Traceability in continuous integration pipelines using the Eiffel protocol

Alena Hramyka
Martin Winqvist

Bachelor Thesis, Spring 2019
Computer Science and Applications Development
Department of Computer Science and Media Technology
Faculty of Technology and Society
Supervisor: Helena Holmström Olsson
Examiner: Yuji Dong
Abstract

The current migration of companies towards continuous integration and delivery as well all service-oriented business models brings great benefits but also challenges. One challenge that a company striving to establish continuous practices is the need for pipeline traceability, which can bring great enhancements to continuous integration and delivery pipelines as well as offer a competitive edge. This exploratory case study looks at the current and desired states at Axis Communications, a global leader in network video solutions based in Lund, Sweden. It further evaluates the practical and organizational aspects of the adoption of the Eiffel protocol in the company’s pipeline tools through developing a proof-of-concept Eiffel plugin for Artifactory. Based on the discovered technical and organizational needs and obstacles, it draws conclusions and makes recommendations on a possible strategy when introducing Eiffel in a company.

Keywords: continuous integration, continuous delivery, Eiffel protocol, software traceability, Artifactory.

Vocabulary

Artifactory, an open source repository for binary files developed by JFrog [1], a “universal artifact repository”. It is language-independent, which makes it easy to integrate it with a wide range of automated pipeline tools. Artifactory promises to make the process of development, delivery and scaling faster, easier and more secure [2].

Groovy, an Apache-licensed object-oriented, dynamically typed programming language that runs on the Java platform and can enhance Java program with such features as functional programming, scripting, type inference, etc. It was first released in 2003.

Pipeline, a sequence of software tools in a specific order, each responsible for performing certain activities on a software artifact. For instance, a typical pipeline will include shared repositories for source code, building/compilation activities, followed by various tests, both manual and automated, and packaging for release.
## Contents

Abstract 1
Vocabulary 1

1 Introduction 4
1.1 Research questions 5

2 Background 5
2.1 Continuous practices 5
  2.1.1 Continuous integration 5
  2.1.2 Continuous delivery 6
  2.1.3 Notes on terminology 7
  2.1.4 Benefits and challenges 7
2.2 Software traceability 9
  2.2.1 Benefits 10
  2.2.2 Traceability strategy 10
  2.2.3 Challenges 12
    2.2.3.1 Challenges due to agile practices and CI 13
2.3 The Eiffel Framework 14
  2.3.1 Usage of Eiffel in the industry 15

3 Research method 15
  3.1 Single-case study 15
    3.1.1 The case: Axis Communications 16
  3.2 Design and creation 16
  3.3 Research process 17
  3.4 Data generation 17
    3.4.1 Literature review 17
      3.4.1.1 Literature search method 18
    3.4.2 Interviews 18
      3.4.2.1 Theme-coding and analysis 18
    3.4.3 Participant observation 19
  3.5 Threats to validity 19
  3.6 Implementation 20

4 Results 20
  4.1 Interview results 20
    4.1.1 Interview A 21
      4.1.1.1 Current state 21
      4.1.1.2 Desired state 22
4.1.2 Interview B 23
  4.1.2.1 Current state 23
  4.1.2.2 Desired state 25
4.1.3. Interview C 27
  4.1.3.1 Current state 27
  4.1.3.2 Desired state 28
4.1.4 Summary of interviews 29
4.2 Implementation results 30
  4.2.1 Plugin design 30
  4.2.2 Eiffel protocol implementation 31

5 Discussion 35
  5.1 Technical implementation 35
    5.1.1 Plugin development 35
    5.1.2 Event publishing 36
    5.1.3 Artifactory plugin prototype 38
    5.1.4 Eiffel protocol schema evaluation 39
  5.2 Eiffel and organizational strategy 42
    5.2.1 Traceability needs 43
    5.2.2 Traceability strategy 44

6 Conclusions and future work 46

References 47

Appendices 50
  7.1 Appendix 1 — Interview questions 50
  7.2 Appendix 2 — Research activities 52
1 Introduction

Today, companies are moving towards supplementing their product line with services in order to remain competitive. There is added value in using services to add distinctiveness and differentiation to a product than to only sell the product itself [3].

As companies adopt a more service oriented business model, they need to make adjustments to their tools and processes to support it. The development cycle of a traditional plan-driven process is too slow to be able to handle the need of shorter time-to-market, changing customer demands leading to volatile requirements and easily disrupted markets. In light of this, software companies shift toward agile methods. This shift happens gradually with the first step being an agile product development department. While this allows for a more flexible software development process, other areas of the organization that are not yet agile stand in the way of further shortening feedback cycles. To expand agile practices to the system validation department, the next step in the shift is to adopt continuous integration [4].

As one integral practice in agile development, continuous integration is a practice where developers integrate their work frequently, at least daily. The changes are then built and tested automatically. It allows for quick implementation of new features and efficiently handles the needs that emerge in a fast changing business environment.

Keeping track of software artifacts such as source code changes, documentation, test cases, product versions and requirements and how they relate to each other is known as software traceability.

Previous research has established the benefits of traceability and the challenges of achieving it in large scale systems [5], [6], [7], [8]. One of the major identified barriers for accomplishing end-to-end traceability is the nature of the agile workflow, specifically in the context of continuous integration. The technical diversity of the actors in a continuous integration pipeline requires large scale efforts towards ensuring communication between services that do not “speak the same language”.

A proposed solution to the problems of traceability in a continuous integration context is the open source Eiffel framework, developed at Ericsson AB. Eiffel is a technology agnostic communications protocol and a set of tools built around this protocol. One previous study has validated Eiffel’s ability to solve traceability issues in continuous integration pipelines in large scale systems [8].

However, while the benefits and challenges of achieving traceability in a continuous integration context are well documented, there is a lack of research regarding the current and desired state of traceability practices at companies that have adopted continuous integration. Since Eiffel is a relatively newly proposed solution, there is no research that explores how an organization that currently does not use Eiffel can
implement it in order to achieve end-to-end traceability in a continuous integration pipeline.

In this thesis, we conduct a single-case study at a large Swedish embedded systems company that is in the midst of servicizing their product offerings. Their development process is agile and continuous integration practices are well established. There is an explicit desire to improve traceability efforts in order to visualize activities in continuous integration pipelines, optimize testing efforts, empower decision making when it comes to process improvement and future development efforts and to drive the pipeline using traceability information.

In our study, we explore the current state of traceability in the case company as well as the desired future state. The aim is to identify obstacles for traceability and how these can be addressed by using the Eiffel protocol. As one critical part of this research, we implement a subset of the Eiffel protocol, resulting in a prototype that is used as a vehicle for exploration of the technical and organizational issues and strategies when adopting Eiffel.

1.1 Research questions

- **RQ1**: What is the current state of traceability practices in an organization that has adopted continuous integration?
- **RQ2**: What is the desired state of traceability practices in an organization that has adopted continuous integration?
- **RQ3**: How can the Eiffel framework be used to achieve the desired state?

2 Background

2.1 Continuous practices

2.1.1 Continuous integration

*Continuous integration* (CI) was born as part of the Extreme Programming, a popular agile development methodology [9], at the end of the previous century [10].

CI refers to a philosophical and practical approach to the software development process, where the developer commits his or her code changes to a shared repository at least daily, after which the new changes undergo automatic building and testing, making it possible to release updates to the customer frequently [9].

The main idea behind CI is that the changeset should be small and contain only one functionality. The smaller the change, the faster it is to build, test and debug new changes as the single bug-introducing commit is easier to identify because less code has to be inspected [11]. The main contributions of CI to the software development process are thus envisioned to be: the reduction of risks; reduction of repetitive manual processes; easy and fast generation of deployable software; enabling of better project visibility; and establishment of greater confidence in the product on the part of the development team [12].
Continuous integration has a set of cultural values at its center. Firstly, it requires developer discipline to commit code to the shared repository often and not to accumulate changes in local branches. Secondly, it expects a certain level of responsibility from developers. Since changes are committed often, one failed build can break the workflow for the rest of the team. It is difficult to understand whether all the commits coming after the problematic are fault-free. Fixing the bug in the “bad” commit becomes more time-consuming with every new commit pushed on top of it. Getting back to “green”, i.e. fixing the bugs in the failed build, has thus to be prioritized by the whole team and especially the developer responsible for the “bad” one [9].

Automation of such pipeline activities as building and testing is the domain of CI’s sibling practice, continuous delivery [11].

### 2.1.2 Continuous delivery

Continuous delivery (CD) is the practice of always keeping the recent changes, and the code base in general, in a releasable state [10]. It concentrates on the importance of constructing and fine-tuning automated pipelines for each development project. An automated pipeline implies that the tools that perform building, testing, packaging etc. communicate between themselves and trigger activities in each other without a developer’s participation [13].

Setting up a traditional, manually configured pipeline for a new project can take up to a month: the engineers need to request machines from the IT department, configure them, setup the environment, and perform many other steps manually. Moving integrated changes down the production pipeline can imply rebuilding several times for different tests, and doing it manually in different environments, which often leads to the infamous phenomenon of “it works on my machine” [11].

In CD, most of the activities in the pipeline, including configuration, are done automatically, which means fewer chances of introducing human errors or producing differing builds due to different build environments, and less time spent on taking care of the pipeline. Hence, even if a build takes a long time, that does not matter to the team as it is an autonomous process that needs human intervention only when a build or test fails [11], [13].

CD is thought to empower the development teams as it allows the developers to integrate and test their changes in a complete, fault-free product; it reduces errors because many of the previously manual activities become automated; and it lowers stress as every successful build is the proof that the product works, as opposed to having to wait for the build and test results after massive, multiple-commit integrations [14, p. 9]. Such industry giants as Facebook, Amazon, Etsy, Flickr, Google, and Netflix are early successful adopters of continuous practices. For instance, Flickr boasted around 10 deployments per day in 2009, and Etsy had 11,000 deployments in 2011 [15].
2.1.3 Notes on terminology

CI and CD are often used as two interchangeable terms, or are discussed as part of each other as they are highly interrelated notions [16], [17].

One approach [17] treats automated builds and tests as part of the CI practice. Another approach sees CI’s domain stop at the committing stage, with testing and building being CD’s responsibility, while CI is considered to be a part of CD [16] (see Fig. 1). According to a third approach [10, p. 445], the line between CI and CD should go where the developer’s activities stop and automated pipeline steps begin. CI is thus a developer practice, i.e. an individual programmer’s adopted habit and discipline of integrating new code often, while CD is a development practice, where developer’s code is run through an automated pipeline of tools, each of which performs its task in preparing the updated code for release.

Every “green” build is potentially a new release [10], and the ability to automatically ship a new release to the customer is called continuous deployment (CDP), which can be considered an operations practice [10], [17]. However, as with CI and CD, CD and CDP are often used synonymously or interchangeably in literature as deployment to the customer as a separate step is not even relevant in some development contexts [16].

CI, CD and CDP, as well as their contiguous practices of continuous release and continuous software engineering, commonly known as continuous practices [10, p. 440], [17], are now becoming the industry standard development automation techniques. However, even though there is literature and guidelines on how to introduce CD/CD in an organization, migration to it implies significant efforts, and those who manage to successfully adopt it face a number of problems on their way [16].

2.1.4 Benefits and challenges

Continuous practices promise and bring great benefits to the companies who succeed in implementing them successfully, such as faster and more reliable software releases, improved quality and user satisfaction, better competitive edge and improvements in developer productivity and efficiency, as well as improved
communication within and between teams and better predictability of the project [13], [18]. Nevertheless, there is a number of unsolved challenges in the domain.

Some of them revolve around the problem of introducing continuous practices in an organization. It involves restructuring the whole development process and the way teams collaborate, i.e. changing the mindset and the workflow processes in the company and adopting new techniques [13], [19]. Other organizational challenges include lack of skilled and experienced engineers; lack of awareness and transparency; resistance to change; and skepticism [17].

