Project Report

An Extension of CodeFeedr

Team 1Up

Project Report

An Extension of
CodeFeedr

by

Roald van der Heijden,
Matthijs van Wijngaarden,
Wouter Zonneveld

in order to obtain the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on the 5th of February 2020, 10:30

Project duration:	November 11, 2019 – January 31, 2020	
Thesis committee:	Dr. G. Gousios,	Client, TU Delft
	Dr. A. Katsifodimos,	Supervisor, TU Delft
	Dr. H. Wang,	Bachelor Project Coordinator, TU Delft

TUDelft Delft University of Technology

Contents

Foreword

This document is part of the effort to obtain a Bachelor's degree in Computer Science at Delft University of Technology. It describes the design, process, result, and evaluation of the Bachelor project.

This report is structured as follows: Chapter 1 starts with an introduction to the report. Chapter 2 gives an overview of CodeFeedr, a streaming analytics tool used for mining software repositories, that is to be extended in this project. Chapter 3 follows by describing the preliminary research done by team 1Up, culminating in the MoSCoW requirements for the extensions to CodeFeedr. Chapters 4 and 5, describe the design and implementation of the extensions respectively. Details on testing can be found in chapter 6. The product evaluation on CodeFeedr's extensions can be found in chapter 7 along with an ethical evaluation. The evaluation of the process of the Bachelor Project along with a personal evaluation of the team members are described in chapter 8. Finally, chapter 9 will give recommendations on the future of CodeFeedr.

During the BEP we received help from many people whom we would like to thank. Georgios Gousios introduced us into the exciting field of Mining Software Repositories (MSR) and gave us the opportunity to work on a product meant to be deployed in a production environment. His constructive feedback as client was valuable as well. Wouter Zorgdrager was helpful with his willingness to clarify any misunderstanding we had with the previous version and by devoting time to help us deploy CodeFeedr on TU Delft's Sallab server cluster. Our coach, Asterios Katsifodimos, gave us tips to guide us through this project, for which we are grateful. Lastly we would like to thank Liam Clark, for showing us how to perform a proper retrospective in practice and giving us tips on how to tackle dealing with extending an existing codebase.

Matthijs, Roald & Wouter
Delft
24th January 2020

Summary

CodeFeedr is a Mining Software Repository (MSR) tool designed to efficiently mine massive amounts of streaming data of projects from various sources using Flink's streaming framework in combination with Kafka. Commissioned by researchers at TU Delft on the field of Data Science and Software Engineering, the goal of this project is to expand further on the product, as it already existed in a development stage. At the start of the project, CodeFeedr consisted of a core pipeline functionality and a limited amount of plugins which process data sources.

CodeFeedr-1Up, as this development team calls itself, aimed to achieve two goals: the first goal is increasing the current amount of available plugins, defined by usable software repository sources, to be used by the client; the second goal is to implement a REPL functionality which requests user-friendly SQL-like queries and outputs the queried data stream.

Maven, Cargo, NPM and ClearlyDefined have been developed and have extended the CodeFeedr tool. Furthermore, querying on the aforementioned data sources depending on their data structure is possible for sequential pipelines. With user aid and documentation in mind, logical data models of a plugin's internal structure have been drawn and supplied in the report.

1 Introduction

Data is generated and gathered at a rapidly increasing pace [23]. Real-time data analysis is an ever growing field in computer science which aims to gather insights from these large of data in a timely fashion [10].

A subfield of real-time data analysis is Mining Software Repositories (MSR). MSR is the field of study between Data Mining and Software Engineering and is interested in questions such as:

- Which developers are most important to project Y?
- Which developer sent in more than 5 consecutive commits within one day?
- Which open source maven repository is most affected by security bug 102?

The goal of MSR is to make the software development cycle more efficient and to produce code of higher quality. To achieve this goal, MSR analyses data from projects, code reviews, build reports from Continuous Integration systems, Q&A sites and more [9] [26].

An example of a MSR tool is CodeFeedr [4]. More precisely, CodeFeedr is a software tool for real-time streaming analytics employing a pipeline abstraction in order to be able to deal with multiple sources of information and can combine this with existing state. This report specifically focuses on how to improve CodeFeedr and make it a more useful tool for MSR researchers.

The goal of this report is to describe the current state of CodeFeedr and explore which options are available to extend CodeFeedr. The report furthermore describes the work done to achieve these goals and giving recommendations on possible future work.

2 CodeFeedr

This chapter first gives a high level overview of CodeFeedr. Afterwards, both the current architecture of CodeFeedr and its most important dependencies are described. A more detailed description of CodeFeedr and its dependencies is given by J. Kuijpers et al. [29].

2.1. Overview

CodeFeedr is a framework built on top of Apache Flink [22]. Apache Flink allows for high performance data streaming, but can take up a lot of time to set up. CodeFeedr therefore aims to ease the setup process of streaming jobs. If a user wishes to set up a streaming job, he does so by creating a *pipeline*. A pipeline consists of one or more *stages* which are connected via buffers as a Directed Acyclic Graph (DAG). There are three types of stages: input stages, transformation stages, and output stages. An input stage takes in data from a data source and feeds this through to the next stage in the pipeline. A transformation stage performs a transformation operation on the incoming data. An output stage is the end of a pipeline, and stores or displays the result by e.g. printing the result in a terminal or writing to a file.

Figure 2.1 shows the a high level overview of CodeFeedr's design. However, not all parts are implemented yet. As seen in Figure 2.1, a pipeline can take in several streaming sources as inputs. In CodeFeedr's streaming processor, several Flink jobs combine the inputs in a user specified manner. The output of the streaming processor can be stored in data silos. The data stored, can later be used as input to perform stateful stream processing, i.e. new events will be processed based on the data which was already collected. The query engine allows users to interact with the system and receive feedback from their queries.

A CodeFeedr pipeline can be run in three different modes: mock, local, or clustered. When using mock, only sequential pipelines are possible as no message broker is used, but instead everything is kept in memory. The main purpose of this mode is to test newly added functionality. Local and clustered mode do use message brokers and thus enable more complex DAGs to be run. The difference between these modes is that local runs all stages on a single thread, whereas in clustered mode each stage is started separately in a new Flink environment.

2.2. Architecture

CodeFeedr is built up of two packages namely *core* and *plugins*. The core package contains the building blocks which are necessary to build pipelines such as storing and managing keys, buffers, and serialization. These building blocks are used in the plugins which are used for particular data inputs. An example of an implemented plugin is *GHTorrent* which monitors and obtains data from Github's public event timeline and converts this stream into usable data for pipelines in CodeFeedr [24]. Because of the separation of the core and plugin packages, it is possible for users to create their own plugin. A new plugin only needs to contain the functionality to process the desired data source(s), as it relies on the core package which provides the necessary pipeline implementation.

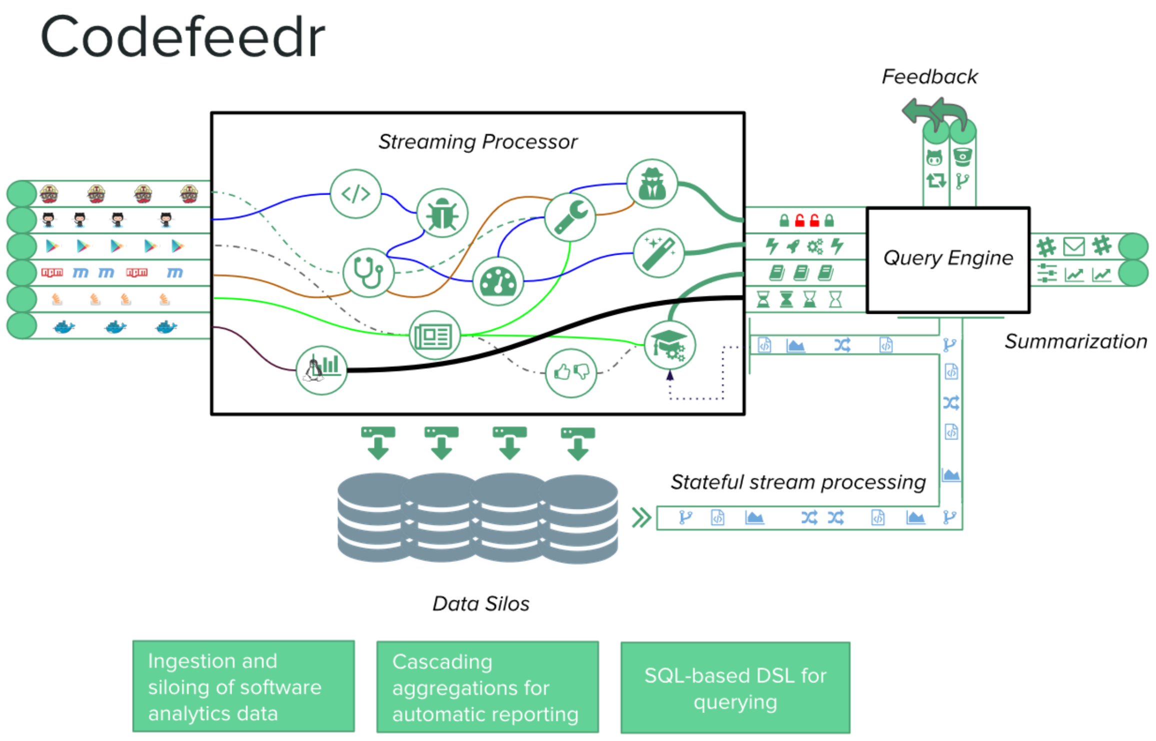

Codefeedr
Feedback
Streaming Processor
Query Engine
Summarization
Stateful stream processing
Data Silos
Ingestion and siloing of software analytics data
Cascading aggregations for automatic reporting
SQL-based DSL for querying

Figure 2.1: A high level overview of CodeFeedr's architecture. Taken from [18].

2.3. Dependencies

CodeFeedr is a framework built on Apache Flink. Flink allows for distributed processing for stateful computations over unbounded and bounded data streams [27]. Since CodeFeedr uses the result of one stream as the input for another stream, a (distributed) message broker is needed. For this both Apache Kafka and RabbitMQ are available. Key management is necessary as public APIs only allow for a limited amount of calls per time interval per key. To maintain a stream of live data, key managers track the remaining usages for multiple keys. Both Redis and Apache Zookeeper are available for key management. For deploying streaming jobs in a distributed setting, containerization is used. Since Docker allows for complex setups and has ready-to-use images for many popular services, Docker is used for containerization.

3 Research Report

The research phase of the Bachelor End Project (BEP) encompasses inspecting the existence and viability of different solutions to the problem statement given by the client, which can be found in appendix A. The final product is based on design choices and goals made in this report.

3.1. Overview

This research report first describes and analyses the problem given by the client. The problem description is listed in section 3.2. In section 3.3 different design goals are outlined which will be adhered to when creating the product to solve this problem. Furthermore, in section 3.4 a requirements analysis is performed to give a more detailed overview of the required features of the product. The development process is given in section 3.5, followed by a comparison with related work in section 3.6. The current implementation of CodeFeedr along with a brief description of its dependencies is given in chapter 2. Finally, several major design decisions which were made at the beginning of this project are motivated in section 3.7.

3.2. Problem Description

This section describes the context of the project and proceeds to give the problem definition. Finally this problem definition is followed by a more detailed analysis of the problem.

3.2.1. Problem Context

CodeFeedr is a project which aims to integrate existing state with (near) real time streaming events, which can be queried by users [40]. The core framework of CodeFeedr has been created in a previous Bachelor End Project and was later expanded upon. However, in its current state CodeFeedr is missing essential implementation details which are required for users to use CodeFeedr with ease:

1. *Difficult setup:* The setup and execution of a data pipeline requires considerable effort from the user, especially if the user has no knowledge of Scala or the surrounding tooling involved with CodeFeedr such as Kafka, Zookeeper, Redis, Flink & Docker.

2. *Missing data sources:* CodeFeedr's codebase has the core functionality to enable data scientists to create their own connectors to other data sources, but it does not provide these data sources in its current state.

3. *Missing output functionality:* CodeFeedr is able to output to a Kafka topic, but is missing functionality to output to for example a CSV file. This has to be done in a separate process, thus increasing the effort which has to be put in by the user.

4. *Ease of interaction:* CodeFeedr requires the user to setup a pipeline using Scala. This requires functional programming experience. It would be much better to be able to use a *declarative language* such as SQL, to be able to specify what results you want in stead of how you want results to be computed.

3.2.2. Problem Definition

The last point mentioned in subsection 3.2.1 is seen by the Software Analytics lab as a large impediment to use CodeFeedr. To ease the interaction between a user and CodeFeedr, the Software Analytics Lab desires an SQL Read-Evaluate-Print-Loop (REPL). A REPL is an interactive shell where a user can type in statements in some language that are *R*ead, *E*valuate, after which the results of evaluation are *P*rinted, after which the systems *L*oops back and the process can repeat until a user selects to quit the program. The REPL should compile a user-provided query and turn this into a corresponding pipeline in CodeFeedr. This pipeline should then be deployed and results should be displayed back to the user.
Furthermore, the client has requested more streaming sources to be added as plug-ins for CodeFeedr. Among these the client is especially interested in the package managers *Maven*, *NPM*, and *Cargo*, licensing information from *Clearly Defined*, and vulnerability information from *Snyk.io*

3.2.3. Problem Analysis

What the team discerned after talking with the client and brainstorming, as underlying parts of the problem, are the following items:

- *Easy to use REPL*
  In its current state setting up the pipelines in CodeFeedr requires the user to program the desired pipelines themselves in Scala. The improvements of this project upon the existing CodeFeedr base should make it possible for someone with the following prior knowledge, to setup and run the pipeline and analyze its results: 1) using a terminal, 2) basic Git commands, 3) SQL statements. The REPL removes the need for knowing extensive functional programming in Scala.

- *More plugins*
  For adding additional plugins a pre-existing template can be used which is provided at CodeFeedr's Github page [41]. One of the desired goals mentioned by the client is being able to query more data sources and being able to output them similar to *Libraries.io Open Data* [8].

- *Pipeline management*
  To obtain the desired functionality of the REPL, a user should be able to combine the results of different pipelines. Therefore it would be useful for users to be able to manage existing pipelines, i.e. get an overview of currently running pipelines and the option to stop them.

3.3. Design Goals

In this section, design goals will be set for this project. The goals are divided into different key properties envisioned for the system.

3.3.1. Maintainability

Since CodeFeedr is under constant development, newly implemented features should be easily extendable and maintainable. New features should furthermore be thoroughly tested on both an individual and integration level. Helpful to this goal is the fact that The code quality will be checked twice during the project by the Software Improvement Group (SIG). To further help extendability, concise documentation greatly improves the ability of future developers to understand CodeFeedr. This documentation should be present in both the source code (comments) and external documents such as a user guide on how to setup the project.

