreover, our RTO adaptation can be utilized in various levels of testing, not limited to unit testing.

Nonetheless, because of the nature of microservice architecture, the implementation of RTO adaptation is distributed in smaller services. This results in an increase in complexity over the whole process. Instead of modeling a single activity in the delivery model, the end users now need to have a considerable amount of understanding of the RTO process itself in order to employ the RTO process in the delivery model.

7.2.4 Flexibility and Extensibility

Flexibility and extensibility are interrelated attributes. According to [Mor07], flexibility refers to the ease in which a system or component can be modified for use or application, other than the designed one. Extensibility is referred to as the ability to extend the system and the level of effort needed for it. As discussed in various places before, the

RTO adaptation is built following the microservice architecture with each component in the process as its own services. This provides us with a lot of room for flexibility and extensibility. New RTO techniques can be implemented as activities without any modifications to the other part of the RTO architecture as well as the framework itself. The process of optimization data collection can also be implemented as a probe activity, separating the concerns of producing the optimization data and its subsequent data collection. This is encapsulated by the architectural requirement AR3.1. Nonetheless, the trade-off for the flexibility and extensibility comes in the form of complexity, similar to the discussion in the previous subsection about usability.

7.2.5 Maintainability

Maintainability describes the extent in which the software can be efficiently maintained (fault correction, performance improvement, adapting to environmental changes, etc.). It encapsulates the concept of modularity, reusability, analysability, modifiability, and testability. Modularity and reusability refer to whether the software follows in a modular and reusable design. Analysability refers to the degree in which defects can be tracked, analysed, and understood. Modifiability is the ability in which software can be modified without introducing new defects or regressions. And testability describes the degree to which the software can be tested.

Modularity is already one of the main concerns of the delivery system framework, described by AR3.2. Therefore, the RTO adaptation was built with this attribute in mind. Reusability is also encouraged by dividing up the RTO library into smaller modules that can be included as dependency in the `pom.xml` file as needed. Analysability is provided in the form of logs. As each step in the RTO process is its own service, separate logs are provided. In a similar manner, modifiability is achieved because the effect of any modification is limited by the scope of the service itself. However, the scope of testing is limited to the scope of the service as well. At the moment, integration tests cannot be automated for the whole RTO pipeline in the delivery process.

7.2.6 Reliability

The evaluation of reliability is not an easy task. As we have not conducted a case study, the RTO adaptation is not adequate enough to be used in the production environment. More effort is needed to test the RTO adaptation as well as adopting new RTO techniques using various types of optimization data. However, because of its modularity and microservice architecture, each service can be easily recovered and made available again in case of failure independent of other services.

7.2.7 Compatibility

Compatibilty refers to the ability to co-exist and interoperate within different systems. The RTO adaptation realizes this by providing the ability to extend the implementation for different test frameworks (see FR2.5). Moreover, since each RTO process is implemented

as a service, it can be used by any system by accessing its REST interface. The compatibility of using the whole RTO process in the delivery model itself is the same of that of Jarvis as these services are added as part of the framework.

7.2.8 Portability

Portability can be evaluated in regard to three aspects: adaptability, installability, and replaceability.

The adaptability describes the extent in which the software can adapt to the new environments and the installability refers to how easy and efficiently it can be adopted or installed. The RTO adaptation itself does not concern these aspects. As it operates as part of the delivery system, it is as adaptable and as installable as its host system.

The replaceability refers to whether the software can be replaced by others with the same functionalities. The testing part (test discovery and run) of the RTO adaptation can be replaced by other testing services such as JMeter service. The users can still run the test with no optimization as part of the delivery pipeline. However, as a whole process, there is no other substitute.

7.2.9 Security

Security is not a concern for the development of the RTO adaptation. Any concern to the security is delegated to the delivery system itself.

7.3 Comparison between Architectures

The following section compares and discusses the differences between the two architectures of the RTO: monolith and microservices. Although the end architecture of the RTO adaptation uses microservice architecture, the initial implementation (see section 6.2) adapted Lazzer as a monolithic service.