Apart from the organizational problems, introducing continuous practices has a number of technical challenges. There is a great variety of tools that can be used in a CD pipeline (Fig. 2), and as each application might require its own pipeline, the result is a significant variety in how the pipeline can look and what tools are used. Since there is no ready-made solution for building pipelines, an organization might need to spend a considerable amount of money and effort on implementing its own solution [13]. System design is often reported as being the main stumbling block when adopting CD [16]: a company might experience difficulties in unifying services or units used in the pipeline, adjusting or working around unsuitable architectures, resolving internal dependencies, and modifying software database schemas to adapt to the needs of CD [16].

The aforementioned diversity of tools is contrasted by the so-called “vendor lock-in” as an opposite technical challenge that can be hard to tackle [13]. Proprietary solutions can impose their own ways of solving a problem, and a vendor’s product often works best with another tool from the same vendor, which makes it hard to achieve interoperability with the tools and services developed by the vendor’s competitors. It can thus be time-consuming for the architects of a CD pipeline to get some of the tools work well in combination with other proprietary or open-source
software. For instance, Atlassian, the company behind the VCS repository hosting service BitBucket and the issue tracker Jira, among other products, offers integration between the two; all 26 of the features are only available if both of the products are used, and if the customer decides to use an alternative to Bitbucket, only 11 of the features will be available [20]. Some of the solutions to the vendor lock-in problem might be to stimulate the acceptance of industry standards and open APIs, as well as the development of a “plug-in ecosystem” [13].

Despite the variety of available tools, lack of appropriate software is identified as a problem [17]. The limitations that available tools have are reported to be an issue when adopting continuous practices. The existing ones either lack needed functionalities or fail to help organizations adopt continuous practices. The deficit of appropriate technologies for delivery and deployment of software with client-specific configurations is an obstacle in the branch of embedded systems. [17, p. 3926]. Further, test automation is reported to be poorly supported, and there is weak support for code reviews and automated test results feedback. Tools used in the delivery and deployment pipeline often suffer from security vulnerabilities and offer poor reliability, and the frequent introduction of new tools and changes in available ones is taxing for developers, who need to spend a lot of time on learning and getting comfortable with them [17].

Further, even though the rapid release model and shorter release cycles has many pros, more effort has to be put into testing and creation of designated testing teams. It also necessitates a better QA process with new approaches to establishing backward compatibility and managing test automation infrastructure. [19, p. 24]

Thus, adoption of continuous practices brings its own benefits and challenges. There is, however, another issue that is currently a challenge but that has the potential of becoming a benefit, bringing enhancements to the CI/CD and beyond: software traceability in the CI/CD pipeline [21].

2.2 Software traceability

A commonly agreed upon definition of software traceability is “the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process”, and was first formulated by the Center of Excellence for Software and Systems Traceability (CoEST) [22].

The importance, benefits and challenges of being able to follow a requirement’s lifecycle in both forward and backward direction during a software development process is widely known since the mid 90s [5]. Since then numerous prominent papers have been written on the subject [6], [7]. Previous research has mostly been interested in a particular area of traceability, namely: requirements traceability.

There are however various other areas of software traceability that are brought up in literature that explore traceability in the context of continuous integration. In [8] researchers interviewed engineers in the industry with the goal of identifying
different traceability needs. They found the following 15 different themes (see Table 1).

**Table 1. Traceability needs identified in [8].**

<table>
<thead>
<tr>
<th></th>
<th>Traceability needs identified in [8].</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test Result and Fault Tracing (Trf)</td>
</tr>
<tr>
<td>2</td>
<td>Content Upstream Tracing (Ust)</td>
</tr>
<tr>
<td>3</td>
<td>Content Downstream Tracing (Dst)</td>
</tr>
<tr>
<td>4</td>
<td>Context and Environment Tracing (Env)</td>
</tr>
<tr>
<td>5</td>
<td>Requirement and Work Item Tracing (Req)</td>
</tr>
<tr>
<td>6</td>
<td>Monitoring (Mon)</td>
</tr>
<tr>
<td>7</td>
<td>Metrics and Reports (Met)</td>
</tr>
<tr>
<td>8</td>
<td>Documentation Tracing (Doc)</td>
</tr>
<tr>
<td>9</td>
<td>Salvageability (Slv)</td>
</tr>
<tr>
<td>10</td>
<td>Usability (Usa)</td>
</tr>
<tr>
<td>11</td>
<td>Persistence (Per)</td>
</tr>
<tr>
<td>12</td>
<td>Culture and Process (Cul)</td>
</tr>
<tr>
<td>13</td>
<td>Security and Stability (Sec)</td>
</tr>
<tr>
<td>14</td>
<td>Speed (Spd)</td>
</tr>
<tr>
<td>15</td>
<td>Automation (Aut)</td>
</tr>
</tbody>
</table>

Out of these, the first three (Trf, Ust, Dst) were singled out as mentioned most often by interview subjects. Test Result and Fault Tracing concerns the ability to generate trace links between test results or faults to a specific source or system change. Content Upstream Tracing addresses the need to track the contents of system revisions, both for documentation purposes and to track the development, integration and test progress of features. Content Downstream Tracing answers the question “where did a particular artifact end up?” This would, for example, allow a developer to follow committed source code through different activities in a continuous integration pipeline [8].

### 2.2.1 Benefits

The benefits of traceability are well known and thoroughly documented. In a report on traceability in agile projects a number of key benefits were highlighted [23, p. 268]: change impact analysis, product conformance process compliance, project accountability, baseline reproducibility, and organizational learning.

A benefit that is often overlooked is that generated trace links can be used to drive the continuous integration pipeline. Traces can facilitate increased automation in combination with improving decision making at some stage in the pipeline by providing an actor with traceability information [8].

A paper on agile methods in regulated environments [24, p.870] specifically linked an increase in traceability to concrete benefits: “audits which used to take two days are now being completed in less than a day, often with no open issues to respond to, and resounding approval from audit assessors who appreciate the complete transparency and flexibility afforded by the living traceability allowing them to interrogate aspects of the development process at will.”

### 2.2.2 Traceability strategy

Even though traceability is considered beneficial to a project, gathering traceability data for every imaginable development activity is neither cost effective nor beneficial. Without proper planning on a per-project basis, captured traceability information can end up being unordered and of no use [25].
The minimum requirements for an effective traceability strategy according to the Center of Excellence for Software Traceability are [22, p. 8]:

- Having retained the artifacts to be traced.
- Having the capacity to establish meaningful links between these artifacts.
- Having procedures to interrogate the resulting traces in a goal-oriented manner.

These requirements, while seemingly simple, include the fact that traces not only need to be created and stored, but also maintained and used to answer project related questions.

![Fig. 3 A generic traceability process model [22, Fig. 1].](image)

In order to formulate a per-project traceability strategy, it is necessary for the project process to include activities, both from an engineering and a management perspective, that enable the planning of traceability from the very start of the project [25]. An organization's traceability solution must be flexible enough to fit into a wide array of different projects. It needs to be lightweight, easily changed and engineered to be used effectively even if the project and its artifacts are prone to frequent changes, as is common in agile approaches [23].

An effective strategy should include descriptions of manual and automated tracing activities, chosen tools, areas of responsibility, when to carry out tracing activities and how they should be assessed. In addition to this, efforts need to be taken to establish the granularity of captured traces, in other words: which events are
important to trace and which data and metadata should be included in the trace links [22].

Fig. 4 Planning and managing a traceability strategy [22, Fig. 2].

2.2.3 Challenges

Despite the benefits of traceability, organizations have historically been unable to implement satisfactory traceability solutions. There are several reasons for this, including a lack of immediate benefits, traceability not being considered cost-effective and inadequate tools for capturing, maintaining and using traceability information.

An important characteristic of a well engineered traceability solution is that it is always there, built into the software development process. For people contributing to the development efforts, traceability should be achieved automatically by leveraging their everyday tools without having to do something extra to generate traceability information. This concept of ubiquitous traceability is called the grand challenge of traceability by the Center of Excellence for Software & Systems Traceability [22]. In order to achieve the grand challenge of traceability, seven prerequisite challenges must be tackled.

Some of these challenges are of a technical nature and concern characteristics such as scalability and portability [22]. The solution must be able to offer stakeholders the data they need, depending on where they operate within the organization or the development process. Traceability information should ideally be available at the right level of granularity and only for the activities that are needed to be traced by
various stakeholders. The solution must also be portable in order to be reused and shared between numerous projects in a technology diverse environment.

Considering the complexity of today’s software engineering process, developing a robust traceability solution that is both scalable and portable requires overcoming numerous non-technical challenges before development can even begin. The needs of stakeholders must be mapped out in detail to be able to establish a clear purpose for traceability and ensure that the solution will support the stakeholders [22]. Knowing that needs and requirements can change quickly, as well as the environment in which the traceability solution operates in, it must be configurable to be able to respond to change efficiently [22].

Having a well thought out traceability strategy that takes different stakeholders’ needs into account, and even a clear view of the benefits that traceability offers, is not enough if the solution is not deemed cost-effective. The resources needed to establish end-to-end traceability must be in line with the return on investment [22].

Stakeholders need to have confidence in the correctness of the traceability information, and the solution has to be trusted. This also touches on challenges that concerns the culture and values of an organization and leads to having to demonstrate and convince people of the value of traceability. Stakeholders must view traceability as a formal strategic goal and be aware of their responsibilities.

Table 2. Traceability challenges identified in [22]. Challenge 8 (ubiquitous) is referred to as the grand challenge, it requires having made major progress with the other seven challenges.

| 1. Purposed | 5. Scalable |
| 2. Cost-effective | 6. Portable |
| 3. Configurable | 7. Valued |
| 4. Trusted | 8. Ubiquitous |

2.2.3.1 Challenges due to agile practices and CI

Traditional approaches of manually keeping track of artifacts might work well in small scale projects or in slow moving, waterfall-like, project processes. However, these methods do not scale well and are not flexible enough to address the traceability needs in agile large scale projects.

In a complex software development process that has adopted continuous integration it is not uncommon to generate thousands of artifacts that may undergo multiple changes made by hundreds or thousands of different engineers, effectively making traditional methods of achieving traceability costly and time consuming [8], [23].

In agile methodologies traceability is often overlooked and not compatible with the core values mentioned in the Agile Manifesto [26].

Agile projects generate few traceable artifacts in the design and requirements phase of a software project [23]. Working software is prioritized over documentation, but documentation is required to achieve end-to-end traceability [24].
Agile processes often go hand in hand with continuous practices. As mentioned earlier in the background, the technology diversity present in a continuous integration pipeline stifles out-of-the-box communication between different system actors. A vast array of different proprietary, open source and in-house tools can be used in the development of software. All these tools may use different formats and standards which must be parsed in order to generate accurate trace links [8].

In addition to this, agile methods and CI dramatically increase the number of generated artifacts compared to traditional processes. Since it lies in the nature of an agile process to respond to change instead of following a plan, artifacts frequently change which means that trace links need to be maintained and updated to ensure accurate traceability information.

This is further complicated by the fact that multinational companies might coordinate engineering efforts across several countries which adds another layer of difficulty when it comes to the development and maintenance of traceability solutions [8].

2.3 The Eiffel Framework

The Eiffel framework was created in 2012 at Ericsson, a Swedish network and telecommunications company. The motivation behind developing the Eiffel framework came from the need to achieve high traceability in a continuous integration and delivery context. The protocol has since then been released under an open source license with the expressed goal of releasing open source implementations of the different parts of the framework in the future.