3.3.2. Usability

As the goal of the REPL is to ease the interaction between the user and the program, usability is a key focus for this project. This means that the whole process of setting up a query and executing it should be easy to understand and perform. In CodeFeedr's current state setting up the project can take up a lot of time which could discourage users from using CodeFeedr.

3.3.3. Scalability

Queries run on CodeFeedr should be able to scale up to an environment or down towards a project. The extent to how much data CodeFeedr can handle in large queries will be examined by benchmarks. These benchmarks will be performed at the end of the project, after the REPL has been created.

3.3.4. Security
As a REPL deals with user input, the input should be validated to prevent attacks on CodeFeedr. Input validation, among other things, prevents malicious users to obtain information about the system which they should not be able to obtain. Malicious users could obtain this information when they know about vulnerabilities of the system and when the input is not correctly validated.

3.3.5. Performance
As CodeFeedr is dealing with real-time stream processing, achieving high performance is crucial to keep up with the data being generated. The REPL should not negatively affect the performance of CodeFeedr. No performance tests on CodeFeedr have been recorded so far, so a zero-measure of CodeFeedr's current state should be performed.

3.3.6. Ethics
During the development of a previous project by the client, there was quite the commotion surrounding privacy, which got known as the GHTorrent issue 32 incident [25]. To prevent a negative impact on people who's data is being analyzed by CodeFeedr, Value Sensitive Design (VSD) [17] should be used. An example of a VSD principle is that data should be anonymised.

3.4. Requirement Analysis
In this section the product requirements will be given according to the MoSCoW method. Furthermore, the success requirements are given which will state the minimal criteria which the final product should meet for this project to be deemed a success.

3.4.1. Requirements
After talking to the client and brainstorming among team members our team thought of the following preliminary *MoSCoW* requirements:

- Must haves: The requirements building up to a Minimum Viable Product (MVP).
  - Plugin for Maven package manager
  - Plugin for NPM (Node Package manager)
  - Plugin for Cargo package manager
  - Plugin for Clearly defined (licensing information)
  - Implement the Read functionality for SQL statements in a Read-Eval-Print loop
  - Implement the Evaluation functionality for SQL statements in a Read-Eval-Print loop
  - Implement the Print functionality for SQL statements in a Read-Eval-Print loop
  - Terminal-based REPL
  - The setup of the application has to be user-friendly with minimal technical background
  - The application has to have decent documentation, in order for someone with a minimum amount of technical knowledge to setup and run queries on the available datasets to CodeFeedr
- Should haves: Important requirements of which most should be present in the final product.
  - Plugin for Snyk.io (security information for open source packages)
  - Plugin for deploying a web-based REPL in the form of a Jupyter Notebook
  - Help or documentation: A user must should be able to consult documentation when encountering CodeFeedr's REPL system for the first time
- Could have: Desirable features implemented when time permits it.
  - Stream processing for Bug tracking (e.g. Jira, Bugzilla)
  - Stream processing for Code review tools (Gerrit)

  - The ability to determine differential metrics on a stream of events coming from a package manager stream. This means that the metrics of a live incoming stream, can be compared to corresponding metrics of the current state

- Won't have: Features which are considered but deemed not viable for the scope of this project.
  - Additional plugins (Bower, Composer, NuGet)
  - Plugins for extra version control systems, e.g. Gitlab, Bitbucket.
  - Stream processing CVs mapped onto package level
  - A binary application with extensive UI

3.4.2. Success requirements

For this project to be a success, all the *Must haves* and at least one of the *Should haves* must be implemented in the final product. The *Could haves* are not a necessity for this project to be a success, but are an enhancement which would further improve the product. All the implemented features should furthermore adhere to the design goals described in section 3.3.

3.5. Development Methodology

This section deals with how the team plans to undertake the project. It elaborates on the process the team has picked to bring this project to a success.

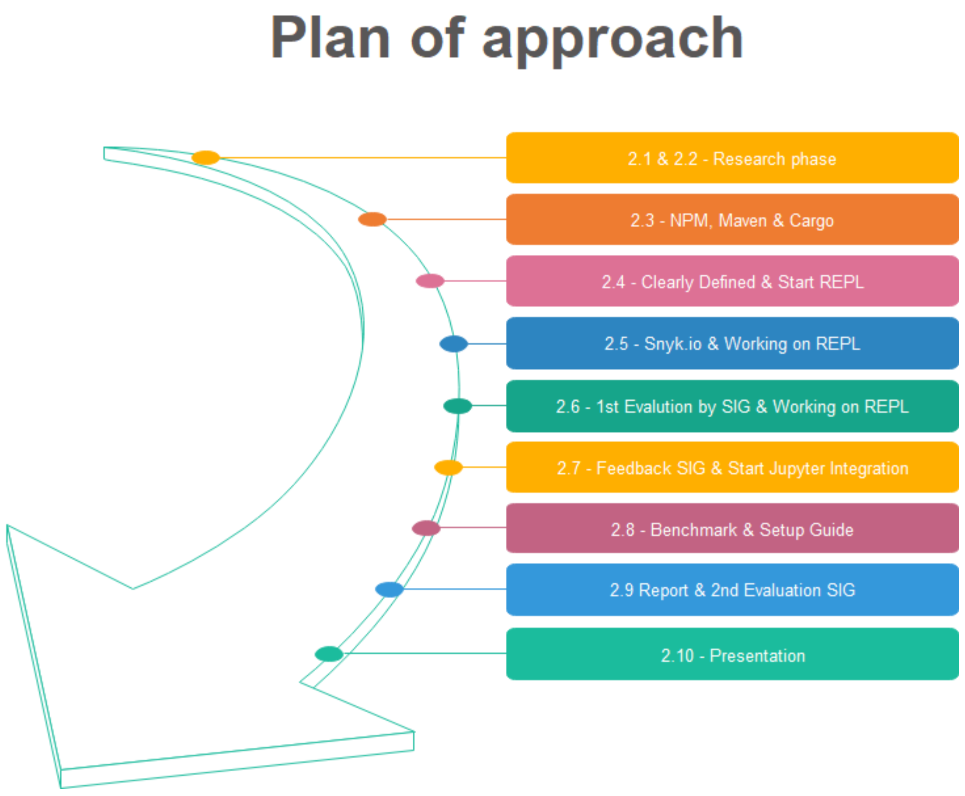

Plan of approach
2.1 & 2.2 - Research phase
2.3 - NPM, Maven & Cargo
2.4 - Clearly Defined & Start REPL
2.5 - Snyk.io & Working on REPL
2.6 - 1st Evalution by SIG & Working on REPL
2.7 - Feedback SIG & Start Jupyter Integration
2.8 - Benchmark & Setup Guide
2.9 Report & 2nd Evaluation SIG
2.10 - Presentation

Figure 3.1: The intended development path.

- *General Process*
  From the start of this project, an agile approach is chosen, as this project will have a small team and an agile approach will ensure that the team can quickly react to changes. Sprints of one week are used as the project only lasts ten to eleven weeks which includes a research and report phase, which both take roughly two weeks.
  Having multiple sprints in the project helps to set small milestones within the project which are good

indicators whether or not the team is still on schedule. Sprint boards are tracked on Asana which contain all tasks which should be completed in a particular sprint. At the end of a sprint, an evaluation will be conducted of how the ending sprint turned out.

- *Communication*
  The communication within the team will mostly be done in person as all members work together daily on the TU faculty. For any pressing matters WhatsApp will be used to contact each other. Since for the majority of the time, all members work together on location, an equal workload distribution is ensured.

- *Version Control*
  For Version Control, Github will be used for pull-based development. During the sprint, features will be developed in separate branches. Once a feature is completed and tested, a Pull Request (PR) will be opened to the *development* branch. A PR must be inspected and approved by at least one other team member before it will be merged into a development branch. The development branch will be merged in the *master* branch once per week which will be the release of a sprint. Both the development and master branch will have Continious Integration which enforces that the project is always in a working state.

3.6. Related Work

Mining Software Repositories is the sub field of Computer Science lying in the intersection between Software Engineering and Data Science. Both fields have a very big interest in combining data to gain insight. CodeFeedr was developed as a tool to be able to achieve this goal. CodeFeedr is however not unique in combining data from different data sources to gain insight. In this section other similar systems are discussed.

3.6.1. CODEMINE

CODEMINE is a platform developed by Microsoft to collect and analyze engineering process data [16]. CODEMINE uses data from source code, process information, build and test results, code reviews, the organization, and work items. This data is retrieved by data loaders and stored to later be queried. Microsoft uses CODEMINE to, among others, understand onboarding processes, optimize individual processes, and to optimize overall code flow. Similar to CodeFeedr, data is mined from repositories, however the key difference between CodeFeedr and CODEMINE is that CodeFeedr incorporates live data whereas CODEMINE requires all the data to first be loaded into the system to later be processed in a batch.

3.6.2. CodeAware

CodeAware is a system for code monitoring and management which is similar to a sensory network defining software versions of monitors and actuators to achieve a fine-grained distributed artifact analysis [1]. Its aimm is to efficiently and proactively prevent faults in stead of what the authors see as fault localization and fixing. The benefits of CodeAware are its ability to monitor at a fine as well as a coarse-grained level. The drawback is that it’s a theoretical system aimed at opening up discussion for future CI ecosystem improvements.

3.6.3. Alooma

Alooma is a real-time data pipeline as a service that can combine multiple data sources such as databases, application data and API [2]. It features a fault-tolerant, high availability ETL system with features such as visualizations of streams of data, managed schema changes and output to Google BigQuery or Amazon Redshift or S3. It supports an API written in Python. In terms of competition, this could be considered the biggest competitor to CodeFeedr. It has recently been taken over by Google, thus it has a great potential userbase. In terms of features and visualizations, it seems richer than CodeFeedr. However, it is aimed at connecting streaming data with data from applications or an API in general and does not have a focus on being a MSR tool.

3.6.4. Amazon Kinesis

Amazon has its own infrastructure for real time stream processing and visualization [3]. Within the AWS environment, Amazon Kinesis is responsible for collecting, processing and analyzing real-time streaming data. AWS infrastructure itself, although advertising scalability and ease of setup is still a commercial option and also is not aimed at MSR specifically.

3.6.5. SQLStreams
SQLStreams is very rich in functionality and has multiple possible compatible streaming sources including Apache Kafka, AWS Kinesis, HTTP, Web and network sockets [37]. They style themselves as "Blazingly Fast, Easy-To-Use and Analyst-Ready" [36]. The main difference with CodeFeedr is also SQLStreams' biggest problem, namely its corporate background and subsequently, the pricing. There are three different types of packages that can be chosen: free, standard and pay (as-you-go). The former is a type of freemium software, where the use of SQLStreams is limited to 1GB of messages a day. The standard packet starts at 51.000 dollars. The last option is price variable, which is not made clear by the company selling the product.

3.6.6. WSO2 Stream Processor
Similar to SQLStreams, WSO2 Stream Processor is a cloud native, lightweight stream processing platform that can digest streaming SQL queries [7]. Similar to CodeFeedr's design goal, WSO2SP emphasizes its simple deployment for users. Furthermore, the software is ran on a Siddhi application and unlike SQLStreams, this product is not connected to a hefty price tag. Its open source nature warrants a free download link on their website. However there is a drawback in this stream processing platform which is the lack of continuous access to improvements, bug fixes, security updates and performance enhancements if the user has only the trial version acquired.

3.7. Design Choices
This section describes the different design choices which were made before the implementation phase of the BEP. For each design choice, advantages and limitations for different implementations are discussed and a conclusions is made.

3.7.1. Platform REPL
From brainstorm sessions in the first week, and meetings with the client and coach, several platforms on which the REPL will run came up as possibilities. The next subsections mention the benefits and the drawbacks of each of the designated options for creating a SQL REPL.

3.7.1.1. Local Terminal
A local terminal means having a SQL REPL on your own computer, either from within an IDE like IntelliJ or Eclipse, or running it standalone as a SQL Command Line Interface. An advantage of a local terminal is that it is the easiest for the development team to implement. This is because no external connectors have to be written.
However, the main drawback of a local terminal is that CodeFeedr would become bound to the computer it is installed on. Using a local terminal also puts the most effort on the user as he has to install the program and all its dependencies.

3.7.1.2. Jupyter Notebook
Having a local terminal is ok, but having the ability of a web-based terminal is even better, since it allows for multiple access from any location in stead of having the REPL run on a local computer. Since MSR is a field lying on the intersection between Software Engineering and Datamining, tools for gaining insight working web-based were investigated. As Jupyter Notebooks are the major tool among Javascript tools for datamining, no other Javascript tools were considered.
Jupyter Notebooks can be created and then shared, to be run on a local instance of a web server or on a central web server. These created documents are able to combine live code, visualizations, and text. This enables teaching or interactive documentation, a form called literate programming [28]. Using Python these notebooks are frequently used by data scientists to visualize and share data. Jupyter Notebooks are powerful, if they can be connected to CodeFeedr by transforming data from Kafka into a CSV or NumPy array, all Python tools becomes available to perform data science on the query results. Or an alternative is to use Python-lenses to query real-time streaming sources like Kafka and gaining the ability to query using SQL in Jupyter. Another advantage of Jupyter Notebooks is that they can be written and hosted on different machines, making it more portable than a local SQL REPL. Furthermore, Jupyter Notebooks are open-source. This is a big advantage in perspective of security, having more eyes prying into source code and the ability of a fast expanding userbase by allowing user contributions. However, there is no experience among the team members in setting up Jupyter Notebooks. Experience in Python is also limited among the team which could increase the time it takes to successfully integrate these notebooks with CodeFeedr.

3.7.1.3. Choice of platform

The final choice for platform development will be that we will implement both options. First, the team will start with a local terminal, since that is easiest to implement. Then, the team sets out to create a Jupyter Notebook-based REPL in order to have all the power of Jupyter available.

3.7.2. Language

The design choice on which language will be used for this project will be split up into two parts: plugins and REPL.

3.7.2.1. Plugins

For each plugin, a new project can be created by using an existing template provided on CodeFeedr's Github [41]. This project will use Scala. As this template is specifically created for CodeFeedr plugins, all plugins will be written in Scala.

3.7.2.2. Language of REPL

As CodeFeedr is programmed in Scala, two programming languages come up as candidates for implementing new features, namely Scala and Java. Scala is a candidate as this is consistent with the current implementation of CodeFeedr. Java is an option as Scala can be compiled into a jar file which can be used in a Java project. In order to stay consistent, have less boiler plate code and keep maintainability up, the language of choice will be Scala initially. Depending on how the project evolves, a choice might be made to implement more than just the local SQL REPL. The advantage of having a Jupyter Notebook hosted with some way of running SQL queries in it and having the results returned as objects which Python can handle are many. The complete arsenal of NumPy and SciPy could be at the user's disposal making it a great option for data scientists. Having a webpage with an SQL REPL in for example Javascript is also a good option. The language for the webpage option isn't set in stone yet, so during the project the team will look at possible alternatives when they get to this point.