7.3.1 Dependence on Data Sources

RTO uses data sources to manage the optimization data used by RTO techniques. Depending on the strategies used, data sources can be different e.g. relational database, time series database, etc. In the monolithic architecture where the whole RTO pipeline is located in a single service or activity, this means that the service itself either:

1. Has a direct connection to the data sources: the service connects directly to the data source(s) through the data storage object. This approach means that the service itself has to make sure that the connection to any data sources must be available. If there are multiple data sources, it can be difficult to maintain it as a microservice and also goes against the separation of concerns principle.

2. Connects via the data storage service(s): This approach delegates the work of data source management to the data storage service. This is considered a better

approach because the RTO service itself does not depend on the connection being available. However, as a monolithic service, the access to all the data storages used for all the RTO techniques must be made available within a single service.

On the other hand, in a microservice architecture which is the resulting architecture of this work, we eliminated the dependence on the data sources from the RTO services. Similar to 2., the work of data source management is delegated to the data storage service. The main difference is that only the RTO optimization and the the data collection service need an access to the data storage service. This means that if there are multiple data storages, the subscribing services know which service(s) they would need for retrieving and storing optimization data. As a result, even when the data storage services are down, the minimal RTO pipeline can still be executed i.e. the tests are still executed without the optimization part.

7.3.2 Collection of Optimization Data

The collection of optimization data is one of the important tasks in the RTO pipeline. As mentioned before, there are two scenarios where the data collection can take place: before the optimization and after the test execution. In the monolithic service, the RTO pipeline is fixed (see Figure 5.2). Since the logical order is implemented internally, the data collection can only take place at exactly these 2 positions, if it is needed. Nonetheless, despite the typical sequential pipeline presented, there might be other situations where the flexibility is needed to model the data collection process i.e. collecting data even before the RTO pipeline. For example, assuming we measure the code coverage data for the optimization. This can be executed during or after the compile time which is even before the test discovery is executed. This is when we can benefit from the microservice architecture. By shifting the RTO pipeline to the delivery system, we obtain the flexibility to model the RTO pipeline however we want. Thus, the optimization data can be collected anywhere, be it early on or even at the end of the delivery pipeline.

7.3.3 RTO Techniques and Strategies

The architecture of RTO adaptation plays quite an important role in the strategies that can be used for the optimization. The ability to freely model the RTO pipeline opens up to the implementation of a wider range of RTO techniques. Because we adopted the implementation from the original prototype of Lazzer which works based on the executables produced by the compilation process, the RTO techniques that can be implemented are limited to those which can be used on the executables and do not need access to the actual source code for collecting the optimization data.

From the bachelor thesis by Schade where the coverage based techniques (see subsection 2.3.1) are implemented, the coverage data generation needs to be triggered outside of the scope of Lazzer and the said data is then used during the optimization [Sch18]. This approach might work for a standalone system. However, as part of the delivery system, this arguably does not make sense and is impossible in practice because the monolithic

service does not concern anything outside of its service scope i.e. it cannot collect the data that is not part of the RTO pipeline. This is why a monolithic architecture of RTO does not fit within the delivery system. On the other hand, a microservice architecture distributes the tasks within the RTO process as services. This decouples the creation from the collection of the optimization data. In the delivery system, the creation and collection of data can be realized as separate services and executed anywhere within the delivery pipeline. As a result, other services in RTO do not need to be aware of this process, keeping the concerns separated as well as hiding the inner workings of the service. The same argument can be used for other categories of RTO techniques as well.

7.3.4 Model Complexity

One of the biggest concerns of implementing the RTO adaptation using microservice architecture is the model complexity. For the monolithic architecture, the RTO activity can simply be added singularly in the delivery model. In this case, the end users do not actually need to understand the internal workings of how the RTO is executed, but only what parameters to be used. Contrarily, the distributed nature of the RTO adaptation as microservices introduces the model complexity over the whole RTO process. As mentioned in subsection 7.2.3, to model the RTO process in the delivery model, the end users need to understand how RTO works in general. While they do not need to understand the implementation details within the RTO services, they still need the knowledge of how to order the sequential pipeline and what parameters need to be passed to what services. Thus, it can be said that the microservice architecture delegates from the internal complexity of the monolithic architecture to the external one.