The framework consists of two different parts, a technology agnostic event-based communication protocol, essentially a vocabulary and syntax that describes events, and a suite of tools built around the protocol. Examples of such tools are services that implement the protocol and enable standardized communication between the actors in a continuous integration pipeline. These services are typically implemented as plugins for various tools like Jira, Gerrit, Jenkins and Artifactory and enables them to send and receive Eiffel events.

The basic concept of Eiffel is that continuous integration and delivery activities generate an immutable Eiffel event that is broadcasted globally on a message bus. It is important to note that these events are created in real time, not generated after the fact. Each event references other events through semantic trace links, resulting in a directed acyclic graph (see Fig. 5) that can be traversed to answer traceability questions. In addition, these events are used to drive the continuous integration pipeline [27].
2.3.1 Usage of Eiffel in the industry

At the time of writing, Eiffel is a relatively new technology and is not commonly used in the industry outside of Ericsson AB. However, other large software companies have shown interest in adopting Eiffel and have started taking the necessary steps to integrate it with their software development process [8].

At Ericsson AB, Eiffel is being used in large scale projects with thousands of engineers contributing to the development from multiple different sites across the world. To facilitate development efforts in such an environment, both continuous integration and deployment systems are in place. One such project with high demands on traceability due to regulations, legislations and official company strategy was part of a paper conducted to validate the Eiffel framework [8]. The paper showed that Eiffel was successfully used in a large scale project to address test result and fault tracing, downstream content tracing and upstream content tracing.

3 Research method

This research was conducted as an exploratory single-case study in combination with the design and creation strategy over a three-month period (March 2019 - May 2019) at Axis Communications in Lund, Sweden. The data generation methods used were semi-structured interviews, participant observation, and literature review.

3.1 Single-case study

Software traceability is a problem that is theoretically trivial in the sense that it essentially boils down to keeping records of certain activities together with metadata and the relationship between these activities. However, in practice this is made difficult due to the complexity of the system in which traceability must be achieved. A case study is a suitable method when the complexities of the real world are integral to the research. It allows the researchers to study the chosen case in a real-life setting, and gain insight about issues that arise due to complicated processes and relationships [28].
This paper carries out an exploratory short-term contemporary single-case study [28] that aims to identify the multi-faceted and interconnected challenges and issues when it comes to software traceability in the context of continuous integration, and to compare the studied case with insights gained from previous literature. As explained in [28]:

An exploratory study is used to define the questions or hypotheses to be used in a subsequent study. It is used to help a researcher understand a research problem. It might be used, for example, where there is little in the literature about a topic, so a real-life instance is investigated, in order to identify the topics to be covered in a subsequent research project.

In our case, exploration was not meant to find questions or topics for the study; rather, exploration was the essence of the research itself. Eiffel is a very new technology rife with topics for study, both technical and organizational. While the findings of this thesis can no doubt lead to further research and more questions and hypotheses, the focus of this study is to explore the current traceability situation, the desired situation and how Eiffel can be used to achieve it, on the example of one company.

3.1.1 The case: Axis Communications

Axis Communications AB is a Swedish embedded systems company. Agile practices are well ingrained within the different departments of the company and software is both integrated and deployed continuously.

The continuous integration pipeline at Axis consists of various tools and services to facilitate the development process, such as Jira for handling issues, Git for version control, Gerrit for code review, Jenkins CI-server and Artifactory for storing artifacts in combination with various custom tools and services for integration testing and persistence.

There is an explicit desire to improve software traceability using the Eiffel framework, but apart from a plugin for Jenkins and experimental implementations the Eiffel framework is not used at all at the company. There is an official long term goal to develop plugins for all actors in the continuous integration pipeline.

Only a small amount of people are familiar with the Eiffel framework, but many stakeholders recognize the need for traceability in order to improve data gathering for metrics, identify bottlenecks in the workflow and processes, inform future development efforts, visualize activities in the pipeline and to drive the pipeline using traceability information.

3.2 Design and creation

The method of design and creation is suitable for research that involves developing new IT solutions. The process results in the creation of one or more of the following artifacts: constructs, models, methods and instantiations [28]. This research offers an instantiation in the form of a proof-of-concept implementation of a subset of the Eiffel protocol. This implementation is presented in Chapter 4.2.
The developed prototype is used to illustrate the drawn conclusions and formulated theories. The contribution of new knowledge is further guaranteed by the fact that the Eiffel framework is a new technology and commonly used solutions and implementations do not yet exist.

While choosing a research method, action research was discussed as an alternative to design and creation. It was dismissed due to the time restrictions of the research project and the fact that design and creation is better suited to explore new technology and solutions while action research is concerned with safe solutions based on commonly used technology [29].

### 3.3 Research process

The flowchart below (Fig. 6) shows a high level outline of the research process. In reality this process was carried out in an iterative fashion. Data collection and data analysis continued throughout the whole research period, leading to frequent re-evaluation of the design and development phases, which triggered a re-examination of analysis, evaluation and conclusion, that prompted a new cycle of data generation and analysis.

![Fig. 6 High-level outline of the research process.](image)

### 3.4 Data generation

The data generation methods used in this study are semi-structured interviews [30, p. 89] and participant observation. Secondary data was generated through a literature review.

#### 3.4.1 Literature review

Literature review was conducted to gather information for the background on continuous practices, research in traceability and the current state of the Eiffel framework. The majority of the sources chosen are research papers published in scientific journals and books. Exceptions were made for [11], originally a blogpost, as
it is a widely referenced source (with 610 citations on Google Scholar at the time of writing) for one of the first well-rounded descriptions of CI.

The information gathered from the literature review was used as input for formulating the interview protocol questions used in the semi-structured interviews.

3.4.1.1 Literature search method

The starting point of the literature review was [8], followed by the techniques of backward snowballing to identify the literature on traceability and continuous practices, and forward snowballing to study what resonance the framework has produced in the research community. Further search for relevant literature included such keywords as software traceability, agile traceability, requirements traceability, continuous integration, continuous delivery, continuous practices, etc. on Google Scholar, where only relevant articles were selected.

3.4.2 Interviews

Semi-structured interviews with open-ended questions [28, p.193], [30, p.90] were conducted with experts from Axis Communications to establish the current situation of traceability practices in the company, as well as the desired state of traceability practices in the future.

[28] and [30] give different definitions of semi-structured interviews. According to the former, one is to follow a set of questions as a guideline but be open to ask new questions as they arise during the conversation. According to the latter, a semi-structured interview has to include a number of closed questions with predefined answer options, followed by open-ended questions. We opted for the first definition in our research.

The semi-structured form of the interviews was chosen to make it possible to discover the issues that interviewees currently face in the areas of their work relevant to the research, but at the same time encourage them to speak freely about their own ideas, concerns and needs.

During the interviews an interview protocol was used (see Appendix 1) with questions grouped into the following predefined themes:

1. Current traceability practices (benefits and challenges)
2. Desired state of traceability practices

Each interview subject received both a transcript of their interviews together with the summary of results presented in this paper in order to make sure that we correctly interpreted what was said during the interviews.

3.4.2.1 Theme-coding and analysis

Data generated in this research is qualitative. In qualitative analysis it is not possible to rely on standard mathematical or statistical methods which are commonly used in quantitative analysis [28].
The first step is to prepare the data for analysis by transcribing the interviews. This process is not only a prerequisite for further analysis, but it offers a chance to get a clearer overall impression of the data.

After having a full textual representation of the interviews, the relevant sections were identified and segmented into two different themes, (1) current state and (2) desired state. These are the same themes as those that were used when designing the interview questions. By using these themes it is possible to quickly identify if the interviews resulted in collecting the necessary data needed for the research.

Next, there is a need to further break down the themes and patterns that emerged in order to acquire a deeper understanding of the problems and goals. To achieve this, the interviews were theme-coded [28, p. 271], using an iterative process, according to the categories identified in [8], previously mentioned in Chapter 2.3 of this paper.

### 3.4.3 Participant observation

In addition, throughout the research process we worked at the offices of Axis Communications in Lund, Sweden. The thesis was done in cooperation with the Platform Management department, and our workstations were at the R&D Tools department. We had an opportunity to easily reach out to experienced engineers, ask for advice and information at any time. Different formal and informal meetings and workshops were conducted in order to gain a better understanding of the technologies involved, the Eiffel framework itself, and the traceability needs in the company (see Appendix 2).

### 3.5 Threats to validity

A common critique of single-case studies is that the results, theories and conclusions they produce can be difficult or even impossible to generalize. This piece of conventional wisdom has been challenged by researchers in the past [31].

While there are aspects of the chosen case that are unique to it, there are reasons to believe that companies of comparable size and software development process are concerned with the same challenges when implementing an end-to-end traceability solution. Companies are using slightly different tools and services in their pipelines, but on the whole they operate in mostly the same way. Any organization that adopt continuous integration will have an automated test suite, a shared repository to which source code changes are continually integrated and a modular software design approach [4].

With this in mind, it is reasonable to assume that the implications that develop from this case study are also applicable to other similar organizations, and the recommendations for action are suitable in other similar contexts as well. Eiffel was developed at Ericsson AB, and its creators are the pioneers of pipeline traceability in the research community. Consequently, a number of papers used in this thesis are relatively new research, may refer to each other and have not gained
large traction or citation numbers yet, e.g. [8], [10], [18] and [21]. We see this as an expected state of affairs for a young area of research.

3.6 Implementation

The practical implementation activities followed the evolutionary approach to developing a prototype [30, p. 103] in an agile manner. The plugin is proof-of-concept software and was not envisioned to be production-ready at the end of the research. The implementation results are described in Chapter 4.2, and the limitations and discussion are presented in Chapter 5.1.

4 Results

4.1 Interview results

In the table below, the different traceability needs that were brought up during each interview are shown. The traceability themes identified in [8] corresponds well to the needs brought up by the interviewees in this case study (see Table 3).

Table 3. Mapping of the traceability needs described in [8] onto the interview results.

<table>
<thead>
<tr>
<th>#</th>
<th>Traceability need</th>
<th>Interview</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>Test Result and Fault Tracing (Trf)</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>Content Upstream Tracing (Ust)</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>Content Downstream Tracing (Dst)</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>Context and Environment Tracing (Env)</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>Requirement and Work Item Tracing (Req)</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>Monitoring (Mon)</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>Metrics and Reports (Met)</td>
<td>X</td>
</tr>
<tr>
<td>8</td>
<td>Documentation Tracing (Doc)</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>Salvageability (Slv)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Usability (Usa)</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>Persistence (Per)</td>
<td>X</td>
</tr>
<tr>
<td>12</td>
<td>Culture and Process (Cul)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Security and Stability (Sec)</td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>Speed (Spd)</td>
<td>X</td>
</tr>
<tr>
<td>15</td>
<td>Automation (Aut)</td>
<td>X</td>
</tr>
</tbody>
</table>
4.1.1 Interview A

The first interview was conducted with a firmware architect who spends half their time in an automation team responsible for services and tools used for testing purposes in a continuous integration environment.

4.1.1.1 Current state

The automation team develops a framework for testing and uses Jenkins for carrying out the tests. Due to non-standard requirements when it comes to selecting products to run tests on, several in-house developed tools are used in this process.

Today, all test results, if they fail or pass, together with the product name and firmware version used when running the test, are stored in an Elasticsearch database. This is an automated process that is facilitated through Jenkins. Traceability information is saved in several other places, and the information often overlaps. Unless the need to trace a test far back into the past is important some traceability information can be found in Jenkins. There is also an in-house developed tool that contains the same information as the Elasticsearch database, but it is used primarily for visualizing the test results.

The main benefit of traceability is that it empowers the decision making when choosing which tests to run or not, “what traceability would lead to is that you can choose tests in a better way. Better meaning different things in different situations, but you get more tools for doing smarter selections of tests.” Traceability also enables the automation team to search through the stored history of previous test results. If a test exhibits unwanted or unusual behavior, it is possible to look at previous test results to examine if the same behavior has been triggered when testing a product in the past. This is mostly valuable when troubleshooting the actual test code, to make sure that it behaves the way it is supposed to. In addition to troubleshooting, experiments are being conducted into constructing dynamic test suites by analyzing failure rates and execution times in traceability data.