3.7.2.3. Choice of languages

The final choice for REPL was a local terminal and a web-based one in Jupyter Notebook. Considering the existing codebase for the local terminal we will use Scala and for the web-based REPL we will employ Python in Jupyter Notebook to develop the web-based REPL.

3.7.3. Streaming SQL

The REPL will use SQL to process user queries, however since the input data comes as a stream, traditional SQL will not be sufficient. Traditional SQL assumes the input is bounded whereas streams are typically unbounded. Streaming SQL is created to query unbounded data and output results as new data comes in. This section describes different Streaming SQL frameworks of which one will be picked to be used in this project.

3.7.3.1. SamzaSQL

While SamzaSQL does provide the functionality to process streams at a massive scale, it is important to note right off the bat that this framework was built upon the Apache Samza framework [31]. This framework is different from the one the original Codefeedr was built upon, namely the Apache Flink framework. While SamzaSQL does provide interesting and useful functionalities such as defining stream processing logic declaratively as an SQL query, among other things, it will not be too useful to dive deeper into this topic due to its inapplicability with CodeFeedr.

3.7.3.2. Calcite

Since Calcite is already being used in CodeFeedr and has Streaming SQL capabilities, Calcite seems like a good choice for this project. While calcite does show some promising functionality in their documentation [11], not all of it is implemented yet. Features such as tumbling and hopping windows to deal with computed aggregate functions do exist, but much more vital and important features such as the JOIN operation for stream-to-table and stream-to-stream functionality is lacking documentation.

3.7.3.3. Kafka KSQL

Similar to SamzaSQL, Kafka SQL, or KSQL, enables real-time data processing without the need to write code in a programming language like Java [14]. This language is built upon the Apache Kafka framework. One important thing to note of KSQL is the available JOIN functionality which Calcite currently lacks [15]. To be exact, as of starting this project KSQL supports INNER, LEFT OUTER and FULL OUTER joins for Stream-Stream. For Stream-Table, INNER and LEFT joins are available. While this means not every tool would be available in the development process of CodeFeedr, this is a considerable step up compared to Calcite's built-in join operations. Some other functionalities in KSQL include transforming, aggregating, windowing and sessionization.

3.7.3.4. Conclusion

In conclusion, since the original version of CodeFeedr was already heavily dependent on functionality of Flink's SQL language, this extension has built further upon that framework. The creation of tables based off of the structure of a object inside a DataStream object and the subsequent querying on such tables are all Flink SQL functionalities. Therefore we chose to continue using this library, despite initial concerns about missing and inconsistent documentation.

4 Software Architecture

This chapter describes the architecture of the software and different design patterns that have been implemented, supported by explanations of those design pattern choices. Furthermore, it explains the different design decisions that were made during the project with regard to both the plugins and the SQL REPL sections of Codefeedr.

4.1. Design Patterns

In order to deliver a sound product with high maintainability, certain design patterns have been taken into consideration during the development process.

4.1.1. Strategy Pattern

Due to the overlapping nature of the plugins, the *Strategy* design pattern was utilized [34]. This choice was made due to the plugins being very cohesive, but simultaneously completely independent of each other. To realize the Strategy design pattern the choice was made to perform an abstract coupling of some of the shared plugin functionality with shared parent classes. The same goes for each plugin's configuration class, as well as the plugin-specific *SQLStage* functionality. With the amount of shared functionality in each individual plugin's source files, this design choice reduced a lot of code duplication. By using proper inheritance and low coupling, the use-case of fixing a bug in the future for one plugin and forgetting about or overlooking it in other plugins is simpler, due to much of its implementation now being located in a shared, abstract parent class. As a result, maintainability of Codefeedr's codebase is improved.

4.1.2. Singleton

Furthermore, multiple *Singletons* are used per plugin [33]. Some parts of the code, such as parsers for JSON and SQL table structure creators, are all collectively used as utility classes where only a single instance is necessary. On use, it loads (lazy) variables to be kept track of as a global instance and therefore ensures no other instances initialized. For example, each plugin has the *Protocol* object, which acts like a singleton and defines the internal data structure of a plugin's object in a data stream.

4.1.3. Adapter

Adapter design patterns are also implemented to deal with the problem of sharing functionality per plugin but minimizing the coupling [32]. Specifically, when dealing with the question on how to best implement the registering of SQL table information for multiple different plugins with different data structures. This problem was tackled by creating an adapter class and function, which all plugins could connect to, and depending on the plugin's type input, supplied the proper functionality.

4.2. Plugins

The design of the plugins inherited much of the structure of CodeFeedr. However, much of its functionality had to be altered and modified in order for the plugin to function properly with accordance to Codefeedr's overarching architecture, so some critical design decisions still had to be made. For example, some plugins have two input stages, while others have one. The decisions and the reasoning behind them will be discussed per plugin. Furthermore, each plugin's data structure has been carefully examined and through extensive testing been put in numerous tabled structures. This process and its results will be discussed extensively in chapter 5.

4.2.1. Maven

The Maven plugin contains the functionality of reading a RSS feed of updated/new Maven artifacts, and with a second web request retrieves further project information. However, it became clear that finding an update stream of all new and updated artifacts was challenging. Eventually an update stream was found but this stream does not contain all updated packages, merely a chunk of them. For this reason the Maven plugin was split in two stages. The first stage parses the RSS feed, thus retrieving updated/new artifacts. The second stage takes the first stage as input and retrieves the project information of these artifacts. By designing this plugin in this way, whenever a better update stream is created, the workload required to connect the different polling source to the existing second stage will be minimized.

4.2.2. Node Package Manager

The Node Package Manager (NPM) plugin enables reading an update stream of all updated/new NPM projects and retrieving their project information. Although the update stream of NPM is complete, its uptime is lacking. At seemingly random intervals the website can't be accessed for a couple of minutes. While this is certainly a disadvantage to the sourcing of a data stream, it only partially hinders the functionality of this plugin, since intervals of downtime are often followed shortly after with uptime. If the list of the update stream still has the updated package, then all lost packages are then retrieved and processed with a single poll, as long as the polling interval has a reasonable size. While certainly a downside taken under close consideration by the team, no better alternatives were found for now. It is possible that a more reliable update stream is found in the future. For the same reason as with Maven, namely the possibility of a different update stream being used and implemented in the foreseeable future, the section of retrieving the update stream was separated from obtaining specific package information.

4.2.3. Cargo

Cargo is the package manager and crate registry service for the Rust community [12]. Due to its open-source nature, the update stream of new and updated packages are freely and easily available online. Codefeedr utilizes this update stream by instantly obtaining key information pieces on a particular crate (package). With a check on the latest item polled and processed, the crates which have not been processed yet are then extensively polled by a second, extended web request, thereby obtaining the full package information. By design, Crate has a minimized version, which is used in the first stage of polling the update stream, and an extended version, which is only obtained if not previously processed. By design, this will drastically reduce network payload. Unlike the previous two package managers, Cargo's plugin architecture is not split in two different stages (basic and extended), because the plugin's update stream is reliable and complete. There is a split between a 'CrateFromPoll' and a 'CrateRelease', but this is processed within the same underlying pipeline stage.

4.2.4. Clearly Defined

Clearly Defined is a project by its parent organization, the 'Open Source Initiative', to help make FOSS (Free and Open Source Software) projects more attractive [13]. This is done by publishing information on the software such as licensing and source location and therefore making it, like the name suggests, more clearly defined. The ClearlyDefined plugin gathers information about ClearlyDefined-registered projects. Unlike the previously mentioned plugins, this data source has a single stream containing both the names of all the new and updated projects, as well as their project information. When looking at the proposed architectural design for this plugin, one important factor with this plugin compared to the other plugins is that there is no need for a second web request for an individual package. All the package information is stored in the same polling stream as the fetcher. While this is a dubious design choice by the ClearlyDefined organization, since the result of a single poll is a rather large and convoluted JSON string, this is what Codefeedr will have to deal with. Therefore it follows logically that a single poll will instantly retrieve, process and output the JSON into a fully filled ClearlyDefined object. As with Cargo, ClearlyDefined will therefore have one input stage only.

4.3. SQL REPL

The SQL REPL uses a newly implemented *SQLStage* which extends CodeFeedr's OutputStage. Like CodeFeedr's output stages, an SQLStage can take in one to four inputs. The SQLStage is a generic class, meaning it accepts a data stream of any class. The SQLStage first registers the incoming data streams into tables. By registering data streams into tables, the streams can now be queried much like regular databases. The query can be passed to the SQLStage as an argument when running the application. The output of a query is returned as a data stream which can be outputted to a file, printed to a console, or by using a custom SinkFunction.

4.3.1. Registering tables

Although the SQLStage is generic, registering a table needs a specific implementation for each incoming class. Classes from plugins are often complex and contain list types. For the newly added plugins (Clearly Defined, Cargo, Maven, and NPM), each complex field and list types are registered as a new table with a foreign key relation to its parent. The result is a database in second normal form [19]. Because the database is in first normal form, queries are easier to write as there are no repeating groups. Furthermore, redundancy is eliminated which improves the performance.

When a stream is used as input but no implementation of registering its class to a table is present, an exception will be thrown.

5 Implementation

This chapter describes the implementation details of the additions to CodeFeedr. Section 5.1 first outlines the details of the plugins, followed by section 5.2 which provides a description of the SQL REPL.

5.1. Plugins

Certain facets of the implementation process of the plugins overlapped, such as the polling of an update stream, the necessity to create plugin-specific data structures and the filtering of packages after polling.

5.1.1. Polling

When implementing the plugins, it was important to keep in mind that the process of polling packages from a package source, as well as the parsing of a package had to be built efficiently. Depending on the plugin, the polling interval decided how much time passed between two polls, meaning that if a polling interval was short due to a very active software repository, the efficiency of the aforementioned process was crucial. Therefore, during implementation of the various plugins for Codefeedr, extra focus and effort was put in the development of a most efficient retrieval of package data as possible.

5.1.2. Data Structure

Since every plugin represents a different type of package, the content of each package varies. A protocol class, containing fields with the field name and its corresponding data types is created for every plugin. This class defines the internal structure of a single package using case classes. However not every project specifies the same information even within the same plugin. Where certain fields might be specified in some projects, they could be omitted in others. Due to Scala's Option[] type, these so called nullable fields could be implemented by wrapping an *Option* around the field. However, during implementation the information on which fields to define as an Option was not available. When running a plugin which processed a package with a field which otherwise was thought to be non-nullable to suddenly have a nulled value crashed the plugin. It took a number of weeks of trial-and-error and extensive debugging to be confident with the current state of each plugin's protocol data structures.

```
case class MavenProject(
                      modelVersion: String,
                      groupId: String,
                      artifactId: String,
                      version: String,
                      parent: Option[Parent],
                      dependencies: Option[List[Dependency]],
                      licenses: Option[List[License]],
                      repositories: Option[List[Repository]],
                      organization: Option[Organization],
                      packaging: Option[String],
                      issueManagement: Option[IssueManagement],
                         scm: Option[SCM])
```

An example of a case class from Maven\protocol\Protocol.scala

5.1.3. Filtering

During the polling of an update stream of any software repository, it is highly likely that multiple of the packages, if not all packages have already been processed by Codefeedr. To prevent duplication of packages and simultaneously improve the performance of the system, only packages which had not previously been processed will go through the extended processing and added to the output stream. This is done in all plugins by keeping track of the latest package processed. When the update stream is polled and a number of packages are pending processing, the update field with a Date type is compared to the latest processed package. If the Date is earlier than the saved package, it is skipped, otherwise it will be processed and it replaces the 'checkpointed' package.

```
def sortAndDropDuplicates(items: Seq[CrateFromPoll]): Seq[CrateFromPoll] = {
  items
    .filter((x: CrateFromPoll) => {
      if (lastItem.isDefined)
        lastItem.get.crate.updated_at.before(x.updated_at)
      else
        true
    })
    .sortWith((x: CrateFromPoll, y: CrateFromPoll) => x.updated_at.before(y.updated_at))
}
```

An example of filtering packages from CargoReleasesSource.scala

5.1.4. Cargo

The Cargo plugin utilizes the update feed from *Crates.io*'s own website. The feed, available at *https://crates.io/api/v1/summary*, shows a continuously updated string in the retrieved HTML body. Among other interesting information, the polled string contains basic information on ten of the newest crates (package in Rust's package manager) as well as ten of the most recently updated crates. In the plugin's implementation, these parts of the string are parsed and filtered into *CrateFromPoll* objects, which saves only the crate's name and date when it was added or updated. A crate's name functions as the primary key in Cargo, and the date field can then be used to filter packages out of a poll as described earlier.

When a crate is processed further for the data stream, another web request is sent out to *https://crates.io/api/v1/crates/X*, where X is the name of the crate. The response is another string in JSON format describing the full crate with complete information. This information is then parsed into Cargo's data structure using a static utility class called *JsonParser.scala*. Unlike the other plugins, which use libraries to instantly deserialize the JSON string into a case class, Cargo's deserializer is hard-coded due to development complications.

During implementation it was made clear that a particular field in Cargo's data structure was both incomprehensible and inconsistent in regard to field names and field values, namely the Badge object. After some consideration it was chosen not to include this field to the plugin.

The result of the Cargo plugin is the implementation of a new stage on the pipeline with an output stream of complex Cargo objects. A snippet of an example Cargo object can be seen in Figure 5.1. The complete logical data model of Cargo can be found in appendix C

5.1.5. Clearly Defined

The ClearlyDefined plugin implements a stage for Codefeedr that outputs a *DataStream[ClearlyDefinedRelease]*. ClearlyDefined's polling location is singular, with all information on recent packages centralized at the same source, namely *https://api.clearlydefined.io/definitions?matchCasing=false&sort=releaseDate&sortDesc=true*. The response of this webrequest contains full information on a hundred ClearlyDefined projects in JSON format. As a result, the serialized string is very large compared to the other plugin sources.

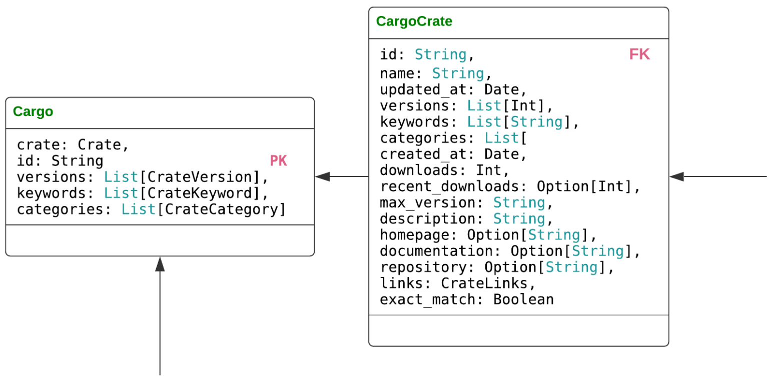


Figure 5.1: A logical data model of the Cargo parent class and one of its children

With each single poll, a *packageAmount* number of packages are polled out of the hundred. This variable is currently set to 10, and is configurable in the code source only. It is not recommended to increase this value due to the slow nature of ClearlyDefined releases. For each package polled, the *Release->meta->updated* field was used to determine filtering and sorting. Deserializing the processed packages to the desired case class was handled by the Jackson JSON library in Scala.