7.3.5 Summary

We discussed in this section the comparison between the monolithic and the microservice architecture. The adaptation of RTO as microservices bring many benefits on the table, comparing to the same implementation as a monolithic service. The flexibility to model the RTO pipeline offers the possibility to work with more diverse RTO techniques as well as loosen the coupling on the data collection process. It also encourages many design principles such as the separation of concern, information hiding, and so on. Nonetheless, with the microservice architecture, we must make a trade-off in the model complexity. In summary, we believe that the benefits of implementing the RTO adaptation as microservices outweighs the monolithic one in the case of the delivery system.

7.4 Discussion

A lot has been discussed about the implementation of RTO adaptation regarding its architecture, requirements, and quality aspects in this chapter. In spite of that, there remains a few discussion topics that do not quite fit in any earlier sections. They are discussed in this section.

7.4.1 CD Maturity Model

One of the important principles of CD is to automate as much as possible. To this extent, our RTO adaptation makes the adoption of RTO in the delivery process easier by eliminating the manual tasks and offering the possibility of modeling the RTO process in the delivery pipeline. Given that the main goal of our thesis is to adapt RTO for CD, it is important to discuss how it contributes in this area. In order to consider the contribution of our work, we should first examine the role of RTO within the context of CD.

According to [PRB13], the CD maturity model is created as a guideline to give some sort of structure and understanding of how to adopt CD for one's organization. It includes the key aspects on the following areas: culture and organization, design and architecture, build and deploy, test and verification, information and reporting. Since our focus is on RTO, the area of concern is test and verification. Figure 7.5 shows the maturity model for test and verification from the basic to expert level. It mentions the automation on various levels of testing e.g. unit testing, integration testing, etc.; however, RTO does not exist in the model at all. This goes to show that as either RTO is not widely adopted or is too difficult to be adopted as part of the CD process. As we discovered and mentioned in section 2.2, for many organizations, RTO is still largely a manual and isolated process. In this regard, we help to realize an easy adoption of RTO in the CD process which supports multiple levels of testing. RTO should be made mandatory for all testing process when its adoption proved to be resource saving. Nonetheless, to which level it should belong to in the CD maturity model is a question for further research.

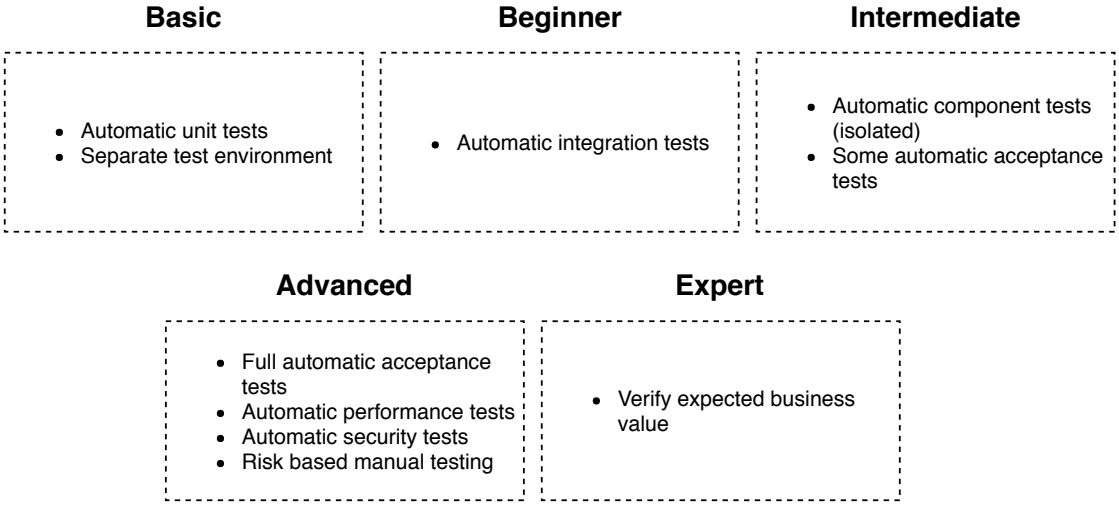


Figure 7.5: CD Maturity Model for test and verification [PRB13]