Something that is deemed difficult to trace today is “connecting specific test cases to specific code packages or lines of code”. There is no efficient and automated way to trace a test to a certain Gerrit review or source code commit, instead it involves a considerable amount of time and manual effort.

In order to select which tests to run on a certain product, a broad feature concept is used. While it is useful for the purpose of choosing tests, “it’s not so good at pinpointing in detail what could have gone wrong, maybe it’s not so much gaps, it’s maybe that it’s not so granular in some places”.

In today’s workflow each test in some of the test suites is tagged with the name of a feature from a list of features. This enables tracing a test result back to a given feature. This does not handle the use case where one needs to trace a test result back to a certain code package or code commit.
A major obstacle in the way of achieving this level of traceability is that “no one has done an initial mapping between the high-level features we have on the camera and the actual low-level code packages or specific reviews”. The captured traceability information has the wrong level of granularity for this particular need. This is further clarified by the interviewee: “one could wish that you could instrument the firmware so when you run tests it’s logged somewhere ‘ok now we are executing in these code packages’, alternatively that everything is tagged with something similar to the features but on a lower level so you actually know which tests affected which code packages”. With access to such information it would be possible for developers who want to integrate a change to selectively run a subset of tests that are designed to “hit that area” instead of executing all tests connected to a certain feature.

The work surrounding Eiffel at Axis is still in an early stage, mostly consisting of different experimental tools and services. This is also true for Eiffel usage in the automation team. At the moment they are experimenting with a tool that listens for Eiffel events broadcasted by Jenkins and other build tools that signals that an artifact is available, and acts upon these events. Without Eiffel it is not possible to know when an artifact is available, instead engineers need to poll or manually check if an artifact has been created.

When an artifact is made available, a test suite is built dynamically to test the published artifact. The test suite then runs the tests and sends out Eiffel events with the test results.

4.1.1.2 Desired state

In the context of testing in continuous integration, there are various different benefits from implementing a traceability solution using Eiffel. Once there is a system in place that enables end-to-end traceability, it is possible to develop tools that visualize the flow of activities in the pipeline, “from source code change all the way to release to client and being able to visualize it, go step-by-step in an efficient way... most of all when there’s a fault, it becomes easier to talk about and discuss what happened”. This is not possible today, instead engineers have to pour through logs and other information in different databases.

The interviewee highlights that it is difficult to achieve the traceability needed for this use-case due to “it becomes very complicated because things [tools and services] communicate in various strange ways” and they believe that Eiffel is the right tool for overcoming this challenge since it facilitates communication across different tools and services through a shared protocol.

There is an explicit want to break out the logic regarding artifact notification from the experimental framework currently in place, and instead implement that part of the Eiffel protocol in Artifactory, a tool for storing artifacts.

A currently unsolved problem with communicating artifact events is which data and metadata to include in the broadcasted event messages. “I don’t know what meta information should be published with these changes, but it might also be interesting to know a bit more about this artifact”. This is an important problem to solve in order to be able to connect an artifact created event sent by a build tool with an artifact published event broadcasted by a separate service for storing artifacts.
Broadcasting and listening to Eiffel events that handle created and published artifacts removes the need to poll and manually check if an artifact is available. The system can be more reactive and automatically be notified when an artifact is available. This could lead to optimizations in how test suites are dynamically created and how tests are run. In addition to the experiments with constructing test suites dynamically, a future goal is to expand the functionality to offer an indication of confidence levels, but it is not known how this should work in practice.

It is clear that a key use case that is enabled by capturing and broadcasting trace links is to trigger activities in the CI pipeline by acting on broadcasted events. In other words: to use Eiffel events to drive the pipeline.

Another benefit of broadcasting artifact events is that developers would not have to rely on manually checking if libraries used for testing have been updated every time they need to run their tests. Instead a service could be built that automates the process and notifies developers when a library update is available.

4.1.2 Interview B

The second interview was a joint interview with two experts: a Firmware Ecosystem Engineer (Interviewee B1) and a consultant Software Engineer (Interviewee B2). Since Interviewee B2 has started working at the company only recently, most of the background information and descriptions came from Interviewee B1. Interviewee B1’s work tasks spin around making improvements in the development process and tooling. He has earlier been actively involved in the integration work. Both interviewees’ answer are summarized together.

4.1.2.1 Current state

The department decided to move towards CI/CD, and eventually CDP and DevOps a while ago. A lot of progress has already been done in recent years, e.g. integration speeds have improved significantly, but there is still a lot to work on. One of the main goals in the context is to speed up the integration even further, effectivize development and improve the integration process. Interviewee B1 and his team look at the DevOps chain and how the components in it and around it move towards CI, CD, CDP and DevOps, as well as the ecosystem around it.

The department places a lot of focus on metrics and the ability to aggregate and fetch them automatically. Nowadays it is hard to understand what data there is and which of it is usable and/or necessary. In certain cases there might too much data generated, and in others very little. Often, it is possible to do “a deep dive” and extract data on a certain aspect for the current situation, but overall such information is not easily available in a constant and systematic manner. There are metrics on how long it takes to fix a bug, for instance, but the time it takes to reach a customer is so long that it is hardly worth measuring. It is unclear how much time it takes for a change or upgrade to reach the customer. The engineers know it is several months, which means that development cycles are too long, which is another problem.
There are, however, some metrics, e.g. around Gerrit commits times, that are done automatically, but they are not very systematic, either, and are not used much. Issues are also measured automatically, i.e. issue and bug tickets in the company’s own issue tracker. Some time metrics on certain activities after the code has been pushed to master or on integration times are available, but gathering, fetching and presenting it implies a lot of manual work.

Documentation is often discussed in the context of internal requests for features and an in-house client. Requested features are first described on a high abstraction level in the product information management tool, and once a product feature has received an official name, it goes through an acceptance process. These features specifications are then stored in a database, which can be thousands in number, but engineers in the department do not get notified or retrieve this information automatically; one has to fetch it manually. When the development process starts, there are no links between the high-level descriptions from the product information management tool and, for instance, Gerrit reviews or feature tickets, and at the moment it is not clear how to create such links in practice.

In the steps following requirements specification, lack of appropriate tools and traceability solutions is a also challenge. Right tooling is a crucial need and is tightly related to changes in the process and culture. The mindset at the company has already changed a lot, but engineers work in different tools and the choice is not centralised. The work tools choices are scattered, so it is hard to get a comprehensive picture of what is going on. For instance, there are several different issue trackers that are used at the company. There are already some in-house solutions to make trace-leaving easier, but overall there is a distinct lack of appropriate tools that would make it possible for developers to generate data about their workflow in an automated manner. For now, trace creation is manual and has to be encouraged. Developers are asked to manually write feature tickets themselves, and when they integrate code, they are supposed to inform others about the integration via traditional channels: “[s]o we need to encourage them to manually do it. [...] Who told you for instance to make this feature, where is it coming from? This information is quite obscure today, in my opinion.”

Further, developers might struggle to figure out on what abstraction level the information should be presented and why it is needed:

The fact is that it is bothersome for a developer to [...] make this extra step that will describe what he/she does. Shall I do it for every commit, or shall I describe the whole thing in its entirety? It is going to take several weeks to get to the entirety, why do you need to know it now? I have just started on the project, for instance. Or maybe it’s not [even] relevant for the customer.

There is therefore a definite lack of information on the development activities before the commit stage. Everything that happens before the code is pushed to the shared repository is an opaqueness:

Basically everything that developers do till a certain point is quite obscure today. There are many developers working in different teams and so on, it is quite difficult to see. Before a Gerrit commit is created we do not know basically
anything [...] about how long the developer has been working on the project, what is it that is going on. So [after a Gerrit commit] time-counting starts for me, but before this point it’s very difficult to work.

Further, only big features get documented, while there a lot of small undocumented changes that might be relevant as well, “but it doesn’t come for free today”. When a feature is being developed, there is no information about it, and once a feature has been finished, there is no way of knowing that it has been delivered and is available.

It is also difficult to get information on the right abstraction and detail level. For instance, release notes management is a very critical need. Today, the developer or product specialist has to go into issue tracker and write “RN” on the bug to make it clear that the bug fix will be part of the next release, which might happen a lot later. There are Git commit messages, which describe feature development and bug fixes, but this information is not available in an easy to use way and is not good material for product specialists who write release notes. Product specialists thus go through tickets continuously, decide which tickets are relevant to have in the release note, and then compose a release note itself. There is a tool for release note management that can fetch information from tickets, but overall it is mostly a manual process done by humans, and it can be error-prone.

The problem with the right abstraction and granularity level for different user needs is also found in the test results domain. Some users, e.g. the interviewees’ department might need only a final report, and generalized test results are used in daily builds. In other cases, the test results have to be presented on a more fragmented level. For instance, if the test results will drive the pipeline, or for the daily work of the QA department:

It is very dependent on what kind of user [it is]. If you work as a tester then you maybe want to know maybe not that you started a test suite and then it went wrong a week later but maybe you want to know that half an hour later it started to fail and which test case it was.

The data usability challenges can thus be summarized as an overall difficulty to get a full picture of different activities, relationships between them, corresponding time metrics, and tools used at the organization, with the detail level suitable for specific users. This challenge is accompanied by the limited ability to visualize relevant development flows, time metrics and available data in general.

*Eiffel* is not used at all in the department at the moment, but migration to it is planned in the future. There are teams and experts who are working on it.

### 4.1.2.2 Desired state

There is an expressed desire to become more data-driven and do all-encompassing metrics with the help of traceability:

We really want to become more data-driven than feelings-driven to make it possible to see objectively what adds up/is tuned up. It has been a challenge here when someone says, ‘How much time does it take to do an integration?’ So being data-driven gives the possibility to work more objectively, more systematically, and then decide whether a given change is worth it, for instance. [Data] can also
be used to propose a change, to really be able to prove it with objective facts rather than one's opinion.

It will be critical in the future to make it possible to work with incoming errors right after as they occur, fetch information in an easier way, have all development and testing activities thoroughly documented, and be able to easily see which code changes contribute to a feature, as well as have a full overview over development activities. Established traceability is a prerequisite to all these goals:

Everything around development itself that one can measure a little to be able to see what takes time — this is in fact the question we have been especially interested in in the recent years. What is it that takes time, why it takes time and if we can shorten it down, and thus empower the developer. And to be able to empower the developer, there have to be present suitable tools so that one gets a chance to do this work. So that it is less manual work and more automation in it.

An accompanying goal is thus to introduce new, better tools that make CI/CD and traceability easier. For instance, it is planned to move the organization to an issue and feature tracker which is well-suited for feature traceability, where developers will be able to easily refer to a user story or task when they start developing. This way, developers will generate traces automatically without having to make extra steps. Adopting better tools is an important step in improving the overview of the development process: “... this way it becomes more natural, and you avoid this whole extra step of ‘You have to [document your work] because...’ because [now it is present] naturally in the tools we use. It is going to surely make it easier”.

New tools’ inherent ability to facilitate the integration process can actually be used to accelerate the shift in developer habits: “[i]t is possible to sometimes use the tools actually as an argument to make a change easier. That is, one says ‘Why [use it]?’ -- because if everyone works this way, then we get this gain at the end. People understand this. It is more difficult to just say: ‘Change it, no objections.’"

There are therefore numerous opportunities with Eiffel, according to the interviewee. It is expected to help establish and improve traceability, visualization and metrics, the three most important needs identified during the interview. For instance, it will be possible to track the whole development process, track when software reaches the customer and what tests have been done, and be able to track the initial product feature description all the way to a specific release, and vice versa. For visualization, Eiffel-powered development flow visualization tool is planned, which will make it possible to visually trace a commit through the development flow. It is also expected that in the future it will be possible to measure the time it takes a commit to go from one point to another and even connect triggers to it, and Eiffel-assisted traceability is an important precursor to triggers.