The result of the ClearlyDefined plugin is the implementation of a new stage on the pipeline with an output stream of complex ClearlyDefined objects. A snippet of an example ClearlyDefined object can be seen in Figure 5.2. The complete logical data model of ClearlyDefined can be found in appendix C.

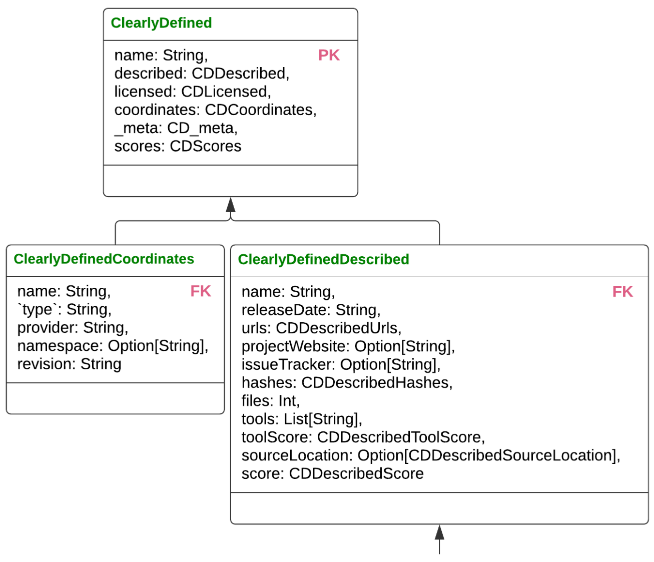


Figure 5.2: A logical data model of the ClearlyDefined parent class and two of its children.

5.1.6. NPM

The NPM plugin implements a stage to the pipeline of Codefeedr which outputs NPM objects. Much like Cargo, the process of retrieving the update stream and requesting complete package information is bipartite. The URL used to obtain the names of all the most recently updated and created packages is *https://npm-update-stream.libraries.io/*. The list of NPM package names is then intersected with an existing list of previously processed packages. The remaining packages are then polled at *http://registry.npmjs.com/X*, where X is the qualified package name.

It is important to note that the update stream is highly unstable. It often happens that the website is unavailable for some consecutive moments. This is why an automatic re-request is sent if the first fails. Another important divergent aspect is that this source is third-party, meaning we aren't fully independent and have to rely on the completeness of what is offered.

For that reason, the NPM plugin splits the staging of data streams in two. The first stage is the process of retrieving the information from the source and deserializing the information into case classes, much like the previously explained plugins. However, the deserialization of the data has only been done on limited information, into an *NpmRelease* object. The next stage then takes the aforementioned object as input, and outputs an *NpmReleaseExt* object, with much more complete information. By splitting the stages in two, core functionality can be left unaltered when future changes might apply. There is a realistic chance that a better update stream is discovered in the future which would then only require a programmer to edit the first stage of the plugin.

During implementation it was noticed that many NPM packages are set as *unpublished*, meaning they lack critical information. For this reason they are of little use and are unable to be properly processed. Many packages will be filtered out due to this complication.

The result of the NPM plugin is the implementation of two new stages on the pipeline with an output stream of complex NpmReleaseExt objects. A snippet of an example NPM object can be seen in Figure 5.3. The complete logical data model of NPM can be found in appendix C.

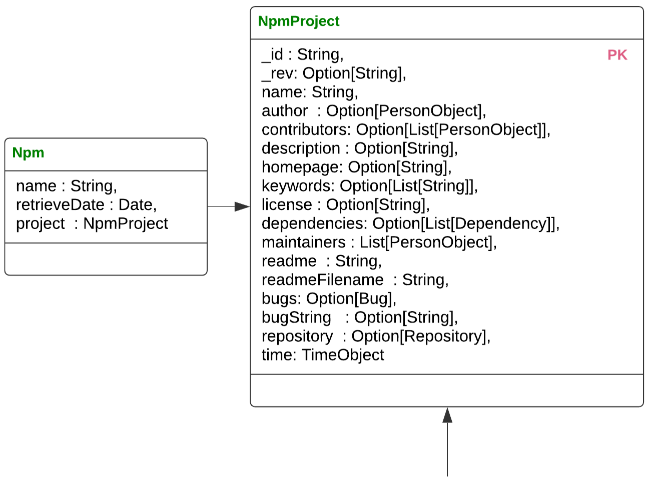

Npm
name : String,
retrieveDate : Date,
project : NpmProject
NpmProject
_id : String,
PK
_rev: Option[String],
name: String,
author : Option[PersonObject],
contributors: Option[List[PersonObject]],
description : Option[String],
homepage: Option[String],
keywords: Option[List[String]],
license : Option[String],
dependencies: Option[List[Dependency]],
maintainers : List[PersonObject],
readme : String,
readmeFilename : String,
bugs: Option[Bug],
bugString : Option[String],
repository : Option[Repository],
time: TimeObject

Figure 5.3: A logical data model of the NpmProject parent class and one of its children.

5.1.7. Maven

The input stage of the Maven plugin reads information from an RSS feed found at *https://mvnrepository.com/feeds/rss2.0.xml*. This feed contains an XML structure with information about the title, link, description, and publication date of newly added/updated Maven artifacts. As discussed in subsection 4.2.1, this feed is not complete.

From the retrieved project names, the project information is retrieved at *https://repo1.maven.org/maven2/*. From this site, the .pom file is retrieved which again contains an XML structure. For the parsing of XML no external library is used. Although libraries which extract Scala case classes from XML exist (such as Xtract [30]), making these libraries work requires a lot of boilerplate code, resulting in no advantages using them. Therefore the team decided to manually extract the necessary information from the XML. Which values are extracted from the .pom file can be found in Listing 5.1.2.

The result of the Maven plugin is the implementation of two new stages with an output stream of complex MavenReleaseExt objects which contains all relevant information from Maven artifacts. A snippet of the structure of a Maven object can be found in Figure 5.4. The full structure can be found in appendix C.

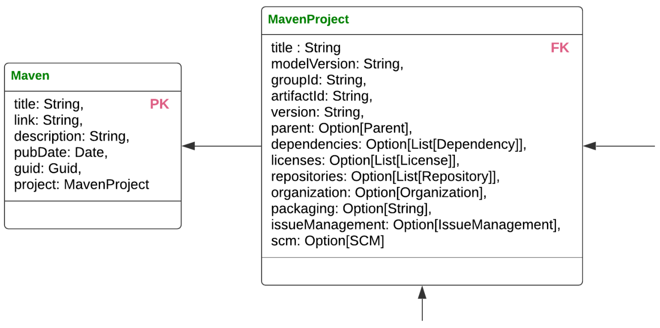


Figure 5.4: A logical data model of the Maven parent class and one of its children.

5.2. SQL REPL

The implementation details of the SQL REPL is split in two parts. First the Streaming SQL aspect will be discussed. Hereafter the required data normalization is examined.

5.2.1. Streaming SQL

The implementation of the SQLStage enables the use of Streaming SQL (SSQL) in CodeFeedr and works as followed: First a Flink StreamTableEnvironment is created [7]. Secondly, each structure of the incoming data stream is registered into tables in the created StreamTableEnvironment. Finally the query is executed again using the StreamTableEnvironment. A downside to the Flink dependency is that it the dependency still under development which means that not all intended features are already implemented. For example, as of now, case classes from Scala are not supported to perform queries on. Since in CodeFeedr case classes are used, this complicated the process of performing queries. For the Cargo, Clearly Defined, Maven and NPM plugins, a conversion from their Scala case classes to a Plain Old Java Object (POJO) is implemented since POJOs are supported by Flink. Although this was a necessity to make SSQL work, it doubles the size of the Protocol object for each plugin.

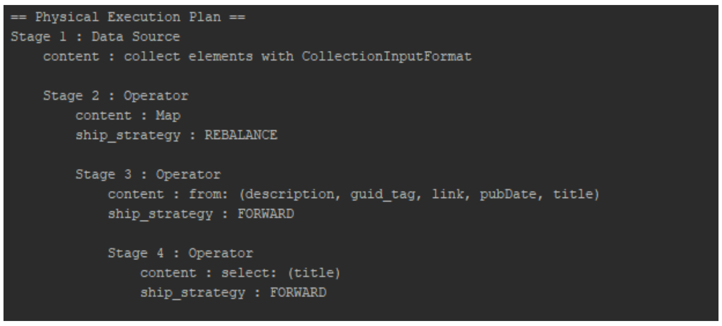
Another issue, which was only encountered late in the implementation phase of the project, is that Flink's SSQL dependencies don't give output when using Kafka topics. This issue was not foreseen as both Flink and Kafka are products of Apache, and as different running modes of CodeFeedr don't alter the call graph of the SQLStage. The root of the issue appears to lay in a changed physical execution plan of the query. As seen in Figure 5.5, two different execution plans are generated for the same query: *SELECT title FROM Maven*. When using Kafka an additional data source is added along with a data sink. Why this is done is unknown to the team even after many hours of debugging.

5.2.2. Data Normalization

In order to properly query on incoming data from different streams using the query tables, some changes to the original data structure have been made. The biggest challenge was that all data structures contained lists of objects. This would be impossible to query on. The solution to this problem was to extract the objects in a list in a separate streaming table. This was done with a simple flatmap operation on the list value.

However, this created the problem that a user would be unable to link the values of the list to the parent object the list once belonged to. Therefore the identifying field of the object was taken and manually inserted in the newly created table. By applying this technique to all list fields, we transformed our data-stream-table structure into 1NF, since at processing time it is already ensured that each row is unique using the previously described filtering technique.

Furthermore, due to each plugin's data structure, parent objects can have multiple child objects. However, since the data was extracted from JSON where the tree structure was inherent, there were no identifying keys in child objects which would allow a user to connect a child with a parent. For that reason id fields have been inserted in children, manually adding foreign key constraints to the plugin's main identifier, which was often the name of the package. As can be seen in the logical data models in appendix C, users will now be able to properly query on package fields originating from various layers within the same object and perform joins if they wish. Additionally, as a result of all four plugins having singular primary key columns, the stream-table



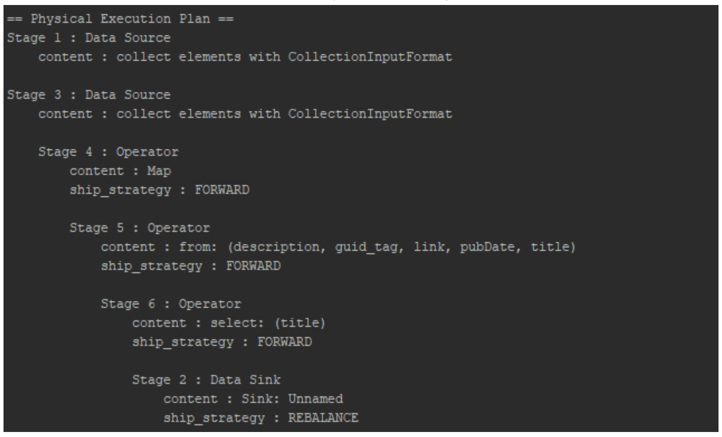
```
== Physical Execution Plan ==
Stage 1 : Data Source
    content : collect elements with CollectionInputFormat

    Stage 2 : Operator
        content : Map
        ship_strategy : REBALANCE

        Stage 3 : Operator
            content : from: (description, guid_tag, link, pubDate, title)
            ship_strategy : FORWARD

            Stage 4 : Operator
                content : select: (title)
                ship_strategy : FORWARD
```

(a) Result query without using Kafka.



```
== Physical Execution Plan ==
Stage 1 : Data Source
    content : collect elements with CollectionInputFormat

Stage 3 : Data Source
    content : collect elements with CollectionInputFormat

    Stage 4 : Operator
        content : Map
        ship_strategy : FORWARD

        Stage 5 : Operator
            content : from: (description, guid_tag, link, pubDate, title)
            ship_strategy : FORWARD

            Stage 6 : Operator
                content : select: (title)
                ship_strategy : FORWARD

                Stage 2 : Data Sink
                    content : Sink: Unnamed
                    ship_strategy : REBALANCE
```

(b) Result query using Kafka.

Figure 5.5: Difference in physical execution plan using the same query: SELECT title FROM Maven

structure transformed into 2NF. 3NF was not achieved in this project, as it would prove very difficult for us to determine which fields shared a transitive, functional dependency, and it was not a top priority in this project.

6 Software Testing

This chapter describes all relevant information related to testing within this Bachelor project. Section 6.1 gives a general overview of how testing was tackled during development, followed in section 6.2 by a description of the different kind of tests that were employed. Section 6.3 describes the test results and gives recommendations for improvements.

6.1. Testing approach

"Testing shows the presence, not the absence of bugs." is a famous quote by Edsger Dijkstra [38]. Indeed writing tests does not guarantee the absence of bugs, but it does give a developer a certain level of confidence in his written code. During this project, tests were written from the start to improve confidence in newly added code. Although no Test Driven Development approach was taken, tests were a prominent review part when merging Pull Requests (PRs). For all new code tests should be present. Occasionally the project suffered from some technical debt in terms of testing [35]. However a great effort was made to pay this debt in the next sprint. Continuous Integration was provided by setting up a Github Workflow for Scala/SBT. Github.com automatically ran tests when pushing local commits to its respective branch on the central repository or when performing a PR to ensure a working build on the development/master branch.

6.2. Test types

Software testing is done on multiple levels. Some tests are aimed to stress test a system whilst others aim to uncover bugs in edge cases. This section describes what type of tests are used in this project.

6.2.1. Unit testing

The vast majority of tests in the test suite of this product consist of Unit tests. Unit tests execute small pieces of code (e.g. a single method) and confirm that the code works according to expectation. Unit tests should furthermore be modular, i.e. the execution of test A, should not affect test B. In this project unit tests were among others used to test for situations where external sources would not respond as foreseen. For example, a website could be temporarily unreachable.

6.2.2. Integration testing

The test suite furthermore contains integration tests. Integration tests don't test on a modular level, but integrate several pieces of code and confirm that the pieces are working conjointly. In practise this could mean that the interaction between two or more classes is tested.

6.2.3. System testing

Besides automatically run test suites, system testing has been performed by making an effort to imitate a real life usage scenario. An instance of Kafka and Zookeeper was installed on Ubuntu 19.04 running on an Intel® Core™ quadcore i5-3470 CPU @ 3.20GHz with 16 GB of internal memory and 320 GB of hard drive space available. This setup was used to perform a dry run of deployment for an extended period of time. Although this setup pales in comparison to the cluster setup of the client, having this setup was still useful.