7.4.2 Smart Planning

In Jarvis, one of the main selling points is the self-organizing context of the delivery process. The end users can use the delivery model, providing within it the minimal information needed to run the process, and the framework should take care of matching all parameters passing between activities by gathering the relevant information on its own. This eases the modeling process as well as being more technically prepared for the project evolution. Nonetheless, as mentioned multiple times, smart planning is not implemented for the RTO adaptation. This problem is technically out of scope for this thesis. Because the parameters and the results of the RTO optimization services and the result of the test discovery service are of the same type. The current smart planning does not have the logic to automatically select the correct input between multiple services for the test run assessment. The problem is depicted graphically in Figure 7.6. This creates a problem to internally plan the test run service with smart planning. Thus, the smart planning itself should be extended further to cope with such a scenario.

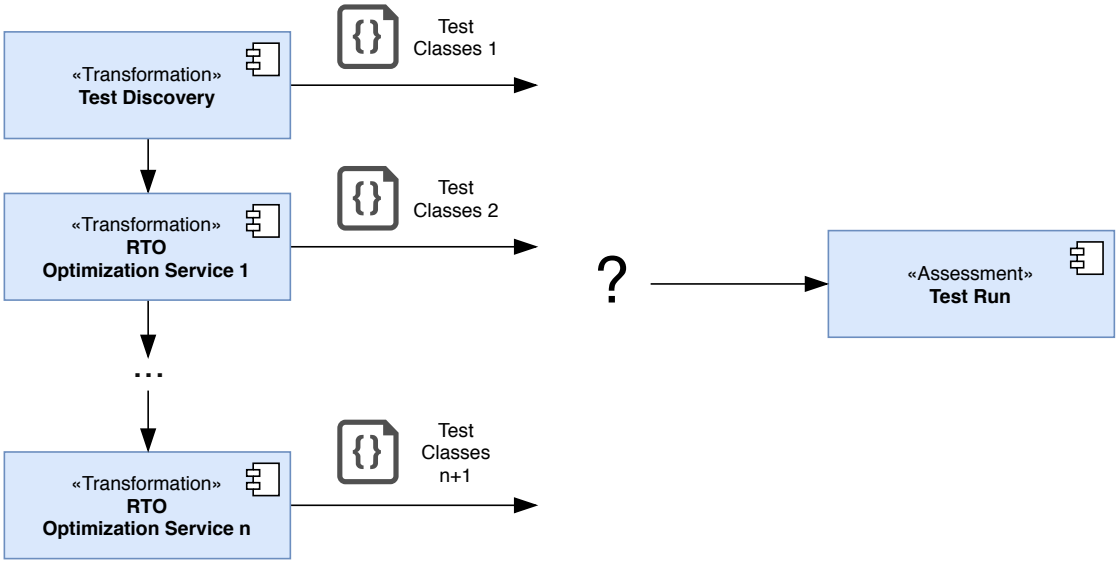


Figure 7.6: Smart planning problem scenario from RTO process

7.4.3 Knowledge Base

One of the earlier ideas for the RTO adaptation is to centralize the data sources within a new component called knowledge base such that all optimization data could be managed within a single service and accessed by the RTO optimization service via a common interface. It could also serve as a base for collecting RTO-related knowledge that perhaps can be used for further optimization in other areas such as performance testing. However, this idea was investigated but not formalized and implemented due to the problem of knowledge representation. From the background research presented in section 2.3, we

learned that there are many types of data used for the optimization process, resulting in a wide variety of data. Representing such heterogeneous data in a single representation is a very difficult, if not at all an impossible task. Furthermore, a single representation may also lose the semantics that we need from the optimization data itself. Therefore, this idea was replaced by an approach to use probe activities for data collection in combination with data storage services.