It is also expected to aid in aggregating test results and connecting them to earlier artifacts, as well as in gathering information from the whole organization without knowing how people work and what exactly they do:

If you want to have the test results, you go to QA because you know that they do tests, but maybe you do not know that the developer organization have also done
tests on the same build. It would be possible to get this information with the help of Eiffel, if both are [...] careful with posting an event, "I have tested something on this build". [...] It really gives you possibilities and available information in a completely different way.

4.1.3. Interview C

The third interviewee is a Senior Engineer who works on tools and services that are used by other developers at the company, primarily for CI, code versioning and metrics. His role involves a lot of work with open source projects.

4.1.3.1 Current state

The Interviewee's department develop tools and build CI chains for the rest of the company. Achieving better CI, and especially automatic testing in pipelines, is a very important goal. The final target is to achieve CD, i.e. to be able to integrate, do tests, and deliver and deploy software without having to take down the service being updated, as well as to be able to iteratively do releases and quick rollbacks, if needed.

In this respect, metrics are very important, but there is still a lot to work on. To get good metrics, and especially to know what to measure, is not always that easy, especially in the endeavour to shorten integration and release times and improve code quality. The final goal at the department is to measure everything, and a great deal of metrics are already generated today, but this data is not used efficiently or much enough. The challenge lies in not just collecting data but also in knowing "how to twist and bend metrics" to make them usable and understandable, e.g. to transform the data to be able to extract KPIs that are valuable for understanding what parts in the CI chain need most attention:

But it’s a whole science to [following the flow all the way] from the point when one starts to write their first code snippet to the point when [the code] is reviewed, integrated, built and released. All these steps have to be measured and you have to understand what you are measuring. This is something we need to become better at.

The department tries to keep to microservices, which generate a lot of data "almost for free", but sometimes it can come in the wrong format or be missing at all, so there is currently no easy way to summarise and interpret it efficiently:

We have a lot of data, but we can spend more time on interpreting the data and see what time gets spent on. But that said, it’s not that we don’t use data, we use data quite a lot, but I believe that there is more to get out of it.

The work at the department is quite siloed from the rest of the company as the department’s direct customers are other developers, not external customers who buy cameras: "We work quite, how do you say, one-dimensionally, where we have one axis and it’s very easy to follow [events]. We don’t have the same problem that firmware developers have, where it splits up in all directions." Thus, the interviewee does not experience acute traceability problems in his daily work as most of needed relevant information can be retrieved from tools that the engineers use, such as Git and
Gerrit, and his work is not directly concerned with the software that goes through external customer-facing pipelines: “We have a very linear workflow. We don’t have many release branches — or at least in that type of services I develop — so it is not especially difficult for me.”

Even though traceability information can be easily retrieved from the tools used, doing it is manual work. A big advantage of well-established traceability is the possibility to visualise development events flow in real time, e.g. where in the pipeline a given commit is at the moment, and lack of a visualization solution is an important obstacle at the moment:

I think that we have quite a lot of event data about what’s happening. But at the same time, it’s a big problem for us too because our events are quite detached: we have a [Git] commit that became a Gerrit event, and then we maybe make a build, then we make a release -- those events are not exactly linked with clear IDs that would connect them into chains, so it’s quite difficult to follow traces all the way.

There are well-working ways to visualize data for certain scenarios, though, e.g. for debugging, but only when one knows what to look for, and it is still manual work based on time-based event flow: “Ok, we uploaded it here, and what happened then around this point in time? Aha, it was merged. And so on.”

Concerning the work of other developers at the company, the interviewee commented that it is important to be able to promptly react to bugs that, for instance, an external customer discovers, understand how they ended up in the release, exactly what changes are in that releases, how it was tested and approved, and in which products the bug ended up. This is needed to be able to release bug fixes and understand who and how got influenced by it:

And it’s also incredibly important to minimise development cycles, to understand where we get stuck in terms of development: is it that review times, build times, is it to do tests, is it when it’s about to be released? All this, both development time and traceability of bugs and such, is extremely important, I would say.

4.1.3.2 Desired state

Currently, Eiffel is not used at the department but is planned to be introduced in the future. One of the main benefits that it is expected to bring is visualization of events in pipelines. An adjacent problem that Eiffel is expected to help solve is modularity of pipeline components, where the pipeline elements will be decoupled into microservices that communicate via a well-defined protocol. It would then be possible to easily add new components or replace existing ones, as opposed to having to modify or build big monoliths from scratch:

If you have Eiffel, you are be able to simply remove [a building block] and put another one in its place, it’s pretty cool. Well, not today, but I hope that Eiffel will become part of my development process, exactly because we will then be able to start working with such things in an easier way.
4.1.4 Summary of interviews

Table 4 below summarizes the interview results and provides answers to RQ1 and RQ2. A quick overview shows that many issues and wishes were raised in all of the interviews, such as full visualization of the pipeline, end-to-end traceability of pipeline activities, better metrics and documentation, and better tools that have inbuilt support for traceability. Automation of the pipeline, testing and notifications about pipeline events was also a common theme. All of the interviewees expressed the desire to have Eiffel adopted for better traceability and visualization, and in a more distant future let Eiffel drive pipelines via events as triggers of activities.

Table 4. Interview results as per the current and desired states of traceability

<table>
<thead>
<tr>
<th>Current</th>
<th>Interview A</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Experiments with Eiffel</td>
<td>✓ Adoption of Eiffel</td>
<td></td>
</tr>
<tr>
<td>✓ Experiments with dynamic test suites by analyzing traceability data</td>
<td>✓ Eiffel triggers: automation of the pipeline</td>
<td></td>
</tr>
<tr>
<td>✓ Traceability data stored for recent tests; used for troubleshooting</td>
<td>✓ Eiffel events for test results confidence level.</td>
<td></td>
</tr>
<tr>
<td>✓ Visualization of test results</td>
<td>✓ Visualization of the whole pipeline</td>
<td></td>
</tr>
<tr>
<td>✓ Many tools → obstacle to traceability</td>
<td>✓ Tools: shared communication protocol</td>
<td></td>
</tr>
<tr>
<td>✓ Links between tests and features</td>
<td>✓ Links between tests, features and code</td>
<td></td>
</tr>
<tr>
<td>✓ No links between features and source code → complicated troubleshooting</td>
<td>✓ Automated notifications of pipeline events</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current</th>
<th>Interview B</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Eiffel is not used</td>
<td>✓ Adoption of Eiffel</td>
<td></td>
</tr>
<tr>
<td>✓ Inappropriate tools: too many, no traces → manual work</td>
<td>✓ Eiffel triggers: automation of the pipeline</td>
<td></td>
</tr>
<tr>
<td>✓ Data/metrics: systematicity, availability issues → manual work</td>
<td>✓ Tools: trace automation → no manual work</td>
<td></td>
</tr>
<tr>
<td>✓ Poor tracing of developers’ activity</td>
<td>✓ Visualization of the whole pipeline</td>
<td></td>
</tr>
<tr>
<td>✓ No links between product description and feature tickets/reviews</td>
<td>✓ Becoming data-driven</td>
<td></td>
</tr>
<tr>
<td>✓ Information abstraction, availability and granularity issues</td>
<td>✓ Metrics/data: automatic generation and easier retrieval; better and more</td>
<td></td>
</tr>
<tr>
<td>✓ No notifications of new features specs. → manual work</td>
<td>✓ Full tracing of all development and testing activities</td>
<td></td>
</tr>
<tr>
<td>✓ Fast feedback for bugs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current</th>
<th>Interview C</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Eiffel is not used</td>
<td>✓ Adoption of Eiffel</td>
<td></td>
</tr>
<tr>
<td>✓ Poor pipeline flow tracing and visualization</td>
<td>✓ Visualization of the whole pipeline</td>
<td></td>
</tr>
<tr>
<td>✓ Many metrics but not always usable</td>
<td>✓ Modularity of tools → easy rolling updates and replacements.</td>
<td></td>
</tr>
<tr>
<td>✓ No links between source code, tests, final releases/products</td>
<td>✓ Metrics of everything</td>
<td></td>
</tr>
<tr>
<td>✓ No acute traceability issues</td>
<td>✓ Testing automation</td>
<td></td>
</tr>
<tr>
<td>✓ Traceability data found in tools → manual work</td>
<td>✓ Identification of weak spots</td>
<td></td>
</tr>
</tbody>
</table>
4.2 Implementation results

4.2.1 Plugin design

The implementation of the prototype consisted of writing an Artifactory user plugin [32] in the Groovy programming language. The plugin is open source and can be found on GitHub [33].

![Fig. 7. Schematic system overview of the plugin prototype](image)

The plugin is to be placed on an Artifactory instance. The library implementing a RabbitMQ connection [34] is written in Java and is placed as a separate Java archive file in Artifactory, too (see Fig. 7). For developing the prototype, the Artifactory Pro edition was used, which was run as a Docker container.

The plugin was designed to both read incoming Eiffel messages and send its own. The callback method `storage { afterCreate { } }`, which is a part of Artifactory’s framework for user plugins, is thought to react to all upload events and get metadata of the uploaded artifact with an `ItemInfo` object. The properties of the artifact can then be extracted with the Artifactory Query Language (AQL) and an `EiffelArtifactPublishedEvent` can then be formed and sent on the RabbitMQ message bus.

By the end of this research, the plugin contained the ground for all the functionality mentioned above. That is, it contained all the above mentioned methods, could potentially receive and send events and react to Artifactory storage events. However, the parsing and sending of actual Eiffel events to make the plugin fully functional was not implemented due to the lack of information that plugin could receive from the Eiffel plugin in Jenkins. The following chapters describe what information was needed for the plugin to be fully functional and how the events coming from Jenkins
need to be modified to make it possible for our plugin to act upon them and send its own Eiffel events. Once corresponding modifications to the Jenkins plugin are made, our plugin can be developed further to complete the relevant functions.

4.2.2 Eiffel protocol implementation

The EiffelArtifactPublishedEvent (see [35] for full description) found in Schema 1 has the following possible fields: red represents obligatory fields, green optional fields, and orange represents fields that are obligatory if their optional green parent is present.

1. **data:**
   1.1. **locations:**
      1.1.1. `type`: String (ARTIFACTORY/NEXUS/PLAIN/OTHER)
      1.1.2. `uri`: String (where the artifact can be retrieved)
   1.2. `customData`: Object []

2. **links:**
   2.1. `type`: String (ARTFACT/CAUSE/CONTEXT/FLOW_CONTEXT)
   2.2. `target`: String (UUID(s) of the corresponding event(s))

3. **meta:**
   3.1. `id`: String (generated UUID)
   3.2. `type`: String (event type name, i.e. EiffelArtifactPublishedEvent)
   3.3. `version`: String (version of the event type)
   3.4. `time`: Integer (in UNIX Epoch time milliseconds)
   3.5. `tags`: String[]
   3.6. `source`:
      3.6.1. `domainId`: String (domain that produced event)
      3.6.2. `host`: String (hostname)
      3.6.3. `name`: String (name of the event sender)
      3.6.4. `serializer`: String (identity of serializer software)
      3.6.5. `uri`: String (The URI of, related to or describing the event sender)

3.7. **security**:
   3.7.1. `authorIdentity`: String (distinguished name)
   3.7.2. `integrityProtection`:
      3.7.2.1. `alg`: String (cryptographic algorithm used to sign event)
      3.7.2.2. `signature`: String
      3.7.2.3. `publicKey`: String
   3.7.3. `sequenceProtection`:
      3.7.3.1. `sequenceName`: String
      3.7.3.2. `position`: Integer

Schema 1. EiffelArtifactPublishedEvent with obligatory and optional fields marked

Artifactory defines its own query language called AQL (Artifactory Query Language). The artifacts pushed by Jenkins to Artifactory can be reached in the following manner, following the AQL architecture. When a build completes, Jenkins adds a corresponding artifact in the build domain (the other primary AQL domains being item, entry, promotion and release, see Fig. 8). The build description holds the description and metadata of the build itself. One of the fields in the description,
accessible in Build Info JSON on the Artifactory’s web interface, is the subdomain field **modules**, which includes a list of **artifacts** that the build produced. The corresponding artifacts are pushed to a corresponding repository (and belong to the domain **item** and subdomain **artifact**). In its metadata, the artifact has references to the build and build number that produced the artifact. It should be noted that the AQL architecture does not map directly onto the actual file structure in Artifactory but only describes the way artifacts can be accessed using AQL.