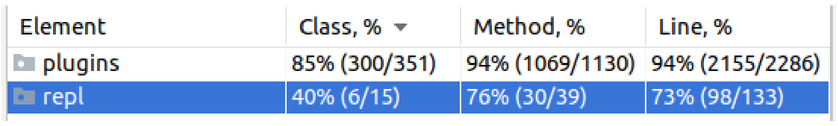
The plugins were deployed on this private server before actual deployment on the cluster of the client, inspecting logging information and checking any unusual situation. Since this server could run for a far longer time than any of the developers laptops, this helped potentially clear the most high-profile bugs before actual deployment.
After this first system test, the plugins were deployed on the TU Delft Sallab cluster, on which the software has been running for almost two weeks as of writing this report.

6.3. Test results

This section presents the results of testing. First coverage result are listed, followed by a discussion of potential pitfalls of missing coverage, ending with a description of the results of actual deployment testing. The code is covered by an overall of 266 unit and integration tests and about a week of deployment testing.

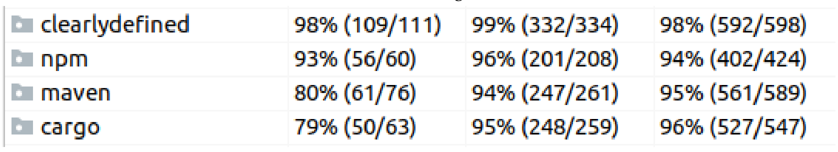
6.3.1. Coverage results

Testing was aimed to be as automated as possible, in order to prevent having to test manually, which is slow, expensive and prone to errors.As test coverage measure, line coverage was selected as it is a simple, yet effective and widely applied measurement. A minimal line coverage of 85% was set at the beginning of the project by the team. 85% was chosen as this was often told to be a good aim during the Software Quality and Testing course. As seen in Figure 6.1, this goal was reached by a rather large margin.



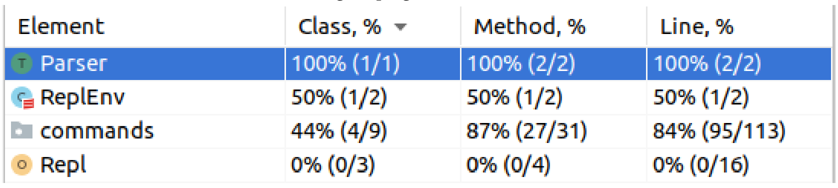
Element	Class, %	Method, %	Line, %
plugins	85% (300/351)	94% (1069/1130)	94% (2155/2286)
repl	40% (6/15)	76% (30/39)	73% (98/133)

(a) Overall coverage: 93%.



clearlydefined	98% (109/111)	99% (332/334)	98% (592/598)
npm	93% (56/60)	96% (201/208)	94% (402/424)
maven	80% (61/76)	94% (247/261)	95% (561/589)
cargo	79% (50/63)	95% (248/259)	96% (527/547)

(b) Coverage of plugins: Minimum of 94%.



Element	Class, %	Method, %	Line, %
Parser	100% (1/1)	100% (2/2)	100% (2/2)
ReplEnv	50% (1/2)	50% (1/2)	50% (1/2)
commands	44% (4/9)	87% (27/31)	84% (95/113)
Repl	0% (0/3)	0% (0/4)	0% (0/16)

(c) Coverage of REPL: 73 %.

Figure 6.1: all attained coverage results.

Although exhaustive testing is usually impossible, aiming for high coverage is nonetheless a valuable goal. There are mainly three reasons for our software product to have lower coverage in certain areas:

1. *Functions or code which can't be tested*
   Some of the code we developed are extensions of base variants in the CodeFeedr core. It turned out to be impossible to mock these situations so some lines in the `stages` and `operators` packages cannot be tested. For specific classes coverage dropped to about 80% but overall on the plugins the results are considered adequate.

2. *Full coverage but not full line coverage*
   To achieve full coverage on a Scala case class, many tedious tests need to be written. The methods *toString*, `hashCode`, and `unapply` need to be tested for all case classes to reach full coverage. Since this takes up a lot of time and clutters the test code with lots of irrelevant code, the team decided this was not worth the effort.

3. *Work in progress*
   Part of the REPL was developed by the team slightly differently than the client had envisioned. This resulted in functionality which was not necessary for the product but could still prove to be useful in

the future. Because this code was not vital for the product however, no time was invested in testing this functionality. This led to a slightly lower degree of coverage of the REPL package.

6.3.2. System testing results on local server

After the team's server was setup and configured to run Kafka and Zookeeper locally four instances of IntelliJ were fired up. At that moment the plugins still had different repositories so it was easy to run them separately and in parallel.

- *Cargo* was initially the most successfully deployed plugin. Being the first in having a complete implementation, it was run on the server before all other plugins and it ran for 56 hours before the team saw an error message pop up in the system log. This turned out to be a small mistake in having a field as required, were it actually was optional. This situation occurred as well with the other plugins, except for ClearlyDefined.
- *ClearlyDefined* has a well defined structure. This plugin has no optional fields and therefore after conferring with the client what fields were desirable to extract, didn't give any error before being deployed on the cluster.
- *Maven* also ran without any issues at all. Optional fields were assessed correctly from the beginning so no surprise exceptions came up during the dry run of this plugin.
- *NPM* turned out to be the plugin with the most difficulties. This was due to a number of different reasons:
  1. *Popularity vs polling interval* Npm is by far the most popular language of the created plugins, which can be noticed from the number of update events received in comparison to the other plugins; It's at least one order higher than the other plugins. NPM was initialized to fetch in near real-time by polling the found update stream every second. As a result the system log got flooded with error messages about http request failures and projects not being pulled. At first the team thought the polling interval was too short, but as it turned out after tweaking the update stream's up-time isn't consistent. The team believes this to be a problem at the side of the update stream's server. However, as discussed in subsection 4.2.2, this is no real issue.
  2. *Error:Not Found* Some packages turned out not to exist in the NPM registry. In these cases the update stream simply returned *"Error: not found"*. A simple check mitigates propagation of any raised exception on this specific package when polling the list of updated packages.
  3. *Unpublished packages* Other packages turned out to have a completely different structure than the standard package information in JSON format as required by the Node Package Manager. These packages raised Jackson (the JSON parser) exceptions. These packages however had the peculiarity that they contain a version field named "Unpublished". After discussion with the client these packages were omitted since they are of no interest.
  4. *NPM not enforcing structure* At some point all authors were unknown. This had to do with the fact that older versions of NPM allowed a field author of type `String` containing a name, optional email and optional url. In newer versions of NPM this is a complex JSON-object with a required name and optional email and url. The library for reading JSON values, called Jackson, isn't able to distinguish between String and complex objects. Therefore all authors got parsed as a String type, which in most cases wasn't filled. This resulted in a None value in the author field. After discussing with the client the team set out at first to manually parse the author in a union type, but having to check all possible variations for author String, quickly switched to parsing only the complex object as this field was used for nearly all new projects.

6.3.3. System testing results on Sallab cluster

Since the deployment on the team's private server had taken place before actual deployment on the client's server, most of the issues had already been ironed out. Cargo, ClearlyDefined, and Maven were running perfectly, but NPM had one issue.

NPM suffered from what can be called an ingestion time bug. As the update stream doesn't provide any time stamp whatsoever the team decided, after discussion with the client, to use ingestion time for the minimum releases and to update the time field by manually extracting it from the JSON package information.

This worked fine, until both stages were deployed on the server. Within minutes there were more than 2000 releases, which didn't make sense. This turned out to be a bug. Every release from the update stream got an ingestion timestamp and due to the way the update stream was handled every time the team polled the update stream, every package got recognized as being new. To properly identify new packages, the difference between consecutive polls is now taken instead.

7 Product Evaluation

In this chapter the final product is evaluated. This is done by firstly showing results of the plugins running in production. Secondly it is assessed if the design goals set in section 3.3 are met. Furthermore, the evaluations from SIG are examined and it is discussed how their feedback is incorporated. Finally an ethical evaluation of the product is given.

7.1. Deployment

All the new plugins were deployed and on TU Delft's clusters have been running for almost two weeks as of writing this document. For each plugin, results will be shown and discussed. The results are taken from Grafana which runs on TU Delft's server.

7.1.1. Cargo

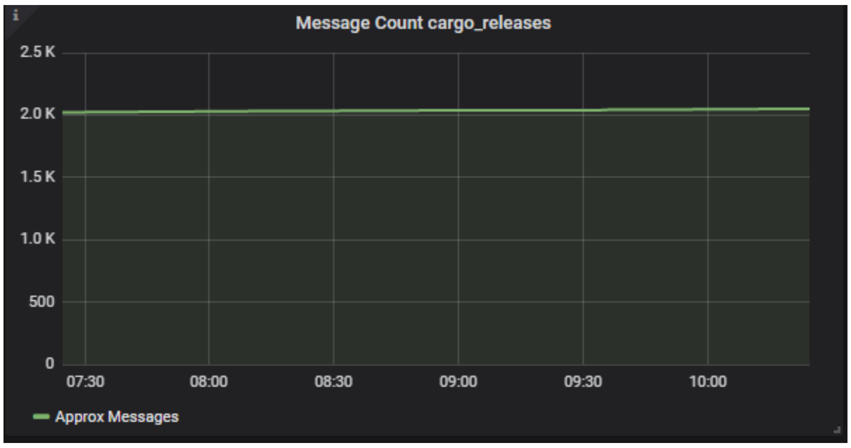


Figure 7.1: Results Cargo.

Cargo has one stage connected to the cargo_releases Kafka topic. As seen in Figure 7.1, the Cargo plugin has collected roughly 2000 projects in two weeks. These contain both updated and new packages. As there are only 35000 crates in total, 2000 is a substantial amount.

7.1.2. Clearly Defined

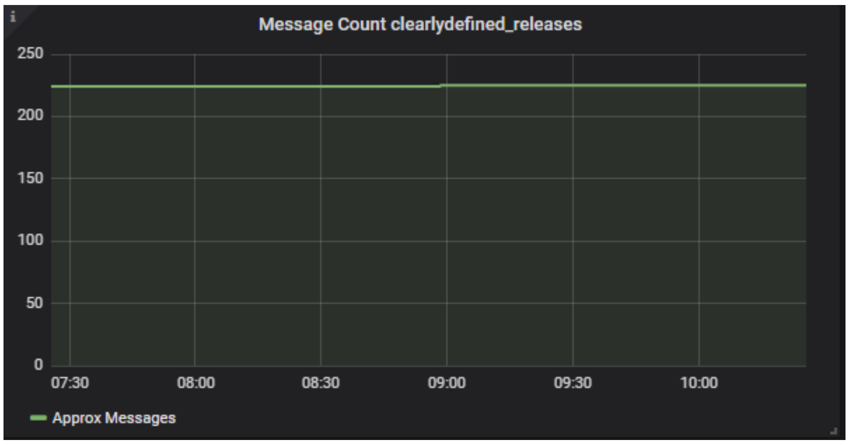
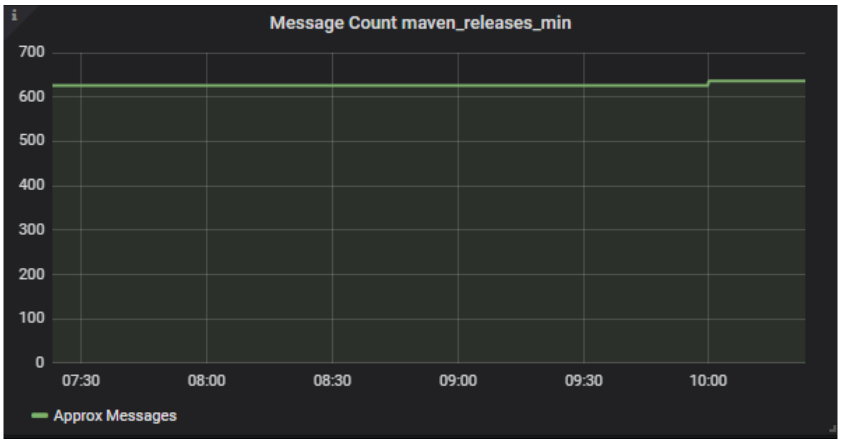


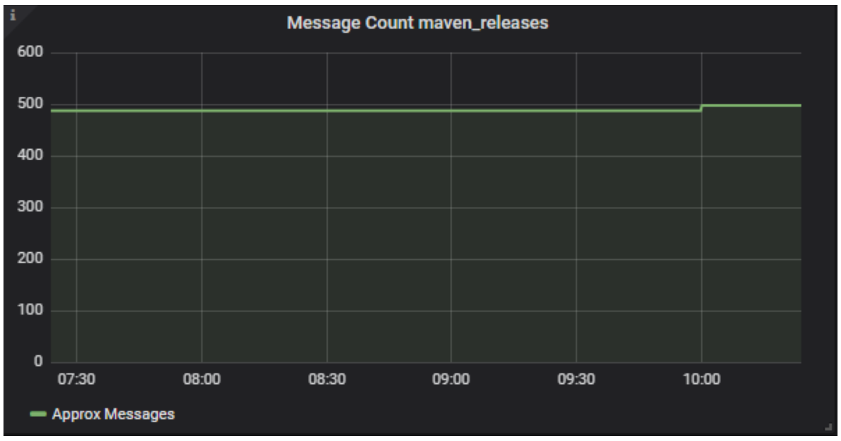
Figure 7.2: Results Clearly Defined

Releases of Clearly Defined projects are stored in the clearlydefined_releases Kafka topic. Figure 7.2 shows that there are approximately 225 Clearly Defined releases processed by the plugin. This is a relatively low amount compared to the other plugins. The reason for this is that there are few packages released on Clearly Defined.

7.1.3. Maven



(a) Results Maven minimal releases.

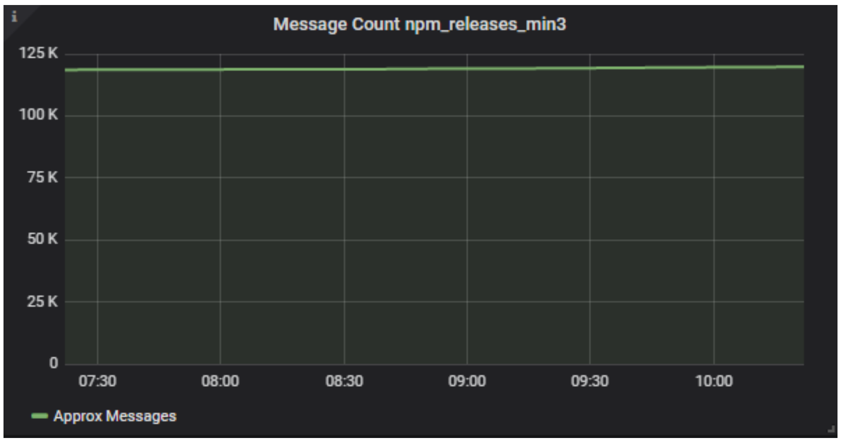


(b) Results Maven extended releases.