8 Conclusion

Contents

8.1 Summary

This thesis presented an adaptation of RTO for continuous delivery process. The main goal of this thesis is to support the adoption of RTO in continuous delivery by adapting RTO into a delivery pipeline with a possibility to integrate new and different RTO techniques with relative ease. From the background study, we found out that RTO still remains majorly a manual process and is only researched extensively by big organizations or for academic purposes. Based on related work, we realized that when RTO is automated, it is usually not integrated in the delivery process. The challenge is to adapt RTO to a delivery system in such that it can be integrated architecture-wise while providing all the needed functionalities of RTO and maintaining all the previous requirements of the delivery system itself.

From the challenge, the concept for RTO adaptation is developed, following the DDD approach. For our work, two existing systems are used, namely Lazzer as an RTO prototype and Jarvis as a delivery system. As the two systems are designed using different architectures, it is not straightforward to integrate them. The delivery system follows a microservice architecture, while RTO operates on a sequential pipeline as a monolith. From the architecture of the delivery system, we know that a new technology can be integrated as an activity service and this is also the case for RTO. To develop the concept for the adaptation, we drilled down the RTO process into the individual tasks. The RTO process includes test discovery, optimization, test run, and data collection(s). As a result, we developed an architecture for the RTO adaptation not as a single RTO activity, but multiple distributed activities with each concerning only its own functionality. The RTO services consist of the following activities: RTO test service for test discovery and test run, RTO optimization service for optimization of test suite, and RTO data collection service for collecting optimization data. To support this, a few extensions to

the delivery system framework are added. First is a new type of activity called probe for realizing general data collection service. Second is a concept for context awareness where activities can distinguish between delivery pipeline executions. The last is a concept for data storage services in the infrastructure layer for managing data sources for the optimization data.

The realization of the aforementioned concepts is developed using an iterative approach. We started the first iteration by integrating the whole RTO prototype in a single activity. The following iterations attempted to disassemble the initial RTO activity into microservices and ultimately moved the RTO pipeline from within the activity to the delivery model. As a result, the end users of the delivery system can model the RTO pipeline in their delivery pipeline to automate the RTO as part of the delivery process.

The implementation is then evaluated against the user's requirements for the RTO adaptation and the architectural requirements for the delivery system. We also evaluated the adaptation in regards to its software engineering qualities. While the RTO adaptation provides the flexibility for modeling the RTO process and integrating new RTO techniques, it makes the modeling process more complex as a whole. Nonetheless, we believe that it contributes such that it makes the adoption of RTO easier for software projects. The evaluation also provided a discussion of the differences between a monolithic and microservice architecture of RTO regarding different points e.g. data sources, optimization data, etc.

Overall, the RTO adaptation which was implemented in a distributed manner as microservices provides all the needed functionalities of RTO as well as the flexibility to realize new RTO techniques. In conclusion, through this thesis, we deliver the adaptation of RTO into the delivery system and make adopting RTO in continuous delivery a much easier task.

8.2 Future Work

The contributions of this thesis are only the first step of adapting RTO for continuous delivery. In this section, suggestions for possible future work are listed.

8.2.1 Improvement for RTO Adaptation

While the current RTO adaptation contains all the necessary functionalities, it still leaves room for improvement. The following outlines some suggestions.

Support for Other Test Frameworks The presented implementation only supports the JUnit4 framework. Adapters for other test frameworks such as the older or newer versions of JUnit should be implemented to be able to support more projects.

Support for RTM Our work on the RTO adaptation includes the concept for supporting RTS and RTP where the actual test suite is not reduced. RTM, while is considered a branch of RTO, permanently minimizes the size of the test suite by removing redundant

tests. This can be beneficial to combine with RTS and RTP techniques to achieve a more optimal test suite. Nonetheless, because the reduction is permanent, a different approach may be needed to conceptualize RTM.

Automated RTO Pipeline Test Because the RTO adaptation is distributively implemented as activities and the current delivery system framework does not support automated test for the delivery pipeline, proper integration testing is needed to test the complete RTO workflow. Furthermore, this is not limited to only the scope RTO. The delivery system itself can also benefit from automating the delivery pipeline test.