![Diagram of AQL domains](image)

*Fig 8. AQL domains that can be used in queries [36].*

As of time of writing, the Eiffel plugin for Jenkins sent out **EiffelArtifactCreatedEvent**s of the following form (Schema 2, for full description and complete list of fields see [38]):
1. **data:**
   1.1. **identity:** purl String of the form
       "pkg:job/PROJECT/job/SUBPROJECT/job/[...]/job/[build number]/artifacts/[artifact name]@[build number]"

2. **links:** [empty]
3. **meta:**
   3.1. **id:** "[generated UUID]"
   3.2. **type:** "EiffelArtifactCreatedEvent"
   3.3. **version:** "2.0.0."
   3.4. **time:** [Integer in UNIX Epoch time milliseconds]

**Schema 2. Current EiffelArtifactCreatedEvent schema the plugin for Jenkins**

Based on the current setup of Jenkins and Artifactory at Axis Communications, the **EiffelArtifactCreatedEvent** should include the fields found in Schema 3 in order to uniquely connect **EiffelArtifactCreatedEvent** sent by the Eiffel plugin on Jenkins with the corresponding build and artifacts that Jenkins uploaded to Artifactory. The proposed fields in Schema 3 are marked in bold, with fields belonging to specific variants (which will be discussed later) prepended by blue numbers in square brackets:

1. **data:**
   1.1. **[1]** identity: purl String of the form
       "pkg:job/DIR/job/SUBDIR/job/[...]/job/JOB_NAME/[build number]/artifacts/[artifact name]@[build number]"
   1.2. **fileInformation**
       1.2.1. **[2, 3]** name: "[the filename of the artifact created, e.g. new_build.zip]"
   1.3. **[2]** customData:
       "key": "uri",
       "value": "https://[jenkins-server-hostname]/jenkins/job/DIR/job/SUBDIR/job/[...]/job/JOB_NAME/[build number]/"

2. **links:** [empty]
3. **meta:**
   3.1. **id:** "[generated UUID]"
   3.2. **type:** "EiffelArtifactCreatedEvent"
   3.3. **version:** "2.0.0."
   3.4. **time:** [Integer in UNIX Epoch time milliseconds]
   3.5. **source:**
      3.5.1. **[1]** host: "[https://[jenkins-server-hostname]]"
      3.5.2. **[3]** uri: "[the URI of the jenkins project that produced the event, including the build number, e.g. https://[jenkins-server-hostname]/jenkins/job/DIR/job/SUBDIR/job/[...]/job/JOB_NAME/[build number]/]"

**Schema 3. Proposed Eiffel schema for EiffelArtifactCreatedEvent for Jenkins plugin**

The information needed to uniquely identify an artifact and connect it to a build and Jenkins job are:
1. The Jenkins server hostname.

2. The Jenkins build job path. i.e. /job/DIR/job/SUBDIR/job/[...]/job/JOB_NAME/
   It then becomes the build name DIR :: SUBDIR :: [SUBDIR ::] :: JOB_NAME on Artifactory. For searching, the formatting as it is formatted in the job path should be used as it always follows the same pattern, while the build name can be subject to change and/or be specific to a project, e.g. slashes instead of double colons.

3. The build number.

4. The artifact filename.

Points 2-4 are already present in the identity field, i.e. the job/build name, the build number and the artifact name, the combination of which is unique to each artifact produced by a given Jenkins server. The Eiffel plugin for Jenkins sends an event for each created artifact, meaning that one Eiffel message will correspond to one artifact in Artifactory. In order to link the event with the message, it is thus possible to parse the identity field and extract the necessary data, and use it in combination with the Jenkins server hostname.

The proposed fields would make it easier and faster to find the artifact belonging to a build. The string that contains all the information needed to uniquely identify the build and that is present in exactly the same form in a build's metadata on Artifactory (“uri” in Build Info JSON) is the build source URI. It has the following structure:

https://[jenkins-server-hostname]/jenkins/job/DIR/job/SUBDIR/job/[...]/job/JOB_NAME/[

The following variants of the schema can be used to further identify the artifact that the Jenkins job produced:

[1] data.identity is parsed to retrieve the build name, build number and the filename. meta.source.host holds the Jenkins server hostname that ran the job.

[2] data.customData holds the build source URI. data.fileInformation.name holds the filename.

[3] meta.source.uri holds the build source URI. data.fileInformation.name holds the filename.

Once all the required information is present in the Eiffel event coming from Jenkins, one AQL query done from the Artifactory plugin in combination with the storage { } callback native to Artifactory itself is enough to identify the links between the received event and corresponding build and artifacts uploaded to Artifactory and send an EiffelArtifactPublishedEvent.

For instance, for schema variant 2 and 3, the query would be:

```javascript
items.find({"name":"[artifact_filename]",
      "artifact.module.build.url":"[build_uri]"})
.include("name","repo","path")
```

The resulting EiffelArtifactPublishedEvent [35] sent by the Artifactory plugin would include only the obligatory (red) fields from Schema 1.
The limitations and implications of choosing one of the schema variants are discussed in Chapter 5.1. It should also be noted that these are only the results of an analysis regarding the way these events and artifacts can be connected and what the event would eventually look like. At the time of writing, the functionality itself was not implemented in the plugin, the reasons for which will also be discussed in Chapter 5.1.

5 Discussion

While conducting the research for this paper it became clear that the challenges of traceability can be divided into two different areas: (1) technical implementation and (2) organizational strategy. In this discussion we use the findings from the literature study and the case study to outline the important aspects of both aforementioned areas, starting with the technical implementation before presenting the organizational challenges.

5.1 Technical implementation

A fully functioning technical implementation that enables end-to-end traceability in a pipeline using the Eiffel protocol is a vast undertaking with several layers of software components at different abstraction levels. A high level outline of components and their interactions in an Eiffel-based traceability solution is presented in the Sepia Eiffel Protocol Implementation Architecture. This paper focuses on the most fundamental component: the event author.

5.1.1 Plugin development

A fundamental component of an architecture implementation based on the Eiffel framework is the event author. The event author is responsible for mapping an activity in the pipeline to a specific Eiffel event and broadcasting the event to a message broker. Some event authors might also listen for other Eiffel events and based on the event trigger an activity in the pipeline.

By implementing event authors as plugins for each and every pipeline actor, progress towards several of the desired states found during this research is made (see Table 5). This is also a prerequisite for all other desired states since without an infrastructure in place to communicate pipeline activities as Eiffel events, there is no traceability information to make use of.

<table>
<thead>
<tr>
<th>Desired State</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable communication between all the different pipeline actors.</td>
<td>Develop plugins for pipeline actors to enable communication using the Eiffel protocol.</td>
</tr>
<tr>
<td>Automation of the pipeline using traceability information.</td>
<td>Develop plugins for pipeline actors that listen for Eiffel events and based on the event trigger an activity in the pipeline.</td>
</tr>
</tbody>
</table>

Table 5. Example of desired state goals and corresponding solutions.
In this study, the issue of technological diversity in a continuous integration pipeline as an obstacle for traceability has been mentioned both in previous literature and subsequently echoed by the interview subjects in the case study. In order to achieve traceability in a continuous integration context it is crucial to enable communication between all the different actors in the pipeline.

The Eiffel protocol addresses this challenge by specifying a protocol for communication. By developing a plugin that implements the Eiffel protocol for each and every actor in the pipeline every pipeline activity can be communicated as a specific Eiffel event.

During this case study the use-case of driving the pipeline using traceability information was made explicit by all stakeholders and is one of the main motivations behind adopting the Eiffel framework as a traceability solution.

Eiffel addresses this use-case by being a part of the actual flow of the continuous integration pipeline. Eiffel events are generated in real time by the actors in the pipeline. The events themselves are designed to correspond with pipeline actor activities. This allows actors to consume and act upon events broadcasted by other actors in real time.

At Axis today there has already been efforts to implement a subset of the Eiffel protocol as a plugin for Jenkins. This plugin is used in production today and broadcasts Activity Canceled, Activity Finished, Activity Started, Activity Triggered and Artifact Created events. There is a formal strategy to develop a plugin for every tool in the pipeline. Both the Eiffel documentation and interview results support this strategy.

5.1.2 Event publishing

When developing a plugin one needs to take into consideration where the responsibility for event generation, validation and publishing lies. Direct publishing means that the responsibility lies with the plugin itself. In order to broadcast events to a message broker, in this case RabbitMQ, the plugin has to act as a fully fledged RabbitMQ client. In order to generate valid events the plugin needs to implement a model of the Eiffel events in the required programming language, and it also needs to serialize them to a valid JSON document that follows the Eiffel protocol specification. This is further complicated by the fact that Eiffel events might have different specifications based on which version of the protocol an actor is using. Direct publishing is the event publishing method used today at Axis when developing plugins that implement the Eiffel protocol.
This method is reasonable in some use-cases, specifically if there is a small amount of plugins that need to be developed and there already exists a mature RabbitMQ library for the language used for plugin development. The main benefit of direct publishing is that it does not require a separate service to manage. However, while developing the prototype for this thesis several drawbacks of this method became obvious.

First of all it is not trivial to implement a library level RabbitMQ integration without previous experience of how RabbitMQ works, and this is provided that a mature library exists for the programming language used to develop a plugin for a given actor. If several different developers need to come up with their own solution for communicating Eiffel events to the message bus, development will be very time consuming. While modelling Eiffel events is not particularly difficult, it does lead to a large amount of boilerplate code across several plugins. Serializing the model to JSON and in particular validating the JSON document to make sure that it is complying with the Eiffel event schema is definitely non-trivial. In addition to this, if there is a need to link events together each plugin would need to interface with a persistence service, typically some kind of database, in order to retrieve the link event.

Direct publishing does not separate concerns in a reasonable way, a plugin should ideally only be concerned with mapping an actor’s activities and content to Eiffel events and if needed act upon other broadcasted Eiffel events. This makes plugins faster to develop and easier to maintain, understand and change.

The disadvantages of direct publishing mentioned above are addressed by assisted publishing. The drawback is that it requires a separate service that provides event publishing and this service has to be maintained.
Based on the insights gathered from developing a prototype during the case study of this research, it is recommended to carefully examine the pros and cons of the two different publishing methods at an early stage and decide which is better suited for the organization. In general we believe that the benefits of assisted publishing greatly outweigh its disadvantages and helps facilitate the development of plugins by letting developers only focus on the main concern of mapping an actor’s pipeline activities to Eiffel events.

The Eiffel framework offers a service called RemRem that facilitates assisted publishing, but the state of the project is still in progress.

### 5.1.3 Artifactory plugin prototype

As was mentioned in Chapter 4.2, a number of obstacles were met when implementing the Eiffel functionality in the plugin prototype.