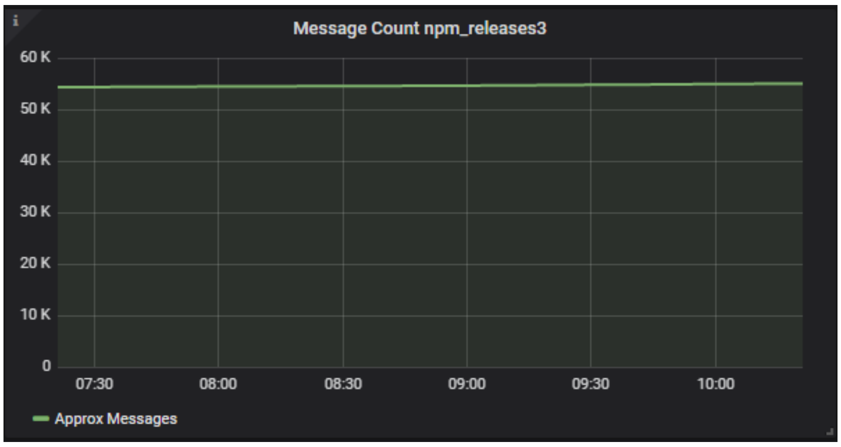
Figure 7.3: Results of the Maven plugin.

Maven is separated in two stages, therefore we got two different Kafka topics, *maven_releases_min* and *maven_releases*. Figure 7.3 illustrates that whenever maven_releases_min retrieves projects, maven_releases also fetches projects. The reason for this is that the second stage takes the output of the first stage as its input. The difference in total amount of projects between the two topics is caused by the fact that the maven_releases stage was temporarily stopped and could thus not retrieve any projects in that interval. As of writing this report, there have been 68000 jars published in 2020 according to the central maven repository [6]. This would average to roughly 100 jars per hour. However as seen in Figure 7.3, over a period of 3 hours, only about 10 projects have been collected. The reason for this is the incomplete update feed as discussed in subsection 4.2.1.

7.1.4. NPM



(a) Results NPM minimal releases.



(b) Results NPM extended releases.

Figure 7.4: Results of the NPM plugin.

The NPM plugin has collected the most packages by more than ten fold. Like Maven, NPM consists of two stages although unlike Maven, the number of collected projects differs greatly. This difference is caused by two reasons. The first reason is that the update stream contains unpublished projects. These unpublished projects are structured differently which would complicate fetching projects. However the client indicated he is not interested in unpublished projects, so the decision was made to not retrieve the information about unpublished projects. The second reason is that no project information is available in the NPM registry. Nevertheless, the project information of around 55000 projects was retrieved in two weeks time.

7.2. Software evaluation

This section evaluates whether or not the MoSCoW requirements and design goals are met and supports why this was or was not the case.

7.2.1. MoSCoW requirements

Requirement	**Met?**
Must haves:	
Plugin for Maven package manager	✓
Plugin for NPM (Node Package manager)	✓
Plugin for Cargo package manager	✓
Plugin for Clearly defined (licensing information)	✓
Read functionality of REPL	Partial
Evaluate functionality of REPL	Partial
Print functionality of REPL	✓
Terminal-based REPL	Partial
User-friendly setup	✓
Decent documentation for CodeFeedr	✓
Should haves:	
Plugin for Snyk.io	No
Plugin for web-based REPL	No
Help or Documentation	✓
Could haves:	No

Although not all of the must haves have been implemented in the way the team envisioned, CodeFeedr does have the ability to be queried using SQL. The team first set out to create a REPL, but after detailed discussion with the client found out that the pipeline creation as they had envisioned it was not in line with the clients' wishes. The client clearly indicated that CodeFeedr needed the ability to be queried using SQL, which it now does. Furthermore, the must-have requirements regarding the plugins have been properly met, as has the documentation regarding source code.

The REPL can cluster-wise be shown as a proof of concept. Users with limited technical background can still query the data sources of the TU Delft cluster. The technology behind streaming SQL and integration with Flink is still relatively new, so advances here will impact the ability of CodeFeedr to provide an SQL REPL greatly as well. However, the complication with evaluating the query were bugs the team was confronted with for multiple weeks without a proper solution. When attempting to deploy CodeFeedr using non-sequential pipelines via Kafka clusters rather than local system memory, an unknown factor added a Kafka sink to the staged execution plan, instantly 'cancelling' the evaulation of a data stream.

Exporting the project to a JAR in order to let users deploy the program with variables as their query also proved difficult, which is why the terminal-based REPL is only partially complete. While it is possible in an IDE-based terminal, as of writing the report it is impossible to run the program as a standalone.

Snyk.io did not allow derivative works unfortunately with its privacy policy, despite it being a wish from our client. The team agreed to skip this requirement due to having no other options. Other requirements, like the web-based REPL, were left out due to the time it took to deal with unforeseen impediments, such as the JAR-bug and the `startLocal`-bug. Sufficient documentation for users was supplied in the form of logical diagrams of all implemented plugins, on top of the source code documentation.

7.2.2. Design goals

This section reiterates the design goals set in section 3.3, checks if they have been attained or gives argumentation why this didn't happen.

- *Maintainability:* The first evaluation by SIG showed a market average maintainability upon which could be improved by tackling unit size and duplication. These two points have been tackled, test code has been created for newly added or changed code so the team is pretty confident the 2nd evaluation by SIG will be positive on this point.
- *Usability:* The biggest issue for usability according to the team was the lacking documentation. Changing existing documentation was indicated to be out of scope by one of the Bachelor Project Coordinators so the team didn't focus on it. But to improve usability for the plugins a `readme.md` in Markdown

notation has been added, describing what the plugins do, how to set them up and how to get data out of them. This improves usability.

- *Scalability & Performance:* The final product is positively scalable, as it has been deployed on a kafka cluster and no problems have been encountered with retrieving packages and processing them into extended releases in appended stages. Performance-wise the product is optimized in such a way that, depending on the plugin, the polling intervals are short enough not to skip packages, and long enough not to strain the network too much. Much of the work is done during initialization, such as the registering of StreamTables, and since CodeFeedr is a product designed to be turned on once and left running, a large chunk of the program is unimportant regarding performance.
- *Security:* As this product will be supplied to a single user only, or users within the same academic facility, the precautions developers must take when distributing software to the wide public aren't as high priority as otherwise. Furthermore, as this software in essence collects data and inserts the data into Kafka topics based on a query, SQL injections are not a concern due to the streaming nature.
- *Ethics:* The client clearly indicated the desire for as much information possible, including publicly available data on authors. The team advises to tackle violating GDPR by following the same steps the client undertook for GHTorrent as described in subsection 7.4.2

7.3. SIG evaluation

During the course of the project, SIG evaluated the code at two occasions, firstly in week six, and secondly in week nine. The aim of these evaluations is to identify points of improvement in terms of code quality. Furthermore, it will be examined if the feedback of the first evaluation, is implemented before the second evaluation.

7.3.1. First evaluation

The first evaluation scored 3.7 out of 5 stars which indicates that the code has a maintainability of the market average. The main areas of improvements were code duplication and unit size. Code duplication is bad for maintainability because having more code, means there is more code to maintain. Furthermore, whenever the duplicate code is changed, it needs to be changed at several places. If one those places is missed, bugs could arise. The following classes were mentioned as having duplicate code:

- CargoReleasesSource.scala - ClearlyDefinedReleasesSource.scala
- getNewCratesFromSummary - getUpdatedCratesFromSummary

A big unit size indicates pieces of code which have a larger than average size, e.g. a long method. Larger unit sizes can reveal that a method has more than one task. Smaller unit are thus more desirable since each method will only task, and all methods are automatically documented by their method names. Methods which had a big unit size were:

- NpmService.getProject
- Protocol.MavenProjectPojo.fromMavenProject
- RetrieveProjectAsync.asyncInvoke

It was furthermore noted that the presence of testcode early on in the development stages was promising, along with the recommendation to keep writing new tests when adding new code.

7.3.2. Second evaluation

Before the second evaluation, other than adding new code, the code which was sent to SIG in the first evaluation was refactored according to SIG's feedback. To eliminate code duplication, a abstract parent class *PluginReleasesSoruce* was created which contains all functionality which was shared between the plugins. The mentioned methods *getNewCratesFromSummary* and *getUpdatedCratesFromSummary* were also refactored into a single method. Furthermore, all methods which were indicated to be too long were split into two or more smaller methods to both reduce method length and to split functionality.

The received second evaluation of SIG confirms these findings. The feedback indicates a grown codebase with increased maintainability. *Duplication* was tackled well, as was *Unit size*, but due to newly added methods, this point was partly undone. SIG praised the amount of test code added for new production code, as testing usually gets less attention when a project progresses, but it was positive that was not the case during our project. SIG concludes by stating their recommendations have been included during development.

7.4. Ethical Evaluation

Aside from the usual evaluation based on the wishes of the client, the design goals & design choices, a broader view on the deployment and use of CodeFeedr is necessary. This section will describe ethical implications of CodeFeedr, discuss possible concerns, and recommend ways to help remedy these concerns.

7.4.1. Publicly Available Data versus Personal Privacy

The rise of online social networks in different areas (LinkedIn, Facebook, Github) and the emergence of big data processing has impacted personal privacy in a big way. The amount of data that can be collected, the speed at which this can be done and the duration it can be retained as well as the kind of information that can be gathered and acquired is at a scale never seen before [39]

7.4.2. Privacy related concerns

Our client has had previous experiences pertaining to ethics with another MSR-tool, called GHTorrent. GHTorrent is a tool to monitor the Github public timeline and for each event, scrape the content and dependencies and store these. One of the issues of GHTorrent in particular, issue 32, raised concerns with Github developers about their privacy [25]. Github, an online collaboration platform for development, has the property that email addresses are published. It turned out that after aggregation and redistribution by GHTorrent, developers were the target of unwanted emails with survey requests and profiling. Developers wanted their emails deleted and this created a very heated discussion between developers of GHTorrent and Github users. It even got attention from renowned IT Lawyer Arnoud Engelfriet, who mentioned it on his blog [20] and popular law & IT forum [21]. To remedy the situation, the client choose to employ an opt-out mechanism.

7.4.2.1. Application to CodeFeedr

CodeFeedr could face similar issues, since it is processing open source data concerning users, such as names, email addresses and links to (personal) websites. The data of for example a package of Node package manager (NPM) is publicly available, so collecting this information on these packages will consist of collecting personal data as well.

To prevent negative consequences like for example spam or recruiters judging future hired developers on their personal data, the EU has deployed the General Data Protection Regulation (GDPR) [5]. These comprise, among others, a strict set of guidelines to which a controller of personal data must adhere. Two of those are the right to be forgotten (so having their personal data removed) and the right to alter their own personal data. The GDPR also does not allow processing of personal data simply using the guise of *"doing scientific research"* but does allow the processing if the controller can demonstrate an urgent need for this data. In the case of CodeFeedr we can claim that even old data is relevant so the right to be forgotten does not prevail over this claim. Complicating things even more is that CodeFeedr employs Kafka, an append only log for real time stream processing. This means that alterations to this log are nearly impossible. Using an opt-out isn't a useful option to implement. There are two reasons for this. Apart from the technical difficulty of having to change many lines in an append only stream per user request, there is another complication which completely negates the right to be forgotten. Although we can remove personal data from CodeFeedr, it still is possible to retrace an event to a user, at least for example with NPM. Having only the id and version of the project is enough to query the NPM repository for this specific data. Removing personal data means someone with less well intentions has to do an extra step, but it is possible nonetheless. The only way to anonymise the data is to anonymise the project id as well, but that will hinder the presentation of scientific results enormously.

How to solve this dilemma is still unclear, but the GDPR states that an organisation or controller of this personal data has to do everything in its power to make this available as much as possible. For CodeFeedr we would recommend to clearly review which data of interest is necessary for scientific research. Use special Kafka topics for internal use and if CodeFeedr is published along with certain Kafka topics as public data, it should be ensured that these adhere to the GDPR by inspected the privacy concerns of each data field.

8 Process Evaluation

This chapter evaluates the development methodology used during the BEP. Firstly the scrum method is discussed, followed by an evaluation of the meetings with both our client and our coach. Lastly personal evaluations of each of the team members are given.

8.1. Scrum

During the project an agile approach was taken with sprints of one week. This was experienced to be suiting to the project as it allowed for enough evaluation moments to assess whether the team was still on schedule. Although the sprint reviews were not too extensive, all important points of the previous week were covered and each member gave his opinion on both his personal progress and the progress which was made as a team.

Sprint boards were set out to be tracked in Asana, however during the project it became clear that ZenHub was more flexible to use. ZenHub is a plugin for Github which allows for the tracking of several Github project boards in one central place. Furthermore, by using Zenhub, PRs could be linked to issues. The tracking of the sprint boards was sometimes mediocre as the team would start a sprint, and sometimes update the board only at the end of the sprint. This would make it unclear how the sprint was progressing when looking at the sprint board. This was however not a big problem for the team, as all members were working daily in the same room. Therefore there was enough communication within the team to keep each other updated on the progress.

8.2. Client meetings

Client meetings took place once every week or once every two weeks depending on the need for feedback. Our client Georgios Gousios would give feedback on new features and the team could ask questions on how to implement new features for the next sprint(s).

8.3. Coach meetings

Meetings with our coach Dr. A. Katsifodimos, occurred once every two weeks. During these meetings newly implemented features were shown to which feedback was given. When the team ran into problems, A. Katsifodimos would give advice on how to tackle these problems. Overall these meetings were appreciated by the team and experienced as very helpful.

8.4. Personal evaluations

This section gives personal evaluations of the project as a whole. In these evaluations, each member will reflect on what he found challenging during the project and what went well. Each member will also give his opinion of the result of the project.

8.4.1. Roald van der Heijden

What I really am glad about is that as a team we were able to output a decent set of useful plugins for our client. That, intra team communication and the reports went well, which is something I'm very proud of. I hope my team appreciates the extras I tried to take care of to improve the process (the rooms, our own deployment test server and communication to external parties).

What do I feel could be improved? Just before writing this evaluation I read a really nice quote: "Don't stop when you're tired, stop when you're done." I found this applicable since Asterios Katsifodimos helped me deal with this. Taking a rest when I needed it and performing better during the day is something which sounds natural but can be lost in the crowdedness of every day life. Unfortunately at the time of writing this report, we're still not where we want to be, but seeing that the technology used is relatively new, the framework distributed and the documentation at times really bad, I can only look at that quote and just think: We're not done.

Another thing that has astounded me is to see how much time and effort real life deployment costs in order to run really smooth. This was something I didn't expect with my current set of learned skills. Learning by doing can mitigate this and I hope to do a lot of both of them in the near future.