8.2.2 Smart RTO

The current work of our thesis only provides a basic structure for RTO in the delivery system. The test framework and strategies used have to be specified beforehand.

Automatic Test Framework Detection The test framework used for the project should be automatically detected by the RTO test service activity.

Automatic Strategy Selection While our work provides the flexibility to model different strategies to be used in the delivery pipeline, automatic strategy selection can be a very convenient feature for the end users because RTO can be adopted without an in-depth knowledge of the area.

8.2.3 RTO for Pipeline Planning

So far, with the current adaptation of RTO, the optimization process occurs only within the activity level. While this is already a big improvement to the delivery pipeline execution, RTO can perhaps be utilized for the delivery pipeline planning. Instead of optimizing tests only during the planning process, it can be more optimal to optimize the build plan e.g. building a subset of the project and test and then another subset and so on. The idea is such that we can create smaller partial builds to test only subsections of the tests according to the optimizer to achieve a more optimal pipeline plan.

8.2.4 Improvement for Smart Planning

We discussed the problem regarding smart planning in the scenario where the planner has to determine the input from the multiple outputs of the same type in subsection 7.4.2. The smart planner should be extended to decide in such a scenario which input is correct.

8.2.5 Improvement for Probe Activity

While the idea of probe activity was introduced to support the RTO process, it can be used for other general tasks as well. In this case, the concept should be developed further.

Probe Aggregation At the moment, the outputs from probe activities are not planned further. This means that they cannot be aggregated. There may be other scenarios where the outputs of probes can be used in other activities as well as aggregating from other probes themselves.

Different Levels of Data Collection We outlined in subsection 5.2.1 different levels of data collection for probe activities. Nonetheless, only the first which is data collection for artifact(s) was implemented to support RTO. The other two, data collection for delivery pipeline and delivery system, should be implemented as well because it can be useful in many cases such as KPIs report and system management.

Bibliography

[Agr+93] H. Agrawal et al. “Incremental regression testing”. In: 1993, pp. 348–357. ISBN: 0818646004. DOI: 10.1109/ICSM.1993.366927 (cited on page 6).

[AO08] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008. ISBN: 9780521880381 (cited on page 4).

[ASK04] K. K. Aggrawal, Y. Singh, and A. Kaur. “Code coverage based technique for prioritizing test cases for regression testing”. In: *SIGSOFT Software Engineering Notes* 29.5 (2004), pp. 1–4. ISSN: 01635948. DOI: 10.1145/1022494.1022511 (cited on page 8).

[Ber07] A. Bertolino. “Software testing research: Achievements, challenges, dreams”. In: *2007 Future of Software Engineering*. September. IEEE Computer Society, 2007, pp. 85–103 (cited on page 4).

[Bis+11] S. Biswas et al. “Regression test selection techniques: A survey”. In: *Informatica* 35.3 (2011), pp. 289–321. ISSN: 1854-3871 (cited on pages 7–9).

[CRK10] A. P. Conrad, R. S. Roos, and G. M. Kapfhammer. “Empirically studying the role of selection operators duringsearch-based test suite prioritization”. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 1373–1380. ISBN: 9781450300728. DOI: 10.1145/1830483.1830735 (cited on page 11).

[DS08] G. Duggal and B. Suri. “Understanding regression testing techniques”. In: *Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology*. 1. 2008 (cited on page 8).

[Dör17] J. S. Döring. *An Architecture for Self-Organizing Continuous Delivery Pipelines*. 2017 (cited on pages 14, 16, 18, 20, 22, 40, 41).

[Eks] *Website of Ekstazi*. URL: http://ekstazi.org (visited on 05/08/2018) (cited on page 27).

[EM00] S. Elbaum and G. Malishevsky, Alexey G Rothermel. “Prioritizing Test Cases For Regression Testing”. In: *ICSM 1998*. ACM, 2000, pp. 102–112. ISBN: 1581132662. DOI: 10.1145/347324.348910 (cited on pages 9, 10).