Bugs and unexpected behavior are part and parcel of the software development process. Due to possible threats to infrastructure critical data that the development could cause to the production environment, the prototyping was done on a separate Docker container running its own instance of Artifactory Pro. The plugin therefore could not act on the real uploaded artifacts on the production Artifactory, and storage events were simulated in order to test the `storage { }` callback. This implies that even if the plugin could receive messages sent by the Eiffel plugin for Jenkins, it was not possible to verify the general correctness and performance of the plugin in the context of a production pipeline Chapter 4.2.

It should be noted that it would still be possible to verify the plugin without having to place the plugin on the production Artifactory instance via listening to `EiffelArtifactCreatedEvents` and requesting information from Artifactory using its REST API and AQL queries. However, there is no guarantee that when the Eiffel plugin for Artifactory receives a `EiffelArtifactCreatedEvent` the artifact itself is already available. To check whether the artifact is uploaded, the `storage { }`
callback from within Artifactory needs to be used, which is not possible due to the aforementioned lack of permissions. Without it, the plugin might end up “missing the target” if the artifact is not available on the production instance yet or running AQL queries multiple times, e.g. in a loop, until the artifact is available, which might impact the general performance of the plugin and overflow the production instance with requests.

Further, the analysis of the link between the Eiffel message contents sent by the Jenkins plugin and artifacts structure and metadata contents was mostly done on the old data available on the staging version of Artifactory. The access permissions model on Artifactory [37] requires one to have administrator privileges in order to execute the AQL queries that were necessary for experimentation and analysis during the development. Queries against the build domain on the production instance were not possible as we could not be granted elevated privileges for it, but it was possible to get administrator access and test all necessary AQL queries on the staging instance as it is partly kept exactly for experimentation purposes. The correctness of resulting queries against the item domain following the proposed schemas and artifact identification steps was however verified to work on the production data, with necessary information (e.g., build.uri) taken from the production version of Artifactory via its web interface, where the access restrictions for viewing do not apply.

5.1.4 Eiffel protocol schema evaluation

The schemas proposed for EiffelArtifactCreatedEvent, however, deserve special discussion. As mentioned in Chapter 4.2., three possible combinations of fields can be used to link an EiffelArtifactCreatedEvent to a build and an artifact in Artifactory, each coming with its own suitability reservations.

*Variant 1:* “data.identity is parsed to retrieve the build name, build number and the filename. meta.source.host holds the Jenkins server hostname that ran the job.”

As already mentioned, data.identity holds the “[t]he identity of the created artifact, in purl format” [38], where all the necessary information is already present. However, in order to retrieve it, it is necessary to parse the PURL string and extract the relevant substring, which could be used to do regex matching against build.uri found in builds’ metadata. Both parsing and regex matching to extract the build name and number can be error-prone and lead to unexpected results, as well as cumbersome implementation. The same goes for the filename found at the end of the PURL.

Since theoretically two Jenkins servers can make the same build with the same build number producing artifacts with identical filenames (e.g. if one is used for production and the other one for experiments), the PURL only is not enough, hence the need for the Jenkins server hostname and the meta.source.host field. As stated in the Eiffel documentation [38], the section is:

> A description of the source of the event. This object is primarily for traceability purposes, and while optional, some form of identification of the source is **HIGHLY RECOMMENDED** [sic]. It offers multiple methods of identifying the source of the
event, techniques which may be select [sic] from based on the technology domain and needs in any particular use case.

The data section, where the PURL is found describes the object itself: “contains all fields specific to the event type – the payload of the event – including trace links to non-Eiffel entities” [39]. Meanwhile, the meta section “contains meta-information describing the event: when it was created, where it came from, its type et cetera” [40]. This means that in this case, event section responsibilities are mixed because fields that are meant to describe the event itself are involved in the description of the event payload.

Variant 2: “data.customData holds the build source URI. data.fileInformation.name holds the filename”.

The Eiffel documentation states [41]:

\[ \text{data.customData} \] is an optional member of all Eiffel events. It is an array of key-value pairs, where the value can be of any type. This grants users a high degree of freedom in including their own, custom content in messages without any disruption to the rest of the syntax: the uninitiated [sic] reader is free to simply ignore any \[ \text{data.customData} \] contents without fear of adverse effects.

It also states that this section should be used “when a highly specific and/or localized use case calls for data members not included in the Eiffel vocabulary” [41]. Even though the introduction of customData satisfies the content needs for this case, it violates the design guidelines put forward by the authors of the protocol since a URI linking back to the source of a build is not a highly specific or localized use case, proven by the event schema already provides a field for it, i.e. meta.source.uri.

The second proposed field, data.fileInformation.name, is “the name (including relative path from the root of the artifact) on syntax appropriate for the artifact packaging type” [38]. In Variant 1, the filename is included in data.identity, which would still be present in Variant 2 and it would still be possible to parse it to extract the filename, but since the protocol already provides a specialized field for the filename in the event schema, it would be best to keep as a separate field and avoid string parsing. Same could be said for Variant 1, i.e. even if this information is present, the first variant of the proposed schema could be augmented with this field.

Variant 3: “meta.source.uri holds the build source URI. data.fileInformation.name holds the filename”.

Like Variant 2, this variant also makes use of data.fileInformation.name for the filename and relies on the source URI being in the event as well. In this case, however, it is put in the meta.source.uri field, which is “the URI of, related to or describing the event sender”. This seems to be the right place for the URI, but with the same reservation as in Variant 1: the information used to identify the artifact, i.e. the payload of the event, is taken both from the data section, which is directly designed for the payload, and meta section, which is designed for the description of the event itself, thus violating the boundaries of the two sections’ responsibilities.
The conclusion that can be drawn after analyzing the three variants is that none of them suits the need well without going against the schema’s areas of responsibility or introducing custom fields and thus not making effective use of the protocol. It can be theorized that the problem could be solved by using the Eiffel framework’s own toolbox: *Eiffel Intelligence*, a data aggregation and analysis solution [27], a part of *Eiffel Sepia*, the Eiffel protocol implementation architecture. In this case, Variant 3 would be the best choice as Eiffel Intelligence allows an actor to subscribe for events and gather needed information from an event’s different field, regardless of the assigned section responsibility [42]. As of time of writing, however, Eiffel Sepia had the status of “in progress” and Eiffel Intelligence “functional, but not mature”, making their introduction at an early stage in a company that is only starting to adopt Eiffel a questionable choice.

An important reservation to take up is the fact that the development and discussion of the Artifactory plugin has only been concerned with events sent by Jenkins. However, Artifactory hosts a great number of artifacts coming from different actors. For instance, builds done by Gradle do not include the URI in the build info, and the described artifact identification process used for Jenkins will not work. Addition of other combinations of fields will be needed in the *EiffelArtifactCreatedEvent*, e.g. the name service account that Gradle uses to upload artifacts to Artifactory in combination with the timestamp or other fields.

The discussion has only covered the Eiffel event that the plugin for Artifactory relies on. As it was mentioned earlier, the prototype of the *EiffelArtifactPublishedEvent* is thought to include the obligatory fields only. As it is obvious from this discussion, the needs of the pipeline actor listening for events clarify the exact contents of these events, as in this case understanding what is needed to connect builds and artifacts on Artifactory sheds light on what data should be included in Eiffel events sent by the Jenkins plugin. The same logic is thought to apply for Artifactory: during the development of Eiffel plugins and tools for pipeline actors that depend on events sent by Artifactory, it will start getting clearer what data and fields are missing or misplaced in *EiffelArtifactPublishedEvents*.

This leads to a suggestion of the technical process on the way to implementing Eiffel in an organisation. Since the needed contents in an Eiffel event becomes clearer once one inspects the data exchanged between two pipeline actors and what Eiffel event fields need to be filled in, the development process could go as follows (illustrated in Fig. 11):

1. Implement a bare minimum Eiffel plugin for Actor 1 that sends events with obligatory fields only.
2. Do the same for Actor 2, which is going to depend on Eiffel events from Actor 1.
3. Define the contents or contents form needed in events from Actor 1.
4. Modify the event structure sent by Actor 1.
5. Add Actor 3 and repeat the process, modifying the events sent by Actor 2.

41
Step 3 can come before step 2, but it can be argued that while developing a plugin for a specific pipeline actor, one gets accustomed with the properties and needs of the given actor, and it can help making design conclusions, including what data to expect from the previous actor.

5.2 Eiffel and organizational strategy

As expressed by most of the interviewees, the currently used set of tools either do not support traceability or even hinder it, often because to solve one task several solutions are used. This technical problem naturally flows over to an organizational issue. A certain department might need one specific tool from the range of similar or nearly equivalent tools, e.g. a certain issue tracker because it has a functionality that other issue trackers lack. However, from the range of the issue trackers used, it can be possible to identify one that is most essential for the company's pipelines, or one that is planned to gradually take over the others. It can thus be suggested that when implementing Eiffel, these primary tools should be prioritized and become one of the actors to be included in the Eiffel development process described in Chapter 5.1.4 at the beginning of the adoption of Eiffel.

Once such essential pipeline pillars are identified, it is possible to see the connections between them and develop an Eiffel plugin ecosystem, where one Eiffel solution naturally grows upon and improves preceding Eiffel plugins. This would reduce the amount of work and iterations needed to be done, as compared to a situation where different departments experiment with their Eiffel solutions in isolation from the reset of organization. For example, the QA department who experiment with Eiffel, as mentioned in Interview A, might benefit from coordinating their Eiffel solution with the department that develops tools that their testing activities will depend on. An excellent example is the prototype for Artifactory that this research has discussed. The next step could be cooperation with
the QA department to explore their technical needs and make corresponding changes in the events that Artifactory plugin produces.

However, while a technical implementation of the Eiffel protocol that enables the actual communication of traceability information is the most tangible part of a traceability solution, including the described approach of gradual extension of the Eiffel ecosystem, it should be seen as the final step of a much longer process of defining and refining a company’s traceability strategy.

5.2.1 Traceability needs

During the case study interviews several of the uttered desired states concerned use-cases that are only possible to achieve after a significant amount of development efforts towards a fully functional technical implementation of a traceability solution have been made. For instance, end-to-end traceability, pipeline automation, full visualization, triggers-driven pipelines were named by all of the interviewees as part of the desired state and can all be seen as end goals. But at the same time, many of the challenges brought up during discussions about the current state reveal that some fundamental obstacles are not yet tackled nor solved.

Further, as shown in Chapter 4.1.4 and Table 3, many of the concerns raised during the interviews coincided with the Eiffel validation interviews from [8]. The three dominant needs found in [8, p. 978] were also crucial points taken up in the interviews for this paper: test result and fault tracing, upstream tracing, and content downstream tracing.

In our interview results (see Table 4), the need for test result and fault tracing took the shape of the need for fast feedback for bugs, links between tests, features and code (for fault tracing), and others. The often mentioned need for full visualization implies upstream and downstream tracing and presupposes that such needs as creating links between product description and feature tickets or reviews as well as full tracing of all development and testing activities have to be satisfied in order to create a well-functioning automated visualization of pipeline events.

Out of all the needs discovered in [8], only one, salvageability, which “concerns adherence to legal requirements,” [p.980] was not taken up in the interviews for this study. As mentioned in [8], this aspect was of low priority in their study, too: it was mentioned by one manager only.

While [8] point out that traceability is not only a concern from the legal point of view but also a big driving force for troubleshooting, development and integration, our interview results showed that legal aspects were not a big concern at all in this case study. The discussion of traceability needs spun around the need for achieving faster and more effective development and integration processes, and the traceability data and metrics are expected to be a substantial contribution in achieving the company’s final goal of becoming data-driven and moving towards service-oriented business model.