8.4.2. Matthijs van Wijngaarden

I personally joined the 1Up group a week later than the rest and vowed to work extra hard to make up for the lost time. The progress done before my entrance was mostly the research report, so I had the chance to dive right into the codebase. I am quite happy with what we have achieved in the following weeks as a team. The work on the Plugins were certainly a small challenge at first, but once we realized the gist of it, it was quite doable. By implementing design patterns, collaborating through peer programming and every now and then even attempt some test driven development, all in combination of an attempted agile scrum environment, this project definitely felt like a personal development experience for me.

When the second half of the project came around the corner however, bugs seemed to plague us. While certainly less enjoyable than the first half due to lower output, this also helped me develop a stronger work ethic towards situations which could be deemed frustrated. Although in the end I'm not too satisfied with the final product, as there are still some nasty bugs I would have loved to have solved, I think that what we now deliver, combined with the process and teamwork behind the scenes is something I can be proud of.

8.4.3. Wouter Zonneveld

The project started out with a research phase. Since I took the course Bachelor Seminar last period, this was a rather familiar process. Because of this the research phase went smooth and without any major issues. The implementation phase of the project was more challenging. This was because instead of starting from an empty project, which is usually the case, we had to build upon an existing project, namely CodeFeedr. Since CodeFeedr is a project with many external dependencies, it was hard to fully grasp how all the different parts work together. Although Wouter Zorgdrager gave us a high level explanation on how CodeFeedr works at the start of the project, many of the details only became clear during the course of the project. Furthermore, I personally spent a considerable amount of time on the streaming SQL parts which I found challenging. This is mostly because the streaming SQL dependencies are still in early stages of development. As a result, documentation is limited and code extracts which are available are often in older versions and no longer work.

Overall I am semi-satisfied with the result of the project. Mainly I would have liked to been able to export the project as a jar, and to have the SQL stages working with Kafka. However the root of both of these issues seemed to be in external dependencies which made them hard for us to solve.

9 Conclusion

Currently, CodeFeedr has implemented four additional plugins to be able to be used on the pipeline. Each one of them is able to process incoming packages in a streaming fashion, and output them in a custom output stage; Whether that is a JSON stage or an SQL stage is up to the user. Furthermore, the 1Up development team has introduced data structures to the CodeFeedr product. Incoming data streams of the implemented plugins are now translated into table structures, enabling querying for the REPL.

However, since only seven weeks of the project could be spent on programming, there are many features which are not yet implemented but would make CodeFeedr a better product. Firstly if the project could successfully be exported to a jar, the project would become more accessible. Furthermore, the easiest way to run something on the TU Delft cluster is to upload and then run this jar. Another high priority addition would be to fix the bug in the interaction between streaming SQL and Kafka described in subsection 5.2.1. Solving this bug would turn the SQL REPL from a proof of concept to a production ready product.

A web-based REPL would be a great addition to enhance the usability of CodeFeedr. A user could connect to a website, which is connected to the TU Delft cluster, input a query and monitor the result. This would circumvent the process of connecting to the TU Delft cluster and running a jar with input arguments.

More plugins for CodeFeedr would be another addition to increase CodeFeedr's value. For example, tracking information such as Jira or Bugzilla would be beneficial plugins. Code review information such as Gerrit would be valuable to have in a plugin as well. Package managers are another type of plugin which are useful for CodeFeedr. Currently Pypi, Maven, Clearly Defined, Cargo, and NPM are implemented, but many more software repository providers exist. Examples of interesting package managers are Go, NuGet, Packagist, RubyGems, and Bower.

Appendices

A
Project description

In this chapter, the problem description from Project Forum (formerly BEPSys, the Bachelor Project administration tool) is given.

Project title
A REPL loop for the CodeFeedr project (Offered by Software Analytics Lab - TU Delft)

Project description
Codefeedr is a platform for streaming (software) analytics. It is built upon Apache Flink, a stream processing platform. Codefeedr allows users to build pipelines (DAGs) of stream processing steps, by combining smaller steps and sharing data inputs and outputs. Using those pipelines, one can express stream processing queries, such as “find me the developer that wrote the line that crashes for all stacktraces that show up more than 5 times per second”. The pipeline abstraction, while useful requires the user to write the program in Scala, compile it and deploy it on the Flink cluster. This is very inconvenient when we just want to check a simple query. A better way to do such queries is an SQL REPL loop. In such a scenario, the SQL REPL loop compiles a user-provided query down to a CodeFeedr pipeline, deploys it and reads the results back to the user console.

Several components required for building a Codefeedr REPL loop exist: SQL parsing is handled by Apache Calcite; Codefeedr automatically exposes live data types to a centralized location; a way to deploy a Codefeedr program on a Flink cluster is part of Codefeedr. The purpose is to combine those existing components in a high-quality engineering project; the end result will be open source and of high value to the stream processing community. Besides the SQL REPL loop, SERG would like to see the following improvements to CodeFeedr: Connecting more streaming sources, connecting more historic sources, improve upon its architecture and improve setup & documentation.

B
Project Info Sheet

Title of the project: A REPL for CodeFeedr
Name of the client organization: Software Analytics LAB (part of SERG TU Delft)
Presentation Date: 5th February 2020
Final report: Can be found here

- **Description:** This project aims to expand upon the current state of CodeFeedr, a Mining Software Repository (MSR) tool for performing realtime streaming analytics on publicly available package information. Goals are to add more streaming sources and to build a REPL to let researchers query data sources using Streaming SQL.
- **Challenge:** Integrating several existing systems, namely CodeFeedr, Flink SQL, and Kafka.
- **Research:** Focused on which streaming sources were important, design goals and MoSCoW requirements needed for the updated version and the choice of Streaming SQL framework to employ.
- **Process:** Scrum with iterations of one week were used. Test Driven Development was not fully employed, but the team focused on paying technical debt by extensive testing.
- **Product:** CodeFeedr has four new plugins running on TU Delft's Sallab's server cluster with positive results. The SQL REPL only works for sequential pipelines running on a local computer.
- **Outlook:** The plugins are thoroughly tested before deployment but could be improved by developing side output for debugging inspections. Following development of Streaming SQL in Flink could be useful in getting the REPL to work with non-sequential pipelines and a clustered setting.

Team Roles:

Roald van der Heijden, *Contributions:* Project Developer: NPM plugin, Process & Deployment Testing
Matthijs van Wijngaarden, *Contributions:* Project Developer: Cargo & Clearly Defined plugins, SQL REPL
Wouter Zonneveld, *Contributions:* Project Developer: Maven plugin, SQL REPL

Client, Coach

Name and affiliation of the client: Georgios Gousios, Software Engineering Research Group (SERG) TU Delft
Name and affiliation of the coach: Asterios Katsifodios, Web Information Systems, TU Delft

Contact

Georgios Gousios, g.gousios@tudelft.nl

C
Overview SQL Tables of CodeFeedr Plugins

This appendix contains an overview of all tables used for the SQL Stage of each written plugin by team 1Up.