[ERS10] E. Engström, P. Runeson, and M. Skoglund. *A systematic review on regression test selection techniques*. 2010. DOI: 10.1016/j.infsof.2009.07.001 (cited on page 7).

[Eva04] E. Evans. *Domain Driven Design: Tackling Complexity in the Heart of Business Software*. Addison-Wesley Professional, 2004 (cited on pages 33, 36).

[Fow+99] M. Fowler et al. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. ISBN: 0201485672 (cited on page 34).

[Fow15] M. Fowler. *MonolithFirst*. 2015. URL: https://martinfowler.com/bliki/MonolithFirst.html (cited on page 34).

[Gli+14] M. Gligoric et al. "An empirical evaluation and comparison of manual and automated test selection". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14* (2014), pp. 361–372. DOI: 10.1145/2642937.2643019 (cited on page 5).

[Ham17] P. Hammant. *The Rise of Test Impact Analysis*. 2017. URL: https://martinfowler.com/articles/rise-test-impact-analysis.html (cited on pages 9, 10).

[HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010 (cited on pages 1, 11, 12).

[Inf] *Website of Infinitest*. URL: https://infinitest.github.io (visited on 05/08/2018) (cited on page 26).

[ISO11] ISO. *ISO/IEC 25010*. 2011. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (visited on 09/30/2018) (cited on page 73).

[KKT07] B. Korel, G. Koutsogiannakis, and L. H. Tahat. "Model-based test prioritization heuristic methods and their evaluation". In: *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM. 2007, pp. 34–43 (cited on page 9).

[KM08] R. Krishnamoorthi and S. A. Mary. "Incorporating varying requirement priorities and costs in test case prioritization for new and regression testing". In: *International Conference on Computing, Communication and Networking, 2008*. IEEE, 2008, pp. 1–9 (cited on pages 10, 11).

[KM09] R. Krishnamoorthi and S. A. Mary. "Factor oriented requirement coverage based system test case prioritization of new and regression test cases". In: *Information and Software Technology* 51.4 (2009), pp. 799–808 (cited on pages 10, 11).

[KP02] J.-M. Kim and A. Porter. "A history-based test prioritization technique for regression testing in resource constrained environments". In: 2002, pp. 119–129. ISBN: 158113472X. DOI: 10.1145/581339.581357 (cited on page 10).

[KTH05] B. Korel, L. H. Tahat, and M. Harman. "Test prioritization using system models". In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE. 2005, pp. 559–568 (cited on page 9).

[LF14] J. Lewis and M. Fowler. *Microservices - A definition of this new architectural term*. 2014. URL: http://martinfowler.com/articles/microservices.html (cited on page 34).

[LR06] M. Lippert and S. Roock. *Refactorings in Large Software Projects: performing complex restructurings successfully*. John Wiley \& Sons, 2006 (cited on pages 34, 35).

[LW89] H. K. N. Leung and L. White. "Insights into regression testing". In: *Proc. Conf. Software Maintenance*. IEEE, 1989, pp. 60–69 (cited on pages 1, 5).

[Mor07] G. Morgan. *Implementing System-Quality Attributes*. Microsoft Corporation. 2007. URL: https://msdn.microsoft.com/en-us/library/bb402962.aspx (visited on 10/01/2018) (cited on pages 75, 79).

[Mye+11] G. J. Myers et al. *The Art of Testing*. 2011 (cited on page 4).

[Ono+98] A. K. Onoma et al. "Regression testing in an industrial environment". In: *Communications of the ACM* 41.5 (1998), pp. 81–86. ISSN: 00010782. DOI: 10.1145/274946.274960 (cited on page 4).

[Opea] *OpenClover documentation*. URL: https://www.openclover.org/documentation (visited on 05/08/2018) (cited on page 24).

[Opeb] *Website of OpenClover*. URL: https://www.openclover.org (visited on 05/08/2018) (cited on page 23).