In [8], it was discovered that traceability “is not only considered a crucial challenge in the context of continuous integration, but also an absolute and non-negotiable
prerequisite for making large scale continuous integration work in the first place” [8,
p.982]. Similarly, the endeavour to improve CI and establish CD with the help of
Eiffel was a recurring theme in all of the interviews. Further, the essence of the
traceability challenges largely overlaps with challenges met when introducing
continuous practices in an organization (see Chapter 2.1.4), e.g. need for suitable
tools and cultural challenges. This leads to the conclusion that pipeline traceability
should be seen as an integral part of migration to continuous practices, and not an
add-on on top of an existing CI/CD solution.

One such issue, the need for pipeline actors that would support automatic
generation of traces, was raised by all the interview participants and is listed as one
of the future work paths in [8]. While it is not a traceability need per se, it is one of
the technical obstacles that impede effective traceability and, once solved, can aid in
meeting many of the traceability needs identified in [8] and echoed in this study.

### 5.2.2 Traceability strategy

Implementing the technical parts of a traceability solution at an early stage without
the necessary groundwork in place can lead to a traceability solution that is not cost
effective nor useful to the various stakeholders. Previous research highlights the
importance of having a traceability strategy in place that spans both engineering
and management needs. An important aspect of creating trace links is to carefully
examine the granularity of captured traceability information. The interview subjects
in the case study were often sure of the fact that more information should be
captured, but unsure about the level of granularity or in some cases even what that
information should be.

In this particular case, when a solution such as Eiffel has already been decided
upon, the communication protocol is already specified. One could argue that the
granularity of trace links should correspond to the specification of the Eiffel protocol.
There is however a lot of flexibility when it comes to which fields one must include
when generating a specific Eiffel event. Not all fields are mandatory, but they might
still be crucial to a particular use-case. There is also a consideration to be made
when it comes to which events to generate and one could view the discarding of an
event to be the same thing as removing a level of granularity. In addition to this, the
Eiffel protocol supports custom data fields even though it is somewhat discouraged.

Deciding on the granularity of trace links and which events to include or not is
important because of several reasons. First of all, by involving all stakeholders on a
management level and outlining use cases, an organization can better ensure that
the technical implementation will deliver traceability information at the right level
of granularity. It also brings to light any technical difficulties that could arise from
capturing a significant amount of trace links since they need to be both broadcast
and stored.

During Interview A with a senior engineer that works with automated testing, a
discussion of trace link granularity highlighted that fact that if trace links were
captured every time a test case is started and finished, together with the result of
each test case, an overwhelming amount of trace links would be generated in a very
short amount of time. This is due to the fact that most or all testing is done at the
same time. Such spikes in activity are not possible to handle with the current hardware setup at the company and improving the hardware just to handle such spikes is seen as too costly.

In order to solve this particular challenge this paper suggests two different approaches:

1. **Do not run tests for all products at the same time.** With a service in place for notifying when an artifact is available, the process for choosing when to test an artifact can be changed and testing can be spread out over time.

2. **Do not generate all possible test events.** Only broadcast the result if the test fails. Since all tests broadcasts a trigger event it is possible to deduce that all tests that did not fail must have succeeded, it is not needed to communicate the success result explicitly.

While none of these solutions are perfect and come with their own challenges and caveats, the point is that discovering scenarios like the one above that brings up hidden challenges and obstacles of implementing a traceability solution is only possible by systematically documenting the needs of the end users of the traceability solution. Some challenges can not be solved through the mechanics of capturing trace links but have to be solved on a project management level before a technical implementation is developed.

Previous research has pointed out that there is a focus on the software and processes that facilitate the mechanics of traceability but the needs of the stakeholders and users are not well understood.

The resulting suggestion for a traceability strategy including both the organizational and development steps could take a course similar to the following:

1. **Find use cases:** identify weak spots in the pipeline and decide whether traceability/Eiffel can help it and how, e.g. the case of product specialist who needs release notes information on the right abstraction level.

2. **Explore use case traceability needs:** work with users to understand what information and on what abstraction and granularity level is needed; collect requirements: what information/events do engineers want to react to, and whether it should be an automatic or manual process. E.g., do product specialists want to be notified about developers’ activity regarding a certain feature or do they need information to be available on demand but summarized and presented according to their needs?

3. **Find the chain of tools able to solve the use case:** understand which pipeline actors should be involved in the use case solution and how the communication between them in the form of Eiffel events should look like.

4. **Implement the solution:** extend the identified tools with Eiffel solutions following the implementation steps described in Chapter 5.1.4.
Who is responsible for the activities above, whether the initiative comes from the managements or a certain department, or even engineers themselves, depends on the way a given organization works and how changes are introduced in it.

6 Conclusions and future work

In this thesis we have studied the current and desired state of traceability at a company that has adopted continuous integration. We have explored how the Eiffel framework, in particular by developing plugins that implement the Eiffel protocol for pipeline actors, can be used to make progress towards achieving the desired state. We have made recommendations of best practices for plugin development and in detail described an implementation for a specific pipeline actor.

Through insights gained from interviews and previous research we have come to the conclusion that pipeline traceability is a complex problem to solve with both technical challenges and organizational obstacles.

Based on the findings we conclude that while previous research shows that Eiffel is a viable solution and developing plugins for various system actors is a viable technical strategy, the mechanics of traceability can only be developed after a substantial effort into documenting the use-cases and expectations of the various stakeholders has been made.

It is recommended to further research the best practices of the Eiffel framework and its various tools and architecture implementation guidelines in order to better understand how to integrate the Eiffel framework into an existing continuous integration pipeline. In addition to this, while research shows that a traceability strategy is crucial, it is not well understood how to develop such a strategy in a continuous integration context successfully.
References


## Appendices

### 7.1 Appendix 1 — Interview questions

<table>
<thead>
<tr>
<th>ID</th>
<th>Theme</th>
<th>Type</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQ1</td>
<td>All</td>
<td>Closed</td>
<td>What is your role at Axis?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vad är din roll på Axis?</td>
</tr>
<tr>
<td>IQ2</td>
<td>All</td>
<td>Open</td>
<td>In short, what does your development process involve, in the context of continuous integration?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Kortfattat, vad innefattar din utvecklingsprocess i kontexten av “continuous integration”?</td>
</tr>
<tr>
<td>IQ3</td>
<td>Current State</td>
<td>Open</td>
<td>What solutions, manual or automated, are used today for achieving traceability in your development process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vilka lösningar, manuella eller automatiserade, används idag för att uppnå spårbarhet i din utvecklingsprocess?</td>
</tr>
<tr>
<td>IQ4</td>
<td>All</td>
<td>Open</td>
<td>What are the benefits of traceability in your development process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vad är fördelarna med spårbarhet i din utvecklingsprocess?</td>
</tr>
<tr>
<td>IQ5</td>
<td>Current State</td>
<td>Open</td>
<td>What are the most critical gaps in traceability in your development process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vilka är de mest kritiska luckorna i spårbarhet i din utvecklingsprocess?</td>
</tr>
<tr>
<td>IQ6</td>
<td>Current State</td>
<td>Open</td>
<td>What are the obstacles for achieving traceability in your development process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vilka hinder finns det för att uppnå spårbarhet i din utvecklingsprocess?</td>
</tr>
<tr>
<td>IQ7</td>
<td>Current State</td>
<td>Open</td>
<td>How is Eiffel used today to solve traceability problems in your development process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hur används Eiffel idag för att lösa spårbarhetsproblem i din utvecklingsprocess?</td>
</tr>
<tr>
<td>IQ8</td>
<td>Desired State</td>
<td>Open</td>
<td>How could Eiffel be used in the future in your development process?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hur kan Eiffel användas i framtiden i din utvecklingsprocess?</td>
</tr>
<tr>
<td>Extra</td>
<td>Open</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part of our thesis work is to develop a plugin for Artifactory that implemented a subset of the Eiffel protocol. Based on your skills and experience, what tips and advice can you give us when it comes to implementation?</td>
<td>En del av vår uppsats är att utveckla ett plugin till Artifactory som implementerar ett subset av Eiffel-protokollet. Utifrån din kunskap och erfarenheter, vilka tips och råd kan du ge oss när det kommer till implementationen?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 7.2 Appendix 2 — Research activities

<table>
<thead>
<tr>
<th>Date</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-03-08</td>
<td>Axis supervisor meeting</td>
<td>Talked to Ola and Isac about current progress. Was told to talk to S. about Artifactory and development environment. Agreed to set up interviews after talking to Helena about the questions.</td>
</tr>
<tr>
<td>2019-03-15</td>
<td>Artifactory overview</td>
<td>Talked to S. about how Artifactory works and how to set up a local development environment. Publish events to file while developing/testing. Think about what kind of information should actually be stored in an Eiffel event or if it’s better to just provide links to where to find certain information.</td>
</tr>
<tr>
<td>2019-03-15</td>
<td>MAU supervisor meeting</td>
<td>Supervisor commented on current writing progress in the document. Interview questions should focus on establishing the current and desired state at Axis when it comes to traceability. We should not lean too much on the traceability themes from previous literature. Advice on restructuring on Introduction and LR, as well as directions for methodology.</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>Axis supervisor meeting</td>
<td>Sync-meeting with Ola Söder and Isac Holm.</td>
</tr>
<tr>
<td>2019-03-21</td>
<td>Eiffel pipeline at Axis</td>
<td>Meeting with Isac Holm, Axis supervisor, to establish the current state of the traceability and communication in the pipeline at Axis and discover the future vision of the pipeline with Eiffel. Discussion of the Eiffel events.</td>
</tr>
<tr>
<td>2019-03-27</td>
<td>Interview #1</td>
<td>“Spårbarhet - Våra behov, exjobbsintervju, Eiffel, Artifactory-plugin”. Interviewee A</td>
</tr>
<tr>
<td>2019-03-27/28</td>
<td>Eiffel Hackathon at Axis</td>
<td>Two-day hackathon at Axis with Eiffel as the topic. Developers discussed different aspects surrounding Eiffel and worked on their own solutions/plugins that build upon the Eiffel framework.</td>
</tr>
<tr>
<td>2019-04-01</td>
<td>Interview #2</td>
<td>“Spårbarhet - Våra behov, exjobbsintervju,</td>
</tr>
<tr>
<td>Date</td>
<td>Event</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2019-04-05</td>
<td>Axis supervisor meeting</td>
<td>Sync-meeting with Ola Söder, Isac Holm and the department manager</td>
</tr>
<tr>
<td>2019-04-11</td>
<td>MAU supervisor meeting</td>
<td>Discussion of the current progress on the text</td>
</tr>
<tr>
<td>2019-04-12</td>
<td>Axis supervisor meeting</td>
<td>Sync-meeting with Ola Söder and Isac Holm.</td>
</tr>
<tr>
<td>2019-04-16</td>
<td>Jenkins overview</td>
<td>Mini-workshop with S. about the ways Jenkins stores builds and build information in Artifactory.</td>
</tr>
<tr>
<td>2019-04-17</td>
<td>Interview #3</td>
<td>“Spårbarhet - Våra behov, exjobbsintervju, Eiffel, Artifactory-plugin”. Interviewee C</td>
</tr>
<tr>
<td>2019-05-10</td>
<td>Axis supervisor meeting</td>
<td>Sync-meeting with Ola Söder and Isac Holm. Discussed direct and assisted publishing methods.</td>
</tr>
<tr>
<td>2019-05-15</td>
<td>MAU supervisor meeting</td>
<td>Supervisor commented on structure of discussion part. Suggested a “stairway”-structure where the discussion shows the steps an organization has to take in order to achieve traceability. Both low/technical level and on an organizational level.</td>
</tr>
<tr>
<td>2019-05-20</td>
<td>MAU supervisor meeting</td>
<td>Supervisor commented gave feedback and suggestions on results and discussion chapters and suggested ways to evaluate the thesis with the study participants.</td>
</tr>
<tr>
<td>2019-05-22</td>
<td>Feedback</td>
<td>Feedback for Chapter 4 and 5 with Interviewee A, B1, and C.</td>
</tr>
</tbody>
</table>