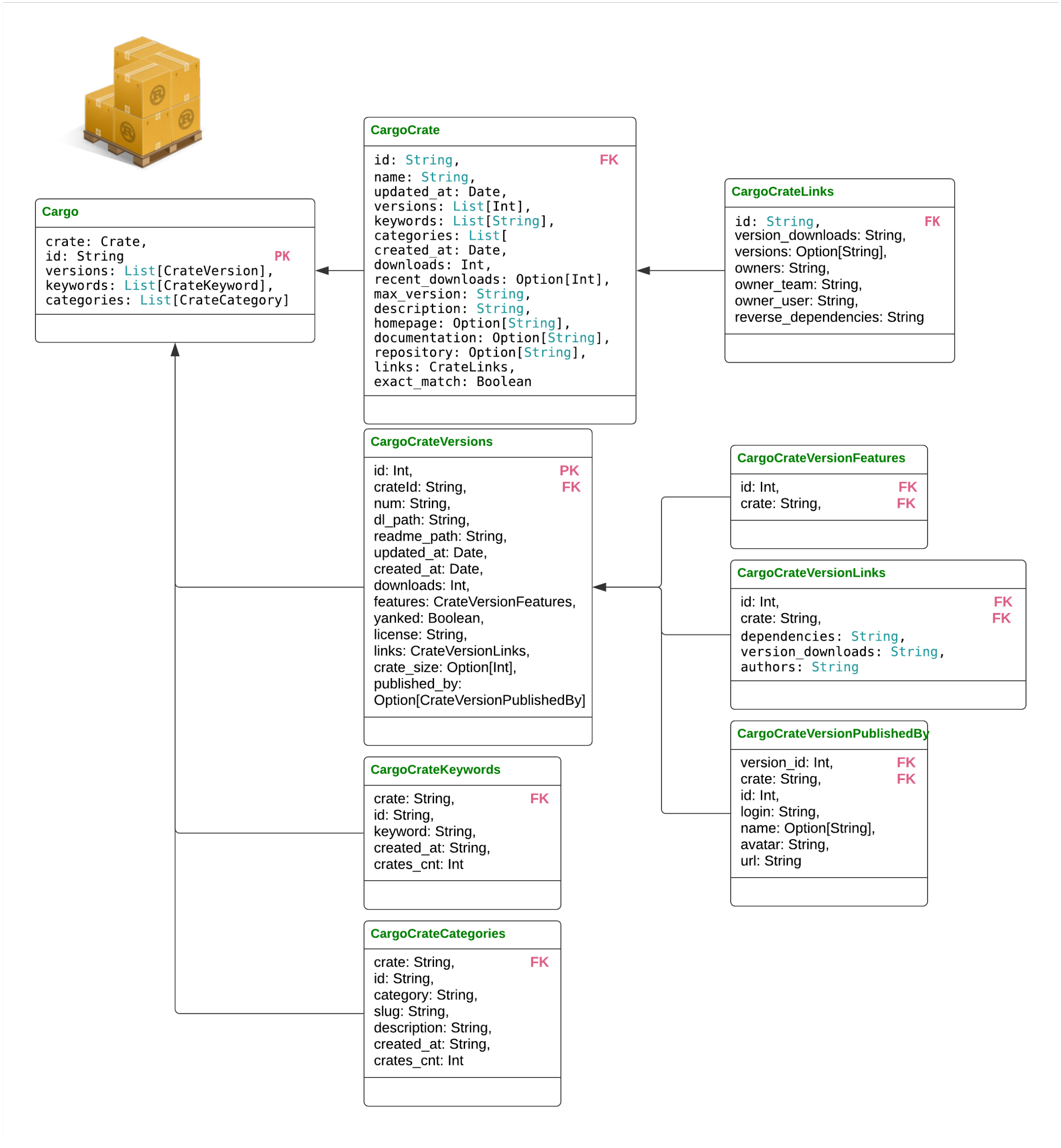


Figure C.1: Overview of table structure for the Cargo plugin

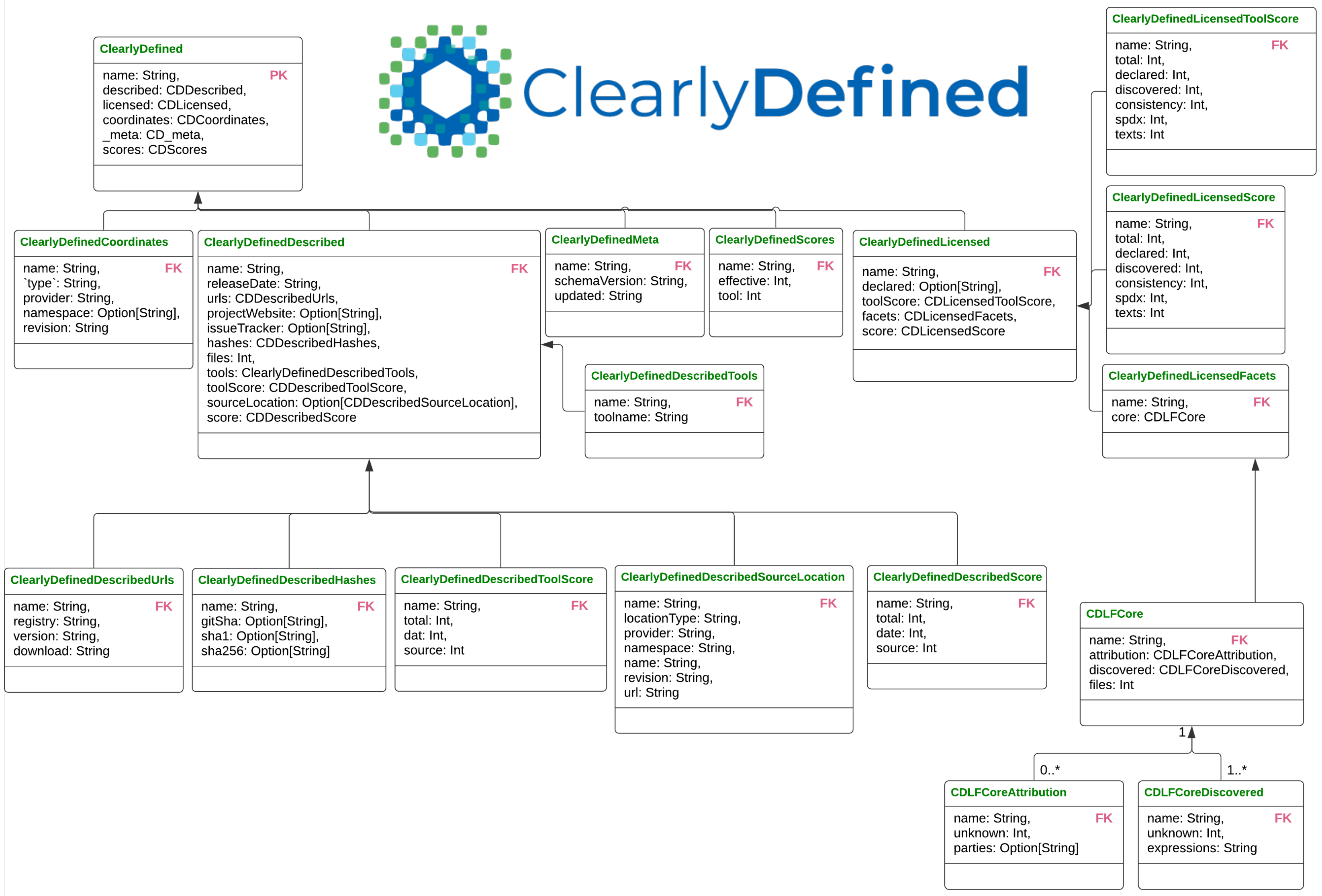


Figure C.2: Overview of table structure for the ClearlyDefined plugin

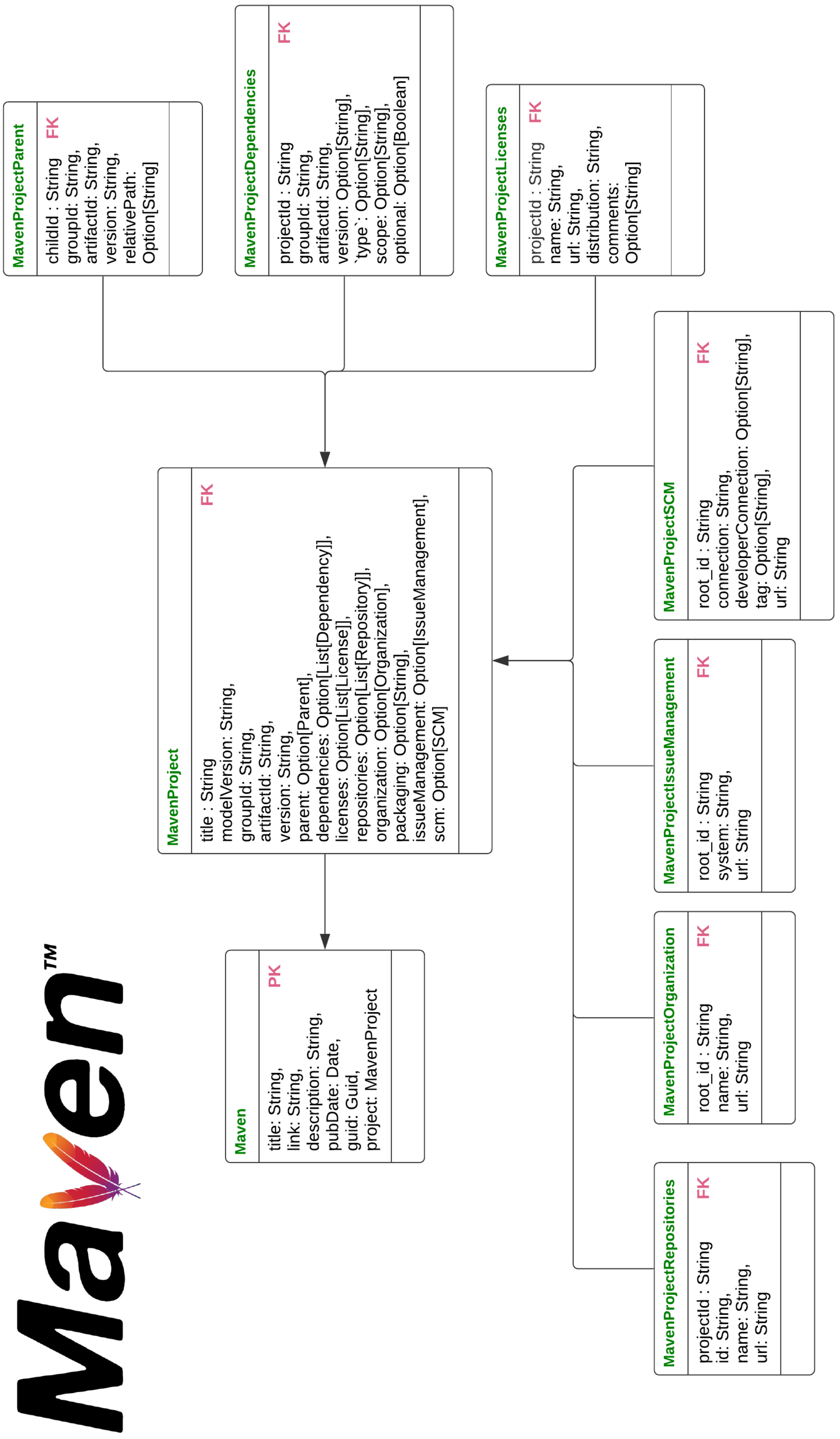


Figure C.3: Overview of table structure for the Maven plugin

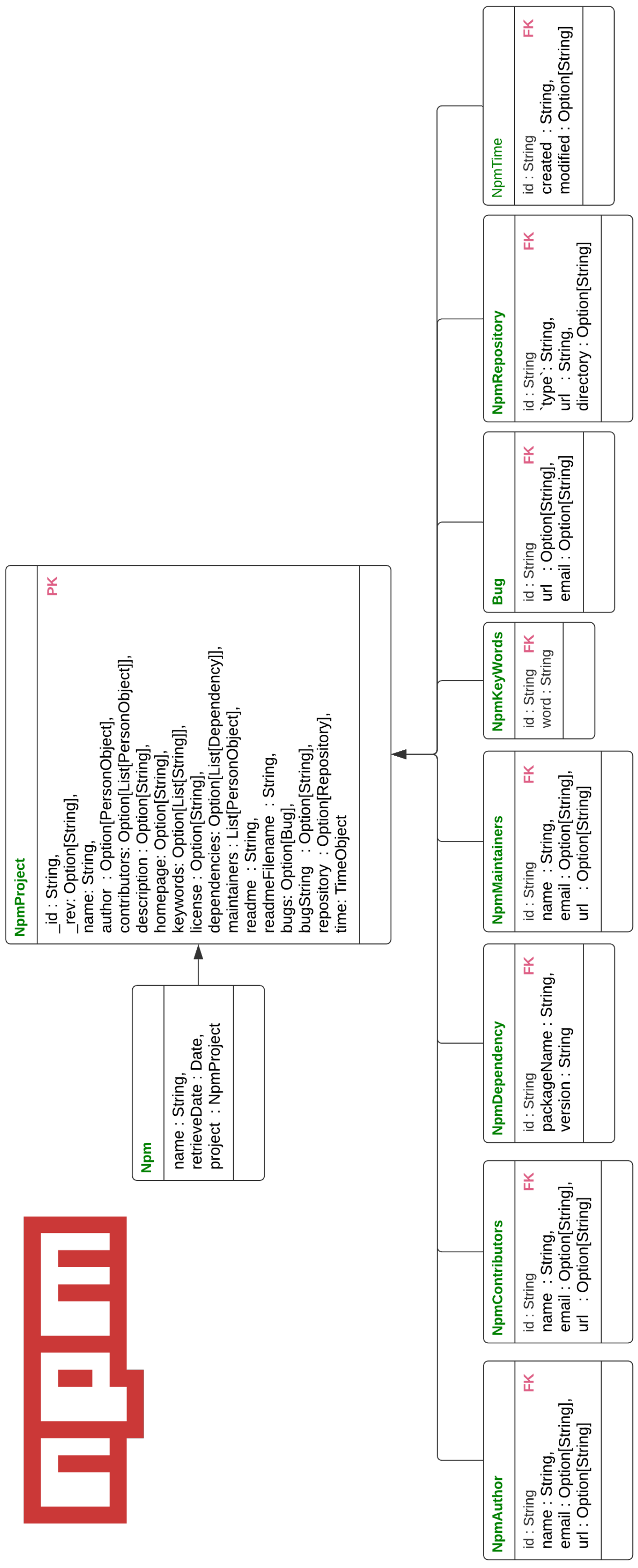


Figure C.4: Overview of table structure for the NPM plugin

D Feedback SIG

D.1. 1st Evaluation moment: week 2.6

De code van het systeem scoort 3.7 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Duplication en Unit Size vanwege de lagere deelscores als mogelijke verbeterpunten.

Bij Duplication wordt gekeken naar de hoeveelheid gedupliceerde code. We kijken hierbij ook naar de hoeveelheid redundantie, dus een duplicaat met tien kopieën zal voor de score sterker meetellen dan een duplicaat met twee kopieën. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om de hoeveelheid gedupliceerde code zo laag mogelijk te houden. Na verloop van tijd zal de gedupliceerde code moeten worden aangepast. Dit leidt niet alleen tot extra werk, aangezien op dat moment alle kopieën tegelijk moeten worden veranderd, maar is ook foutgevoelig omdat de kans bestaat dat één van de kopieën per ongeluk wordt vergeten.

Voorbeelden in jullie project:

- CargoReleasesSource.scala versus ClearlyDefinedReleasesSource.scala
- JsonParser.scala (getNewCratesFromSummary versus getUpdatedCratesFromSummary)

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- NpmService.getProject
- Protocol.MavenProjectPojo.fromMavenProject
- RetrieveProjectAsync.asyncInvoke

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

D.2. 2nd Evaluation moment: week 2.9

[Hermeting]

In de tweede upload zien we dat het codevolume is gegroeid, terwijl de score voor onderhoudbaarheid is gestegen. De grootste verbetering op het gebied van deelscores zit bij Duplication, wat in de feedback op de eerste upload ook als één van de verbeterpunten werd genoemd. We zien dat jullie bij het andere aandachtsgebied, Unit Size, ook maatregelen hebben genomen. Die zijn echter weer deels ongedaan gemaakt omdat in de nieuwe code ook weer lange methodes zijn toegevoegd.

Op het gebied van testcode zien we dat jullie bijna net zoveel nieuwe testcode als nieuwe productiecode hebben geschreven. Dat is positief, meestal zie je dat unit testen minder aandacht krijgt naarmate een project loopt.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.

Bibliography

[1] Rui Abreu, Hakan Erdogmus, and Alexandre Perez. Codeaware: Sensor-based fine-grained monitoring and management of software artifacts. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2:551–554, 2015.

[2] Alooma. Alooma | enterprise data pipeline platform. `https://www.alooma.com/`, 2019. Accessed: 2019-11-30.

[3] Unknown author. Amazon kinesis. `https://aws.amazon.com/kinesis/`, 2019. Accessed: 2019-11-30.

[4] Unknown author. `http://codefeedr.org/`, 2020. Accessed: 2020-01-29.

[5] Unknown author. `https://en.wikipedia.org/wiki/General_Data_Protection_Regulation`, 2020. Accessed: 2020-01-29.

[6] Unknown author. `https://mvnrepository.com/repos/central`, 2020. Accessed: 2020-01-25.

[7] Unknown author. `https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/common.html`, 2020. Accessed: 2020-01-29.

[8] Unknown author. `https://libraries.io/data`, 2020. Accessed: 2020-01-29.

[9] Unknown author. `https://2020.msrconf.org/`, 2020. Accessed: 2020-01-29.

[10] Unknown author. `https://www.sisense.com/glossary/real-time-analytics/`, 2020. Accessed: 2020-01-29.

[11] Apache Calcite. Streaming. `https://calcite.apache.org/docs/stream.html#joining-streams-to-tables`, 2019. Accessed: 2019-11-29.

[12] Cargo. crates.io: Rust package registry. `https://crates.io/`, 2019. Accessed: 2019-11-29.

[13] ClearlyDefined. `https://clearlydefined.io/about`, 2019. Accessed: 2019-11-29.

[14] Confluent. Ksql: Streaming sql for apache kafka | confluent. `https://www.confluent.io/product/ksql/`, 2019. Accessed: 2019-11-30.

[15] Confluent. Join event streams with ksql - confluent platform. `https://docs.confluent.io/current/ksql/docs/developer-guide/join-streams-and-tables.html`, 2020. Accessed: 2020-01-07.

[16] Jacek Czerwonka, Nachi Nagappan, Wolfram Schulte, and Brendan Murphy. Codemine: Building a software development data analytics platform at microsoft. *IEEE Software*, July 2013. URL `https://www.microsoft.com/en-us/research/publication/codemine-building-a-software-development-data-analytics-platform-at-microsoft/`.

[17] Janet Davis and Lisa P Nathan. Value sensitive design: Applications, adaptations, and critiques. *Handbook of ethics, values, and technological design: Sources, theory, values and application domains*, pages 11–40, 2015.

[18] SERG TU Delft. Software analytics. `https://se.ewi.tudelft.nl/softanalytics.html`, 2019. Accessed: 2019-11-18.

[19] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015. ISBN 0133970779.

[20] Arnoud Engelfriet. `https://blog.iusmentis.com/2016/02/29/mag-ghtorrent-openbare-data-github-aggregeren-als-onderzoeksdataset/`, 2016. Accessed: 2020-01-25.

[21] Arnoud Engelfriet. `https://legalict.com/2016/02/28/is-it-legal-for-ghtorrent-to-aggregate-github-user-data/`, 2016. Accessed: 2020-01-25.

[22] The Apache Software Foundation. `https://flink.apache.org/`, 2019. Accessed: 2020-01-25.

[23] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[24] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL `http://dl.acm.org/citation.cfm?id=2487085.2487132`.

[25] Georgios Gousios. The issue 32 incident – an update. `http://gousios.org/blog/Issue-thirty-two.html`, 2016. Accessed: 2020-01-19.

[26] Ahmed E Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57. IEEE, 2008.

[27] P Carbone Asterios Katsifodimos, S Ewen Volker Markl, and S Haridi Kostas Tzoumas. Apache flinktm: Stream and batch processing in a single engine. *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng*, 36(4), 2015.

[28] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 01 1984. ISSN 0010-4620. doi: 10.1093/comjnl/27.2.97. URL `https://doi.org/10.1093/comjnl/27.2.97`.

[29] Quist Joris Kuijpers, Jos and Wouter Zorgdrager. Codefeedr, connecting streaming jobs. Technical report, Delft University of Technology, 2018.

[30] Thayne McCombs. `https://www.lucidchart.com/techblog/2016/07/12/introducing-xtract-a-new-xml-deserialization-library-for-scala/`, 2016. Accessed: 2020-01-29.

[31] Samza. Samza- samza sql. `https://samza.apache.org/learn/documentation/latest/api/samza-sql.html`, 2020. Accessed: 2020-01-28.

[32] Alexander Shvets. Adapter design pattern. `https://sourcemaking.com/design_patterns/adapter`, 2020. Accessed: 2020-01-28.

[33] Alexander Shvets. Singleton design pattern. `https://sourcemaking.com/design_patterns/singleton`, 2020. Accessed: 2020-01-28.

[34] Alexander Shvets. Strategy design pattern. `https://sourcemaking.com/design_patterns/strategy`, 2020. Accessed: 2020-01-28.

[35] Harry M. Sneed. Dealing with technical debt in agile development projects. In Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann, editors, *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, pages 48–62, Cham, 2014. Springer International Publishing. ISBN 978-3-319-03602-1.

[36] SQLstream. Sqlstream | streaming sql analytics for kafka & kinesis. `https://sqlstream.com/`, 2019. Accessed: 2019-12-03.

[37] SQLstream. Capabilities - streaming analytics platform | sqlstream. `https://sqlstream.com/capabilities/`, 2019. Accessed: 2019-12-03.

[38] Rod Stephens. *Beginning Software Engineering.* Wrox Press Ltd., GBR, 1st edition, 2015. ISBN 1118969146.

[39] Herman T. Tavani. *Ethics and Technology: Controversies, Questions, and Strategies for Ethical Computing.* Wiley Publishing, 4th edition, 2012. ISBN 1118281721.

[40] Enrique Larios Vargas, Joseph Hejderup, Maria Kechagia, Magiel Bruntink, and Georgios Gousios. Enabling real-time feedback in software engineering. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 21–24. ACM, 2018.

[41] Wouter Zorgdrager. codefeedr/codefeedr-plugin-template.g8: This is the official (giter) template to create your own codefeedr plugin. `https://github.com/codefeedr/codefeedr-plugin-template.g8`, 2019. Accessed: 2019-11-18.