[Ple15] C. Plewnia. "A Framework for Regression Test Prioritization and Selection". PhD thesis. Research Group Software Construction, RWTH Aachen University, 2015 (cited on pages 13–15, 18, 37, 48, 76).

[PRB13] T. Palbourg, A. Rehn, and P. Boström. *The Continuous Delivery Maturity Model*. 2013. URL: https://www.infoq.com/articles/Continuous-Delivery-Maturity-Model (visited on 10/05/2018) (cited on page 84).

[Pre10] R. S. Pressman. *Software Engineering : A Practitioner's Approach*. 7th Editio. Vol. 32. 1. McGraw Hill, 2010. ISBN: 9780073375977 (cited on page 1).

[Rot+01] G. Rothermel et al. "Prioritizing Test Cases For Regression Testing". In: *IEEE Transactions on software engineering* 27.10 (2001), pp. 929–948 (cited on pages 7, 8).

[Rot+02] G. Rothermel et al. "Empirical Studies of Test-Suite Reduction". In: *Software Testing, Verification and Reliability* 12.4 (2002), pp. 219–249 (cited on page 6).

[Rot+99] G. Rothermel et al. "Test case prioritization: An empirical study". In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 179–188. ISBN: 0769500161 (cited on pages 8–10).

[RS16] C. Richardson and F. Smith. *Microservices: from design to deployment*. NGINX, 2016 (cited on pages 34, 35).

[Sch18] L. Schade. "Evaluation of Regression Test Optimization Techniques Evaluierung von Techniken zur Optimierung von Regressionstests presented by". bachelorthesis. RWTH Aachen University, 2018 (cited on page 82).

[Sin+12] Y. Singh et al. *Systematic literature review on regression test prioritization techniques*. 2012 (cited on page 9).

[SKS06] Y. Singh, A. Kaur, and B. Suri. "A new technique for version-specific test case selection and prioritization for regression testing". In: *Journal of the CSI* 36.4 (2006), pp. 23–32 (cited on page 8).

[SWO05] H. Srikanth, L. Williams, and J. Osborne. "System test case prioritization of new and regression test cases". In: *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 2005. ISBN: 0-7803-9507-7. DOI: 10.1109/ISESE.2005.1541815 (cited on pages 10, 11).

[Tes] *Website of Testsigma*. URL: https://testsigma.com/automated-regression-testing-tool (visited on 05/08/2018) (cited on page 25).

[Tlba] *TLB documentation*. URL: http://test-load-balancer.github.io/doc-0_3_2/concepts.html (visited on 05/08/2018) (cited on page 27).

[Tlbb] *Website of Test Load Balancer*. URL: http://test-load-balancer.github.io (visited on 05/08/2018) (cited on page 27).

[Tri] *Website of Tricentis Tosca*. URL: https://www.tricentis.com/regression-testing (visited on 05/08/2018) (cited on page 25).

[Wal+06] K. R. Walcott et al. "TimeAware test suite prioritization". In: *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006, pp. 1–12. ISBN: 1595932631. DOI: 10.1145/1146238.1146240 (cited on page 10).

[Won+97] W. E. Wong et al. "A study of effective regression testing in practice". In: 1997, pp. 264–274. ISBN: 0-8186-8120-9. DOI: 10.1109/ISSRE.1997.630875 (cited on pages 5, 8, 9, 11).

[YH07] S. Yoo and M. Harman. "Regression Testing Minimisation, Selection and Prioritisation : A Survey". In: 2 (2007). DOI: 10.1002/000 (cited on pages 1, 5–9).

[Zül05] H. Züllighoven. *Object-oriented construction handbook: Developing application-oriented software with the tools and materials approach*. Elsevier, 2005 (cited on pages 29–31).

[Sil+10] R. S. Silva Filho et al. "Supporting concern-based regression testing and prioritization in a model-driven environment". In: *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*. IEEE. 2010, pp. 323–328 (cited on page 9).

[Sri02] J. Srivastava, Amitabh Thiagarajan. "Effectively prioritizing tests in development environment". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 27. 4. ACM, 2002, pp. 97–106 (cited on page 9).